ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Μορφές Αποθήκευσης Εικονικών Δίσκων για την Αποδοτική Υποστήριξη Κλώνων, Στιγμιοτύπων και Απαλοιφής Διπλοτύπων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Εμμανουήλ Χ. Ανδρουλιδάκης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2016

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Μορφές Αποθήκευσης Εικονικών Δίσκων για την Αποδοτική Υποστήριξη Κλώνων, Στιγμιοτύπων και Απαλοιφής Διπλοτύπων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Εμμανουήλ Χ. Ανδρουλιδάκης

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Οκτωβρίου 2016.

....	....	....
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αν. Καθηγητής ΕΜΠ	Επ. Καθηγητής ΕΜΠ

Αθήνα, Οκτώβριος 2016

. . . .

Εμμανουήλ Χ. Ανδρουλιδάκης
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

Copyright © Εμμανουήλ Χ. Ανδρουλιδάκης, 2016
Με επιφύλαξη παντός δικαιώματος. All rights reserved.
Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περιεχόμενα

Περίληψη viii

Extended Abstract x

Αντί Προλόγου xi

Extended Abstract xiii

0.1 Introduction xiii
0.2 Backround xiv
0.2.1 Storage xiv
0.2.2 Virtuaization xvii
0.3 Design and Analysis of Virtual Disk Formats xix
0.3.1 Space Efficiency techniques xix
0.3.2 Case example: qcow2 xxvii
0.3.3 Problem Statement and Shortcomings xxix
0.3.4 A first approach: Flat recipe xxx
0.3.5 Our Design: Tree recipe xxxi
0.3.6 Theoretical evaluation and performance model xxxiv
0.4 Implementation xxxvi
0.4.1 Block driver and QEMU integration xxxvii
0.4.2 Caches and Flush xxxviii
0.4.3 Concurrency and locks xl
0.5 Experimental evaluation xlii

0.5.1 Micro-Benchmarks xliii
0.5.2 Metadata xlvi
0.5.3 Macro-Benchmarks l
0.5.4 Conclusion and future directions li

1 Εισαγωγή **1**
1.1 Διατύπωση του προβλήματος 1
1.2 Κίνητρα 1
1.3 Ελλείψεις 2
1.4 Σχεδιασμός 2
1.5 Αποτελέσματα 3

2 Υπόβαθρο **5**
2.1 Αποθήκευση δεδομένων 5
2.1.1 Μέσα και τεχνολογίες αποθήκευσης 5
2.1.2 Συστήματα αρχείων 8
2.1.3 Στοίβα Ε/Ε 14
2.1.4 Εναλλακτικές αρχιτεκτονικές αποθήκευσης 21
2.2 Εικονοποίηση και εικονικοί δίσκοι 24
2.2.1 Εικονικοποίηση 24
2.2.2 Υπολογιστικά περιβάλλοντα νέφους 28
2.2.3 Εικονικοί Δίσκοι 29
2.2.4 Στιγμιότυπα και κλώνοι 33

3 Σχεδιασμών και ανάλυση εικονικών δίσκων **35**
3.1 Τεχνικές εξοικονόμησης αποθηκευτικού Χώρου 35
3.1.1 Αραιά αρχεία 35
3.1.2 COW, στιγμιότυπα και BTRFS 39
3.1.3 Απαλοιφή διπλοτύπων 47
3.2 Παράδειγμα μορφής εικονικού δίσκου: qcow2 64
3.3 Παρουσίαση του προβλήματος και των δυσκολιών του 70
3.4 Ελλείψεις 75
3.5 Μια πρώτη προσέγγιση: Επίπεδη Συνταγή 79
3.6 Ο σχεδιασμός μας: Δενδρική Συνταγή 80
3.7 Θεωρητική αξιολόγηση 95
3.8 Μοντέλο Απόδοσης 101

4 Υλοποίηση οδηγού εικονικού δίσκου **109**
4.1 Πυρήνας υλοποίησης: οντότητες ως αρχεία 109
4.2 Οδηγός εικονικού δίσκου QEMU 117
4.3 Κρυφές μνήμες και συγχρονισμός με δίσκο 126
4.4 Κλειδώματα και ταυτοχρονισμός 133
4.5 Περιγραφή βασικών λειτουργιών 145

5 Πειραματική Αξιολόγηση **149**
5.1 Ρύθμιση πλατφόρμας μετρήσεων 149
5.2 Micro-Benchmarks 153
5.2.1 Μέγεθος τεμαχίου 154
5.2.2 Αριθμός επιπέδων 158
5.2.3 Χρόνος λήψης στιγμιοτύπου 164
5.3 Μεταδεδομένα 165
5.3.1 Υπολογισμός μεταδεδομένων 166
5.3.2 Εξοικονόμηση μεταδεδομένων 171
5.4 Macro-Benchmarks 179

6 Συμπεράσματα και μελλοντικές δυνατότητες **185**

Βιβλιογραφία **193**

Περίληψη

Η παρούσα εργασία στόχο έχει να προτείνει μια μορφή εικονικού δίσκου, η οποία θα καταφέρει να συνδυάσει τα πλεονεκτήματα της εξοικονόμησης χώρου από την απαλοιφή διπλοτύπων και την τεχνική COW για υποστήριξη στιγμιοτύπων και κλώνων, με την ακαριαία λήψη των στιγμιοτύπων, τον περιορισμό των εισαγόμενων μεταδεδομένων, την επιτάχυνση του συγχρονισμού απομακρυσμένων αντιγράφων και την υψηλή απόδοση που απαιτείται για εικονικούς δίσκους πρωτογενούς αποθήκευσης.

Γι᾽ αυτό αφού διερευνάται ο υποχώρος παραμέτρων της απαλοιφής διπλοτύπων και μελετώνται οι υπάρχουσες μορφές εικονικών δίσκων και οι ελλείψεις τους, προτείνεται ο χωρισμός του εικονικού δίσκου σε τεμάχια. Η αντιστοίχιση από τομείς του εικονικού δίσκου σε τεμάχια, γίνεται μέσω μιας δεντρικής δομής μεταδεδομένων, με εσωτερικούς κόμβους που βοηθούν στην πλοήγηση και κόμβους-φύλλα που δείχνουν στα τεμάχια. Η δενδρική δομή έχει το πλεονέκτημα της ακαριαίας λήψης ενός στιγμιοτύπου, αφού αντιγράφεται μόνο ο κόμβος-ρίζα. Επιπλέον προσφέρει και δυνατότητες για εξοικονόμηση μεταδεδομένων και ταχύτερο συγχρονισμό απομακρυσμένων αντιγράφων, εάν σε μια ακολουθία στιγμιοτύπων οι αλλαγές εντοπίζονται σε ένα κομμάτι του υποδέντρου. Σε σχέση με υπάρχουσες υλοποιήσεις, ο σχεδιασμός μας προσφέρει ενοποίηση των λειτουργιών εξοικονόμησης χώρου και ανεξαρτησία των τεμαχίων, των στιγμιοτύπων και των κλώνων οδηγώντας σε μια πιο ευέλικτη υλοποίηση που δεν εισάγει αρχιτεκτονικούς περιορισμούς και προσφέρει περισσότερες ευκαιρίες για εξοικονόμηση χώρου σε περιβάλλοντα υπολογιστικού νέφους.

Υλοποιήσαμε τον παραπάνω σχεδιασμό θεωρώντας τις βασικές οντότητες ως αρχεία και γράψαμε έναν οδηγό μπλοκ συσκευής για τον προσομοιωτή εικονικών μηχανών QEMU. Έπειτα, δοκιμάσαμε διάφορα μικροσκοπικά και μακροσκοπικά φορτία εισόδου-εξόδου, ώστε να διαπιστώσουμε την απόδοση του δίσκου για τους διαφορετικούς συνδυασμούς μεγέθους τεμαχίου και ύψους του δέντρου. Παράλληλα, εξαγάγαμε θεωρητικά την επιβάρυνση σε μεταδεδομένα και διαπιστώσαμε τα οφέλη της δενδρικής δομής στην εξοικονόμηση μεταδεδομένων, με την βοήθεια ενός πραγματικού ίχνους λειτουργιών Ε/Ε.

Λέξεις κλειδιά: εικονικός δίσκος, στιγμιότυπα, κλώνοι, απαλοιφή διπλοτύπων, Copy-On-Write, εικονικοποίηση αποθήκευσης, εικονική μηχανή, υπολογιστικό νέφος, QEMU

Abstract

The problem under investigation is the development of a virtual disk format, that will not only be efficient in terms occupied storage space, by supporting deduplication, sparseness, constrained metadata overhead and the COW techinque for snapshots and clones, but will additionally offer instant snapshot/clone creation, efficient remote replication and low access latency required for primary storage. While there are many solutions that support some of the above characteristics, there is none that combines them all efficiently under a unified management, offering the agility, scalability and architectural independence required in a cloud environment.

After we analyze the design choices associated with deduplication and we study existent virtual disk formats and their shortcomings, we propose a novel design. We chop the virtual disk into independent chunks of fixed size. This inherently supports sparseness as chunks are allocated on demand. The mapping between virtual disk's sectors and chunks are carried through a recipe, which is a chunk-indexing array, that in the general case takes the form of a tree, with inner nodes pointing to nodes and leaf nodes pointing to chunks. The tree mapping structure reduces the amount of metadata that need to be copied in a snapshot/clone creation and allows greater sharing of data and metadata, thus offering instant snapshot/clone creation, avoiding metadata explosion and saving network bandwidth in a remote replication process. Deduplication can be employed for both chunks and nodes.

The above design is implemented with chunks and nodes as files in the host filesystem, and the related logic for sector translation and snapshot/clone management is implemented as a QEMU backend block driver, integrated in the QEMU block model. We use micro-benhmarks, macro-benchmarks and real I/O traces to explore the parameter subspace and conclude on a tree height and chunk size configuration, that in combination with theoretical deduplication findings can yield a time and space efficient virtual disk.

Keywords: virtual disk, virtual disk format, snapshot, clone, deduplication, Copy-On-Write, COW, storage virtualization, virtual machine, cloud, QEMU

Αντί Προλόγου

Σε αυτό το σημείο θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου Νεκτάριο Κοζύρη για τα ερεθίσματα, την έμπνευση και την καθοδήγηση που μου προσέφερε από τα πρώτα βήματα της ακαδημαϊκής μου πορείας μέχρι και αυτήν την διπλωματική εργασία. Θέλω ακόμη να ευχαριστήσω τον Βαγγέλη Κούκη, για την άρτια συνεργασία μας, την εμπιστοσύνη του, τις γνώσεις, τις ιδέες και την υποστήριξή του κατά την εκπόνηση αυτής της εργασίας. Ένα ιδιαίτερο ευχαριστώ οφείλω και στον Χρήστο Σταυρακάκη, για την υπομονή, τον χρόνο και τις υπερπολύτιμες συμβουλές και υποδείξεις του. Τέλος, ευχαριστώ θερμά την οικογένειά μου για την στήριξή της, τόσο κατά την διάρκεια αυτής της διπλωματικής, όσο και σε όλα τα ακαδημαϊκά μου χρόνια, ενώ δεν θα μπορούσα να ξεχάσω και τους φίλους που μου συμπαραστάθηκαν με την ενθάρρυνση και το ενδιαφέρον τους.

Μανώλης Ανδρουλιδάκης
Σεπτέμβριος 2016

Extended Abstract

0.1 Introduction

The problem under investigation is the development of virtual disks, that are efficient in terms of access time and occupied storage space, and can additionally support snapshots, clones and deduplication. While there are many solutions that support some of the above characteristics, there is none that combines them efficiently under a unified management, exploiting the semantics of the above operations and offering architectural independence.

In modern cloud environments that make an extensive use of virtualization, there is an imperative need for agile, storage efficient, low-latency virtual disks. That is fueled by the rapid expansion of IaaS centers, that contain a massive amount of VMs and user data. This inevitably creates a great deal of data redundancy that can be eliminated to save storage space and reduce costs, without sacrificing either performance or snapshot and clone functionalities. Moreover, the design must be scalable and the virtual disks movable, in order to comply with the elasticity of the resources.

The simple solution of sparse files that just allocate data on demand, fails to scale and really leverage data redundancy between different disks. Whereas the COW technique can effectively support clones and snapshots, naive adaptations of virtual disks on a BTRFS filesystem or over an existing underlying deduplication system, either perform poorly or impose architectural restrictions, providing little control over overall design choices. The qcow2 format, that is a widespread format used by QEMU, may combine

data sparseness with snapshot and clone features, but fails both to provide autonomous snapshots/clones that are movable in a cloud environment, and to cooperate in an intelligent way with a deduplication system, since its structures are only internally visible. A possible extension would create duplicate metadata, and its current version also engages extra operations that slow down accesses of snapshots and clones.

In order to deal with the aforementioned shortcomings and harmonically combine snapshots, clones and deduplication, we chopped the virtual disk into chunks of fixed size. This inherently supports sparseness as chunks are allocated on demand, as soon as they are written for the first time. Moreover, upon snapshot creation, chunks are marked as read-only, so if a write is necessary on the primary disk, the appropriate chunk is copied to a new one that stores future changes. This way, all chunks that belong to snapshots are marked as immutable and are subject to a deduplication system that can eliminate redundancy. The mapping between virtual disk's sectors and chunks are carried through a recipe, which is a chunk-indexing array, that in the general case takes the form of a tree, with inner nodes pointing to nodes and leaf nodes pointing to chunks. The tree mapping structure reduces the amount of metadata that need to be copied in a snapshot/clone creation and allows greater sharing of data and metadata. The above design was implemented with chunks and nodes as files in the host filesystem, and the related logic for sector translation and snapshot/clone management was implemented as a QEMU backend block driver, integrated in the QEMU block model.

We used micro-benhmarks, macro-benchmarks and I/O traces to explore the parameter subspace and conclude on tree height and chunk size configuration, that in combination with theoretical deduplication findings can yield a time and space efficient virtual disk. Our experimental results can be further improved, especially with a filesystem independent implementation.

0.2 Backround

0.2.1 Storage

In order to store data persistently, computers use a storage hierarchy composed of different media. Primary storage targets transient, frequently accessible data in need of

low latency. The huge share of the primary storage market is made up of hard disk drives (HDDs), that are based in magnetic recording and store data into basic units called sectors. Due to their physical and geometrical characteristics, HDDs are much more efficient at sequential than at random access patterns. To overcome the sector seek overhead incurred by the rotation of the platters and the movement of the disk head, flash memory based on electrical logic gates has been introduced. SSDs are a more expensive, flash memory equivalent of an HDDs, which are capable of improving latency and bandwidth, by setting an upper limit on the number of possible writes. Further on the hierarchy, the need to combine persistent storage and portability is facilitated by removable media such as CDs, DVDs and flash drives. Secondary storage is associated with backup or archival storage, and to that end, cheaper and high latency mediums such as tape libraries or optical jukeboxes, are used.

A hard disk's capacity can be divided into groups of continuous sectors, called partitions. Each partition stores data that range from simple files to an operating system. In order to store data in a unified and organized manner, a storage stack is being employed. At the top of the stack, a user or a userspace process reads and writes data to files, which are grouped into directories. The abstracted notions of files and directories are provided by the filesystem, a kernel component that divides the physical disk space into blocks and allocates them for data and metadata storage. Metadata, which are maintained in both on-disk and in-memory specific data structures, consists of files' and directories' information, like ownership, timestamps and size, but also includes managing information, such as block bitmaps, indexes and checksums. The variety of data and metadata organization and algorithms, leads to the existence of different filesystems, which, in the Linux storage stack, all operate under the hood of the virtual filesystem (VFS). This VFS creates an abstraction layer that hides the specifics of the different filesystem implementations and offers a unified interface for file access, in the form of system calls. Both read and write system calls, unless instructed otherwise (zero-copy schema), make use of the page cache, which consists of memory pages used to temporarily store data blocks, in order to accelerate future requests. A read system call, stores the derived disk data first in the page cache, so that future requests will avoid another disk access. A write system call, copies the data into the page cache and defers the disk synchronization to a later time, so that write requests in the near future will be carried out in memory and without requiring disk access. If data do not reside

in the page cache or if data should be flushed to disk, an I/O request is initiated by the Generic Block Layer. This layer first creates a BIO structure used to describe the block device region to be read/written and the memory pages of the data, and then inserts that BIO into a request structure, which is member of the block device's request queue that stores pending requests. I/O requests are not serviced instantly even if possible, in the hope of reducing disk I/Os, by grouping and merging of sequentially contiguous requests. Moreover, this request queue is utilized by the I/O scheduler, which bearing in mind the physical characteristics of a hard disk, tries to reorder the requests in order to optimize performance. This is usually implemented through an "elevator" policy, according to which the disk head is preferable to travel in its current direction and not to randomly jump. Next in the stack, the Block Device Driver is responsible for translating high level into low level operations, specific to the device interface. Therefore, a routine of the block device driver pulls the first request out of the dispatch queue formed by the I/O scheduler, initiates a DMA operation between the disk controller and the appropriate memory addresses and then terminates. As soon as a DMA completes, an interrupt is issued and the routine reruns. The disk controller, adhering to a specific interface, such as SATA, PATA or SCSI, translates the low lever operations delivered to the bus's I/O ports, into electric signals for the disk. An disk cache is also usually present, in order to further improve performance.

Beyond the DAS model of a disk connected locally to a computer bus, modern approaches favour network attached storage in order to facilitate resource sharing, management, maintenance, resilience, and logical abstraction. NAS is based on a file-level server connected to a computer network, while SAN is a storage network which provides access to consolidated, block level data storage, so that the devices appear to the operating system as locally attached devices. Block or file sharing on SANs is possible through shared-disk (clustered) filesystems that serialize the requests to avoid data corruption. Other emerging technologies include object storage, which separates data and metadata and indexes them through a global id, and content addressable storage, which indexes data based on their content (hash value). Rather significant is also the concept of data immutability (data cannot be altered after they are stored) for its data protection, version-control capabilities and concurrency-solving properties.

0.2.2 Virtuaization

Virtualization is the process of creating a virtual, rather than actual/physical version of a computer component, including whole hardware platforms, Operating Systems, storage devices, and computer network resources. Therefore, a virtual machine is possible through the software emulation of its hardware components. A VM enables concurrent operating systems, isolation and security between them, portability, efficient resource management and flexibility on application running or testing. On the downside, due to hardware emulation performance is inferior, and due to shared physical resources, a heavy load on a VM may affect the performance of another VM.
The software layer that performs the emulation is called hypervisor, the physical machine is called host and the virtual is called guest. The hypervisor is either installed directly above the hardware (type-1) or is a userspace process communicating with the host kernel (type-2). Virtualization can be divided to full virtualization, where the guest is agnostic of its virtual state, the guest OS can be used without modifications and all hardware is emulated, and to paravirtualization, where in order to improve performance, the guest OS is altered to use extensions that communicate with the host or use directly the underlying hardware. Specialized hardware extensions have been developed to boost performance. Moreover, there are special kernel drivers and interfaces for the efficient communication between the guest and the host, so that a VM can choose to use paravirtualization for other components such as disks, network devices, boot firmware, page tables etc. We will use the QEMU-KVM virtualization platform, which compounds the KVM linux kernel modules necessary to exploit the CPU virtualization extensions and the QEMU emulator, which is an open source platform for the creation and management of virtual machines, providing also the emulation of various hardware devices. Virtualization is vastly leveraged in cloud computing environments, where the isolation and abstraction of virtualization can be ideally combined with the intelligent resource management algorithms of the cloud, providing scalability, flexibility and effective load balancing.

Virtual disks are one of the components that need to be virtualized to store the guest's OS and data. The simplest way is that each virtual disk present in the guest corresponds to a regular file in the host filesystem. Hence, the hypervisor needs to intercept the guest requests to the virtual disk interface and transfer data from the regular

file to the guest's memory and reverse, thus emulating a DMA operation. The guest will maintain the illusion of actual hardware and so the guest's sector requests will be transparently converted by the hypervisor to file regions requests. For that conversion to work, a mapping layer is necessary so that each virtual sector will correspond to a 512-byte file region on the host file. In the simplest case, adopted also by the QEMU raw block format, this layer is merely a linear mapping, that maps sector x to the file offset $512 * x$. In a richer approach that can offer extra characteristics such as snapshots, a mapping structure may be maintained (table, B-tree, hash buckets) to perform the translation. This mapping structure constitutes metadata and can be placed in the same file as the virtual disk's data. The mapping layer policy along with other metadata structures, define the virtual disk format.

When guest the OS issues an I/O operation, the guest block driver sends the appropriate instructions to the disk controller, so that a DMA operation will be initialized. Those instructions require the use of unprivileged commands that come from a userspace process (hypervisor) and so invoke a software exception in the host. The exception instructs the hypervisor to deal with the request and so the hypervisor acquires the necessary information that were about to be send to the disk controller. From that point, it uses a handful of software layers, firstly to interpret the command according to the virtual disk interface and afterwards to translate the sectors requested to file offsets, with the aid of course of the mapping layer. This translation is carried out by a component of the hypervisor, called backend block driver, and for each virtual disk format there is a different backend block driver. As soon as the translation or other necessary operations complete, the backend block driver issues `read()/write()` system calls to the host VFS, in order to read/write the desired data from/to the file. When the data transfer is complete, the hypervisor notifies the guest.

Instead of depending on software exceptions for the guest requests to be processed by the hypervisor, a paravirtualized disk interface can be used to boost performance. VirtIO is a Virtualization standard for network and disk device drivers where just the guest's device driver knows it is running in a virtual environment, and cooperates with the hypervisor. Under the hood, VirtIO works by sharing memory between host and guest. The latter uses the VirtIO device drivers and instead of usual I/Os, it writes data to a circular buffer and notifies the former.

A virtual disk may not correspond to a regular file to the host file system but may also be a file in a shared-disk filesystem or at a remote NAS node. Moreover, it can correspond directly to a block device to store data and metadata into raw storage, thus removing the filesystem overhead and possibly utilizing SAN capabilities. Finally, a virtual disk may use a different storage technology, such as object storage, creating the need for specialized hypervisor block drivers.

0.3 Design and Analysis of Virtual Disk Formats

0.3.1 Space Efficiency techniques

Sparse files

Sparse files are files whose blocks that only contain zeros are not actually stored on disk. This means you can have an apparently 16G file only taking up 1G of space on disk. This can be particularly useful in situations where the full disk may never be completely used which is quite common in virtual machines' disks. A file system that does support sparse files allocates a data block only when the process needs to write data into it. When reading sparse files, the file system transparently converts metadata representing empty blocks into "real", blocks filled with zero bytes at runtime. File holes, namely the "empty" space in between the allocated blocks, come to life through the file system mapping between file blocks and disk or logical blocks (groups of continuous sectors equal to the size of the filesystem block).

Using the ext3 filesystem as a case of study, the translation between a file offset and a logical block number is carried out through a mapping structure, located in the file's inode. This is an array-like structure which is indexed by the file block number (file offset divided by the filesystem's block size) and contains pointers to logical block numbers or in case of high file block numbers, pointers to a higher-level indirect structure to be recursively indexed. Rather than allocate logical blocks of null bytes for the holes in a file, the file system just marks with the value 0 all pointers that correspond to empty file blocks. This technique applies also to pointers to indirect blocks.

Except for the fact that storage is only allocated when actually needed and so large files can be created even if there is insufficient free space on the file system, sparse files also

reduce the time of the first write, as the system neither allocates blocks or writes all zeros to the "skipped" space. On the downside, sparse files may become fragmented, file system free space reports may be misleading (disparity between utilities that use allocated blocks and those that use the nominal file size), and they are not fully supported by all backup software or applications. That means that any backup or copy utility that does not recognize or utilize sparse files can replicate the entire file on disk, so that a 500M sparse file could suddenly balloon to 6G.

Snapshots, Clones, COW and BTRFS

In computer systems, a snapshot is the state of a system at a particular point in time. More specifically, a virtual disk snapshot is a read-only copy of the disk data, frozen at a point in time. A snapshot enables the virtual machine's state at the time of the snapshot to be restored later, effectively undoing any changes that occurred afterwards. This capability is useful as a VM backup technique, for example, prior to performing a risky operation such as testing new software, examining a virus or applying a service pack. Apart from a VM backup and restore point, snapshots can be used as a source for data mining.
Clones are writable snapshots. The state of the system at the point of time the clone was taken, can be loaded and run with new changes to be effected. Clones are extremely popular on cloud environments, for the creation of multiple VMs with a particular OS and configuration, out of a single image.

There are several techniques to implement snapshots and clones. A naive approach would be to copy explicitly all data blocks of the virtual disk to a new location on the disk (split mirror approach). This requires advanced planning, high storage overhead, cannot be done on-line and applications must be stopped. However, smarter approaches such as Copy-On-Write and its slight modification Redirect-On-Write are those that are actually used.

Copy-on-write (COW) is method of allocation that enables a point-in-time, read-only snapshot of data to coexist with a live, writable version of the same data, without consuming more space than necessary. The principle of the COW idea is that when many tasks use initially identical copies of the some information, that they may occasionally need to locally modify, then it is not necessary to immediately create separate copies of that information for each task. Instead they can all be given pointers to the same

resource, and whenever a task attempts to make a change to the shared information, it should first create a separate, private copy of that information to prevent its changes from becoming visible to all the other tasks.

At the time when the snapshot is created, a small volume is allocated as a snapshot volume with respect to the source volume. Upon the first write to a data block after the snapshot, the original data of the block is copied from the source volume to the snapshot volume. After copying, the write operation is performed on the block in the source volume. As a result, the data image at the time of the snapshot is preserved in the combination of the source and the snapshot volume. All subsequent read I/Os and all write I/Os after the first change to a block, are performed on the source volume. If a process attempts to read the snapshot at some point in the future, it accesses it through a dedicated structure that keeps track of the blocks that changed since the snapshot was taken.

Copy-on-write requires 3 I/O operations upon the first write to a block: one to read the original block from the source volume, one to write the original block to the snapshot volume, and one to write the new data in the source volume. These I/O operations are done at production time, which may negatively impact application performance. To overcome this, one can adopt the redirect-on-write policy, which leaves the original block in the source volume intact and the new write operation is performed on the snapshot volume. So if a block needs modification, the storage system merely redirects the pointer for that block to another block and writes the data there. The data image at the time of a snapshot is preserved in the read-only source volume. All subsequent write I/Os are performed on the snapshot volume, while read I/Os may go to source or snapshot volume, depending on whether the block has changed since the snapshot. A dedicated structure knows where all of the blocks that compose the primary disk are.

The snapshot can be often created instantly and without taking the system offline, since it does not involve any data copying at that time. The original copy of the data continues to be available to the applications without interruption. In general, copy-on-write performs well on read-intensive applications while redirect-on-write performs well on write-intensive applications.

COW is also used as a disk block update policy. The target block is read into memory, modified, and then written to disk at a new location leaving the old data unmodified.

Since it never overwrites data, many filesystems, such as BTRFS, use it for data integrity and protection. BTRFS metadata is structured on a B-tree, a tree of block-sized nodes which are filled with items [btra]. The items of the inner nodes of a tree help navigation to the leaves which store items describing different kinds of metadata, such as pointers to data extents. Because in BTRFS each block modification (data extent or node) incurs a write to a new location and hence an update to the ancestor's pointer, which in turn is updated with COW of its block, the COW process propagates up to the file system root.

The BTRFS tree structure enables a snapshot of the whole filesystem, since it suffices to copy the root and the underlying nodes can be shared between many snapshots [Rod08]. If a leaf node needs to change, then all of the nodes in the tree path will be COWed and now both the original nodes will be accessible from the snapshot root and the new COWed nodes will be accessible from the original root. To differentiate between the nodes that after a COW operation are shared and those who are obsolete and need to be discarded, reference counters are used. Each block's reference counter describes the amount of ancestors pointing to that block and thus marks not only the specific block as shared, but also its underlying subtree, without explicitly altering the reference counters of its nested nodes.

Although COW allows sharing of common blocks between different volume versions, always writing in new spots incurs fragmentation and so access latency increases. Therefore, COW is not suitable for virtualization or anything that has small and random write patterns. Moreover, constant COW on tree structures introduces an unpleasant recursive updating procedure, that propagates up to the tree root.

Deduplication

Deduplication is a technology that can be used in a storage system to reduce the amount of storage required for a set of volumes, by dividing them into "chunks", identifying duplicate ones and storing only one copy of each chunk. For example in a email storage system that contains 100 instances of the same 1 MB file attachment, only one instance is actually stored; the subsequent instances are referenced back to the saved copy. The automatic removal of duplicate data has been widely used in backup systems, since versions of backup data expose a great deal of redundancy. However, in modern virtualized cloud environments, deduplication can be also used for primary

storage, to eliminate common chunks of data between different virtual machines or even snapshots of the same VM. Deduplication saves disk space, omits the need for golden images, may lead to fewer disk writes increasing the lifespan of SSDs, and may reduce network bandwidth. On the downside, it de-linearizes and de-localizes data placement, introduces an extra overhead at the write path and increases metadata.

Deduplication can be divided into intra deduplication, which leverages redundancy in a sequence of snapshots/backups of the same image in time and inter deduplication, which exploits redundancy between different images. Moreover, intra deduplication can be further divided into temporal, where we exploit redundancy between different versions of the same image in time, and spatial, where we leverage redundancy between different chunks of the same version. Finally, a storage system can perform inter-intra deduplication to exploit all of the above types of redundancy in the storage pool.

In virtualized environments, sources of redundancy between the VM images include images with the same or similar operating systems, distributions and kernels, and out of them many contain the same or similar applications and dependency packages. User's private images are slightly modified public images with passwords, public/private keys etc. Software patches are also a major benefit while redundancy can even be found when multiple users of public cloud infrastructures independently store the same files, such as media and popular Internet content. There are also a lot of intelligent ways to exploit and predict redundancy by the use of caches for popular images, prefetching mechanisms and bias for popular file extensions or positioning in the address space. As a token of the paramount importance of deduplication in virtualized environments, we mention that space reclaimed improves roughly linearly in the log of the number of file systems in a domain, with the effect of grouping file systems together being larger than that from the choice of chunking algorithm or chunk size [JPZ+11].

The base procedure of a deduplication system is the following. Firstly it divides the data stream into non-overlapping chunks of fixed or variable size. Then it calculates the fingerprint of each chunk, which is a hash value of the chunk's contents (like SHA-1 or MD5) and it is the chunk's unique identifier. Two chunks with identical fingerprints are considered duplicates without requiring a byte-by-byte comparison of their contents. A fingerprint index maps fingerprints of the stored chunks to their physical addresses. As soon as the chunk acquires its fingerprint, a lookup in the fingerprint index is nec-

essary to check if there is a duplicate chunk already stored. The chunks with unique fingerprints that do not exist in the fingerprint index, must have their contents physically stored, so they are aggregated into fixed-sized containers (typically 4 MB), which are managed in a log-structure manner. Then, the fingerprint along with the container's physical address is added to the fingerprint index. Finally, the data stream should be restorable and so in order to be able to reconstruct it from the various chunks, there is a file, called recipe, that contains the sequence of fingerprints of the chunks of the data stream. Therefore, whether a chunk is duplicate or not, its fingerprint will be stored in the recipe. As an enhancement of the above mentioned process, a fingerprint cache is employed in order to avoid costly lookups in the fingerprint index, which is usually too big to be stored in RAM.

We can outline a taxonomy of the basic deduplication parameters based on the major design decisions implicit in all deduplication systems. The implications of those parameters can be generally measured in terms of storage savings, real-time performance (time to complete a data stream storage request), restore time, cpu time, memory footprint, network traffic and privacy guarantees.

A smaller chunk size leads to a more fine-grained and so to a more effective deduplication, since smaller chunks can help to isolate the parts that have changed from the parts that have not. Moreover, a smaller chunk size leads to more efficient upload transmission since less data is sent. On the other hand, smaller chunks mean more chunk fingerprints for the same amount of data, so not only we will have more metadata overhead from bigger recipes and fingerprint index, but also the latter implies an increase in the lookup time of fingerprint index searches and thus a performance decrease. Furthermore, smaller chunks limit the opportunities for efficient post-chunking compression and may also incur increased fragmentation since we disrupt even more the data locality. Results of numerous papers [NKO$^+$06], [JM09], [JPZ$^+$11], [MB12], [WDQ$^+$12] and the choices of existent systems (table 3.1), more or less converge to the 2KB-16KB region for effective deduplication. Deduplication storage savings for those sizes typically both outweigh metadata overhead and are more significant than compression gains. They also agree that the deduplication ratio decreases quickly with chunk size increase. However, even with bigger chunk sizes, like 128KB, one can yield decent storage savings, especially if we take into account the metadata overhead introduced by smaller chunks [SKM$^+$16].

As for the chunking algorithm, a system can implement whole file chunking where chunks coincide with files, fixed size chunking where the data stream is divided at fixed boundaries (e.g. every 4KB), and variable size chunking or content defined chunking, where the selection of chunks is based on some property of the contents, usually the Rabin fingerprint of a sliding window. The variable chunking is resilient to insertions, since it will not cause every subsequent chunk's fingerprint to change, but it demands extra computations that are detrimental to real time performance. While almost all papers agree that variable size chunking is better, there are reports that fixed size chunking is able to detect a significant level of deduplication [JM09], [JPZ+11].

If we have a storage system that receives network requests for data storage, chunking and hashing can be performed either prior (client) or after the data transmission (server). In client's case, we first send the fingerprints of all chunks, and so we avoid transferring the contents of those chunks already present in the server. This way, network traffic and server's cpu resources can be saved, but the chunking and hashing process may be burdensome for the client, and so inappropriate for some applications.

Another parameter is the timing. In the inline approach deduplication is carried out in the critical write path, with chunking, hashing, lookup of chunk fingerprint, physical storing and recipe update, are carried out before a write request is completed. In the offline approach, deduplication is run as a background process at a later convenient time outside of the write path, and data are first saved in some temporary location on disk. Inline deduplication can be combined with client chunking and hashing, demands less writes, reads and -even temporarily- disk space, but increases the latency of a request, as processing is done in the write path. This increase, due to fingerprint index lookups which require random disk I/Os, is far from negligible and may induce an unacceptable overhead for primary storage writes. Offline deduplication offers flexibility to choose a convenient low system-load time and the chance to deduplicate only older files, in order to avoid additional latency for the most accessed files on the server.

The fingerprint index can be used for either exact deduplication, where all duplicate chunks are eliminated for highest deduplication ratio or near-exact deduplication where a small number of duplicate chunks are allowed for higher real time performance and lower memory footprint. The exact approach (ED) needs a fingerprint index cache to avoid disk I/Os of lookups to the key-value store and it is usually paired with a Bloom

filter in DRAM, to reduce the lookup requests. Fragmentation can make the fingerprint cache ineffective and so make the fingerprint index lookup-intensive over time. In the near exact approach (ND) we have a partial index where only sampled representative fingerprints (features) are indexed and so a fingerprint index cache is necessary to maintain a high deduplication ratio, since the prefetched unit will contain fingerprints that are not in the key-value store. Near-exact deduplication generally indicates a cost increase, but may be useful for large storage systems to decrease write latency.

According to the prefetching unit of the fingerprint index cache, we exploit either logical locality, namely the fingerprint sequence as preserved in recipes, or physical locality, namely the fingerprint sequence as preserved in containers. Physical locality will either increase lookup requests in ED or it may possibly reduce deduplication ratio in ND. Logical locality seams more appealing, but since a fingerprint may be present in many recipe offsets, the process of extracting the correct fingerprint sequence out of the many possible, becomes non-trivial. Base procedure performs well in source code and database datasets but underperforms in a virtual machine dataset, because of the self-reference that is prevalent in VM images. The solution is similarity detection, a policy that loads the x most similar stored segments (fingerprint sequences in recipe) in order to deduplicate the processing segment, instead of loading just the stored segment that lied in the same position in the previous stream, as in logical locality.

There are also different choices in case that the deduplication system needs to span across many nodes. In local deduplication each node performs deduplication independently with its own index, and duplicate chunks are not eliminated across distinct nodes. Some systems introduce intelligent routing mechanisms that map similar files or groups of chunks to the same node to increase the cluster deduplication gain. In the global approach, duplicate chunks are eliminated globally across the whole cluster, but this requires an index mechanism accessible by all cluster nodes, which is prone to scalability and fault tolerance issues. Finally there is the centralized approach, with systems that perform deduplication in a single node.

In the context of privacy, the first approach is the strict encryption using a per user private key, which is safe but limits deduplication to a per-user chunk pool. The compromising alternative is the convergent encryption, where each chunk is encrypted using its hash value as key. Therefore identical chunks produce identical crypto-chunks

and inter-deduplication is possible. In order to protect the chunks' hashes from the attacker, each recipe is encrypted with a per-user private key. Convergent encryption is vulnerable to the "confirmation of a file" and "learn the remaining information" attacks, but requires approximately two thirds of the strict encryption's storage resources.

Finally, two chunks can be considered equal just by comparison of their hash values or a full byte to byte comparison of their data may be imposed. Most actual systems trust the hash comparison, in order to avoid further performance penalty and since the probability of hash collisions is much smaller than that of hardware errors.

Backup and archival systems assume immutable data, trade latency for deduplication ratio and mainly use inline deduplication approaches. Different backup versions of the same file system can expose a great deal of redundancy compared to the spatial redundancy of a file system and so they reach a deduplication ratio from 83% up to 90%. Redundancy can also arise in virtualized cloud environments targeted on primary storage, but here low I/O latency is critical, and the fact that data are mutable may demand copy-on-write techniques to prevent updates on aliased data. Moreover, read requests are very frequent and must be served more efficiently than the restore operations of backup storage. The percentage of offline approaches is raised and the deduplication ratio is about 68%, or up to 80% if we focus on large clusters.

Although not as important as deduplication, compression can be additional used to save disk space. This is done by identifying repetitive regions inside unique chunks or groups of them, in the standards of a compression algorithm like DEFLATE. As a very remarkable result we state that high deduplication savings achieved with 4KB chunk size are attainable with larger 64KB chunks with chunk compression.

0.3.2 Case example: qcow2

In order to study a real-world virtual disk format and illustrate a practical application of the notions of snapshots, clones, COW and sparse files, we present the qcow2 format supported by the QEMU emulator. It is a representation of a fixed size block device in a file and it offers smaller file size, even on filesystems which don't support holes, copy-on-write or equivalently clone support where the image only represents changes made to an underlying disk image, snapshot support where the image can contain multiple

snapshots of the image's history, and optional zlib compression and AES encryption.

The image file is organized at the granularity of cluster, whose size is a multiple of a sector, and is the unit in which all allocations are done, both for actual guest data and for image metadata. The image header contains the basic information of the image file and then comes a two- level lookup table, a reference table, and data clusters. Qcow2 uses a two-level index tree, made up from L1 and L2 tables, for the mapping of host clusters (virtual sectors) to guest clusters (groups of bytes). A virtual sector address in the guest is split into three parts, a_1, a_2, a_3: a_1 is used as the L1 table's index to locate the corresponding L2 table; a_2 is used as the L2 table's index to locate the corresponding data cluster; a_3 is the offset in the data cluster. If an offset is 0 then it means that the particular cluster has not been allocated. L1 table has a variable size, and L2 table is one cluster big. The reference table is analogous to the two-level mapping structure, and is used to track each cluster used by snapshots. A refcount of 0 means that the cluster is free, 1 means that it is used, and ≥ 2 means that it is used by at least one snapshot and any write access must perform a COW operation.

Qcow2 supports Copy-on-write images, which are a clone substitute. A COW image can be used to store the changes from an original disk image (backing file), without affecting the latter. The COW image looks like a standalone image, but only the clusters which differ from the original image are stored in the copy-on-write image file itself. Writes are performed solely in the COW image and if a cluster will be partially written, a COW from the the backing image is required. On reads, you first check to see if the clusters in question are allocated within the COW image. If not, the backing image's path is retrieved from the header and the clusters are read from the backing image. A chain of COW images can be used as a series of incremental snapshots of a root image, with each image s_x embodying the disk changes between snapshots s_{x-1} and s_{x+1}.

Qcow2 also supports internal snapshots. Each snapshot is represented by a snapshot header and a copy of the L1 table at the time of the snapshot, both placed inside the image file. Besides those, the refcount of all L2 tables and clusters reachable from the L1 table must be increased, so that all the allocated clusters of the image will become read-only and a write in the original live image will trigger a COW operation. At restore, the live L1 table is overwritten by the snapshot's L1 table. Snapshots are only visible and accessible through the primary disk image file.

0.3.3 Problem Statement and Shortcomings

The problem we are dealing with is the design of a virtual disk format that supports clones, snapshots, sparseness and deduplication, while at the same time demands few metadata and meets the low latency requirements of primary storage. It is also vital that snapshots, clones and virtual disks are instantly created and quick remote replication is also desirable.

If sparseness is supported, the size of the virtual disk grows on demand, as soon as the user writes data to unwritten sectors. The creation of the virtual disk is instant and for as long as not all sectors have been written, space can be saved. We can base our virtual disk on a file and rely on the filesystem implementation of sparse files, but this introduces architectural restrictions, since we are bound to a filesystem for storing virtual disk's data and metadata; a filesystem that must additionally support sparse files.

The process of copying the whole virtual disk file to create a snapshot or a clone (split mirror technique) is an excessively lengthy operation that also leads to duplicate data. To avoid both, data should be divided into chunks that are initially shared between a snapshot and a primary disk, and as soon as a write for a chunk arrives, that chunk must be COWed to both reflect the change in the primary disk and preserve its initial data for the shake of the snapshot. The mapping between virtual sectors of disks/snapshots and chunks and the tracing of sharing chunks require a metadata structure. The smaller the size of the chunks, the finer the sharing granularity, and so we get higher data savings, but also more metadata to store and traverse. Consequently, we need to strike a balance for the appropriate chunk size, and come up with a convenient metadata structure that guarantees instant snapshot creation and facilitates an efficient remote replication. Finally, it is important that snapshots are autonomous and can be moved independently of their initial disk. All of the above are true for clones as well.

Deduplication can be used to eliminate common chunks present in one or between different virtual disks in a cloud environment. However, low-latency requirements and highly transient data of primary storage, require a specialized tunning of the chunk size, the timing and the data selection. Therefore, the use of some existing deduplication system against a virtual disk file, not only restricts the configurability and architectural flexibility, but also rules out the potential of unifying the similar COW and deduplication metadata structures, thus sacrificing space and ease of management.

The popular solution of the qcow2 disk format, for starters, lacks an official deduplication support. Transparent deduplication against the file on an underlying level, discards clusters semantics useful to avoid deduplication on transient data or qcow2 pertinent metadata and introduces duplicate metadata structures that burden storage. A qcow2-tailored deduplication system that will integrate its recipes in the inherent metadata structures, may need to either restrict deduplication flexibility and efficiency or alter the existing qcow2 structures and policies. Besides deduplication, qcow2's internal snapshots are dysfunctional, since they are not accessible and movable outside the primary disk, consume its space, and engage a slow reference count update process at creation. External snapshots in the form a chain of COW images, are incremental and so highly dependable on their backing images, may end up in multiple traversals until data are acquired and a potential merging, which is mandatory in the case of a snapshot deletion, is time consuming. Moreover, COW images acting as clones require double metadata traversals as well and their data provision ability highly depends on the accessibility of their backing image; a design that if left as it is, also excludes any non filesystem-based storage implementation. Other virtual disk formats (vmdk, vdi, vhd) have similar mapping structures and characteristics, so the arguments above apply to them as well. Finally, if a cluster will be partially written, qcow2 may perform in the worst case 5, instead of the minimum 2 I/Os required for a COW operation.

An alternative elegant solution would be to use regular files on top of a feature-rich filesystem, such as BTRFS. Sparseness, snapshots, clones and deduplication will be automatically provided by exploiting the inherent filesystem mechanisms and by disabling constant COW on each write, the BTRFS can surpass the induced fragmentation and recursive update amplification issues. However, this bounds us with a specific filesystem, imposes rigid, inaccessible and inefficient parameters and metadata structures for deduplication (BTRFS extents are too big), and excludes the possibility of using the virtual disk without a filesystem, on top of other storage technologies (e.g object storage, SAN).

0.3.4 A first approach: Flat recipe

Sparseness, deduplication and COW technique for snapshots and clones, dictate that the data of our virtual disk is chopped into chunks. Chunks will hold a number of con-

tinuous sectors and we choose fixed-sized chunks for various reasons. The filesystems in primary storage write data in fixed-sized blocks and so the "insertion in the middle" problem is not so common, fixed-sized chunking can yield deduplication results equal or comparable to variable chunking and so it is preferred by most primary storage deduplication systems, and finally our metadata structures enjoy simplicity and static mathematical properties that can speed up indexing. To translate a sector number to a chunk we use an array called recipe, which describes the sequence of chunk-names that linearly compose the virtual disk. Each cell/entry of the recipe is therefore indexed by dividing the sector number with the sectors per chunk. The entry contains a unique name identifier that points to a physical chunk, which on the proper offset holds the sector's data. Moreover it includes a permission bit that informs whether the pointed chunk is shared and needs COW in the next write request, or not. This design supports sparseness as well, since the physicals chunks are allocated on demand.

In order to take a snapshot, we just have to copy the recipe and clear all permission bits of the original recipe to indicate sharing. From now on, each write request to a shared chunk will trigger a COW operation, that will create a fresh chunk, copy the original data and write the new data parts. The pointer of the entry will be redirected to the new chunk and the permission bit will be set to allow further writes without COW. The original chunks will be intact and still accessible through the snapshot's recipe.

Although it seems that we meet our goals, there is definitely room for improvement. For large virtual disks with small chunks, the number of entries per recipe explodes and so is the metadata storage overhead for a series of snapshots or a lot of virtual disks. Moreover, due to the big recipe size (e.g. 500MB for a 400GB disk with 16KB chunks), snapshot creation is not instant any more (10s with an average HDD). On the other hand, big chunks that shrink the recipe size, are not efficient in eliminating data redundancy.

0.3.5 Our Design: Tree recipe

To overcome the aforementioned obstacles, we shift to a tree mapping structure approach, where the recipe consists of nodes. The leaf-nodes, in the same manner as before, contain entries that point to fixed-sized physical chunks that compose the data of our virtual disk. The inner nodes, contain entries that point to other nodes, so that

the translation of a virtual sector number to a chunk will be carried out through tree traversal from the root node. This tree structure is similar to BTRFS B-trees [Rod08].

The height of the tree and the chunk size are configurable when creating a primary disk, but they remain constant afterwards. All nodes have the same number of entries (epn), which is determined by the fact that the total leaf entries (at level 1) necessary to index all chunks ($\frac{disksize}{chunksize}$) are epn^{levels}, because each node points to epn nodes. Likewise, we can calculate the range of sectors accessible through each entry's subtree, and so navigate a request in respect of the sectors that need to be accessed. After the appropriate entries have been found, their subtrees are recursively traversed in a DFS fashion, until we finally reach the chunks. We preserve the permission bit in the entries, which signals whether the node/chunk pointed is shared, and so a write demands a COW operation, or if it is "hot" and can be directly written. Nodes need to be written in the case that a child-node/child-chunk gets allocated, and so forces its parent entry to be updated with a new pointer and permission bit. A node/chunk gets allocated either because it is the first time that it is accessed, thus supporting sparse disks, or because it is shared and a COW operation is due. Entries point to nodes/chunks by unique ids or in case they are deduplicated, by fingerprints, and are zero in case the underlying subtree is not allocated yet. A read request that meets a zero entry through traversal, gets without further descent the appropriate amount of zeroed sectors out of thin air. A node header is placed in the beginning of every node, to ensure that the node is autonomous and can be traversed out of the context of that specific tree. Finally, a separate recipe header, which is the virtual disk's representative, contains among other useful information, an identifier pointing to root node.

Now snapshot creation requires copying just the root node and clearing its write-permission bits. Automatically, all underlying nodes and chunks become recursively shared. We do not need to clear the write-permission bits of all tree nodes upon snapshoting, since this will be performed gradually. If a write request encounters a read-only entry, then it will not only perform COW for the child-node, by creating a new node and copying the original contents, but it will also clear this node's permission bits. Although this clearance should have logically occurred at the time of snapshoting, our lazy, on-demand clearance policy also implemented by BTRFS, avoids producing bulk I/Os that will delay the snapshot creation. The original nodes/chunks are still accessible for reading through the snapshot root node. For restoring a snapshot,

the contents of the snapshot root are copied to the primary root and write-permission bits are also cleared. As for a clone, it can be treated as a separate primary virtual disk, whose root node is a copy of some snapshot's root node with the permission bits, once more, cleared. Hence, all the clone's nodes and chunks are initially shared and gradually modified. Even for large virtual disks and a small chunk sizes, the size of root node does not exceed some hundreds of kilobytes, so the snapshot/clone creation and restoring operations are instant.

Deduplication can be implemented for both chunks and nodes, although there is not much potential for the latter, since temporal intra deduplication is already eliminated by COW. In addition, it is preferable that only immutable chunks/nodes will be deduplicated, to avoid the excessive overhead and the low savings of highly tentative data. By taking into account the low-latency requirements of primary storage, it seems wise to implement an offline deduplication approach, that when system load is low, will track the immutable chunks/nodes through their recipe headers, and will perform deduplication with an external fingerprint index. Deduplication should propagate in a bottom-up way, so that all children nodes will be hashed before the parent node, or otherwise deduplicating nodes is futile.

Chunks and nodes are uniquely identified by the combination of their virtual disk's id, a snapshot number and for chunks, their logical position in the chunk sequence that comprises the disk, and for nodes their level and position in the level. The snapshot number differentiates them between their many possible COWed versions.

Another prominent feature of the tree-recipe apart from the instant snapshoting, is that because of node sharing, metadata that are not modified between two or more snapshots will be automatically shared. This happens in node-size granularity, and can produce substantial storage savings, especially for snapshots that change slightly and locally. For the same reason, a replication process trying to synchronize a snapshot with a remote backup, instead of sending directly all data and metadata, which would consume network bandwidth, can gradually send entries' fingerprints or ids to the remote server. If the fingerprints exist, then it avoids sending the whole subtree under that entry, which is significantly beneficial if the two versions differ slightly and locally or if the remote server has a rich fingerprint index.

In order to avoid the extra I/O overhead introduced by the update of reference coun-

ters, we choose mark and sweep as our garbage collection technique. In the first pass, we mark all stored entities in the system, and in the second pass we mark all entities that are accessible through the recipe headers. The chunks/nodes marked in the first but not in the second pass, are unused because of some previous primary disk/clone/snapshot deletion, and so their space can be safely reclaimed.

0.3.6 Theoretical evaluation and performance model

The tree recipe supports filesystem-independent sparseness and instant disk creation, through on-demand node/chunk allocation, data and metadata sharing between primary disks, snapshots and clones, through COW, offline inter and spatial intra deduplication of immutable data and metadata.

In contrast to the flat approach, the tree-recipe also offers instant snapshot and clone creation, because only the small root node is copied, and metadata savings because of node deduplication and the ability to store differences in node-granularity. Hence, if a series of snapshots differ slightly and locally, whole subtrees can be shared, which is also beneficial for saving network bandwidth and time for a remote replication process. These two assets allow for a small chunk size that will lead to better deduplication results. On the other hand, the extra levels increase both the latency of a request and the total size of metadata for a full disk, due to the storage space and the I/Os introduced by extra nodes. Partially written disks though, exploit the tree structure and the on-demand allocation mechanism and it is possible that they require less metadata than on the flat-recipe approach.

Comparing with other existing formats, our design does not impose architectural constraints, like the use of a specific filesystem or a filesystem at all, and can be implemented with a variety of underlying storage technologies. Our entities are autonomous and easily movable within the nodes of a distributed cloud environment that constantly rearranges resources for optimizing management and facilitating provisioning. This agility promotes scalability, since intelligent cloud algorithms can avoid performance bottlenecks by optimizing the placement of the entities on the physical storage. Higher level concepts of virtual disks, clones and snapshots are not entangled, but their dependencies are reduced to our rudimentary entities, which strengthens the disk abstraction and allows flexibility. Furthermore, our design promotes a centralized level for

managing all the space-efficiency techniques, which avoids storage and performance overhead of duplicate structures and facilitates monitoring.

The crucial question yet to be answered concerns the configuration of the parameter settings (tree height, chunksize) that will strike a balance between high storage savings, for both data and metadata, and a decent low-latency performance.

To outline a theoretical performance model, guest read requests include three kinds of operations: Firstly, those to access chunks that require $\lceil \frac{request_size}{chunksize} \rceil$ I/Os and so are inversely proportional to the chunk size. Secondly, those to traverse the path from the root of the tree to a leaf and are necessary to find the first chunk of the request. These operations require exactly $tree_height$ I/Os and so are proportional to the tree height. Thirdly, those that access the extra nodes outside the basic path, necessary to find all the chunks related to the request. The exact relation is intricate, but intuitively a decrease in chunk size or an increase in height will increase the number of I/Os. A smaller chunk size will require more entries for the same number of sectors, which in turns translates to more nodes on the bottom level and thus recursively more entries and more nodes on each other level. A taller tree with a constant chunk size, means that the entries per node will be reduced (since $epn = \lceil \sqrt[l]{tc} \rceil$) but the bottom-level entries will remain constant and hence the request-related entries will be distributed to more nodes. Recursively the increased demand for leaf-nodes accesses will trigger more more higher-level entry and node accesses.

The above hold for an individual request, but in reality, a sequence of requests will result in fewer I/Os, because accesses will occur repeatedly on a confined virtual disk area and nodes and chunks will be present in the host-page cache after first access. In a random pattern of small requests, performance is dominated by the tree-height related I/Os (second factor), whereas in sequential patterns performance is dominated mainly by chunk accesses and only secondarily by extra nodes, especially for small chunks and short trees. The percentage of I/Os attributed to extra nodes lies under 15% for trees shorter than 7 levels, and under 2% for trees shorter than 5 levels and chunks smaller or equal than 128KB.

Write requests follow that model, but now in the case of a COW, 3 more extra operations are introduced for a node: two for copying the original chunk -1 read, 1 write- and 1 for updating the parent's write-permissions, assuming that this can be done in 1

I/O for all the node's children (batched). Therefore in the worst case when each node accessed should be COWed, node-related I/Os would be tripled (new write and COW copy write can be merged). Chunks do not have permissions so a COW requires 2 extra operations or just 1 write if the chunk will be fully written. Depending the implementation, more I/Os may be necessary for new node's/chunk's allocation, and those also burden the case of writing on an empty virtual disk.

To conclude, snapshot creation requires 2 I/Os for copying the root node, 1 I/O for writing the new snapshot recipe header and 1/IO for invalidating write permissions on the root node of the primary disk. The latter I/O is unnecessary for clones and the recipe header I/O is also avoided in snapshot restore. We are mainly concerned with the number of I/Os, disregarding the size of each one, since even for 1TB nodes are smaller than 360KB and inspected chunks are smaller than 128KB. Therefore it is the I/O overhead both in kernel and on disk -head movement- that dominates performance, rather than the data transfer time. Of course, especially for big chunk sizes and disks, I/O size can be the reason why performance is not strictly linearly related with chunk size.

0.4 Implementation

We moved on implementing the previous design, by representing all the entities (nodes, chunks, recipe headers) as distinct regular files on the host filesystem and by developing a QEMU backend block driver to incorporate the necessary algorithms for servicing guest requests. Contrary to a qcow2 disk, that is represented by a file that internally encompasses its basic building blocks, our "dtree format" disk is represented by a set of files that collectively provide that abstraction. This file-based implementation is elegant, since it exploits the underlying filesystem's mechanisms for indexing, accessing, allocation and management in terms of storage space. Chunks of all disks reside in a chunk-store directory, nodes of all disks on a recipe-store directory and each primary disk/clone possesses a separate directory where it primary recipe header and all of its later snapshots' recipe headers also reside. These directories also work as distinct namespaces, since chunk and node ids are manifested as pathnames and so are the recipe headers, that are named after the disk/clone/snapshot user given name. Upon

the creation of a new virtual disk, access to a centralized file provides the next available unique disk id.

We did not implement a deduplication system, because of the lack of a large filesystem/image dataset, even with the provision of which, it was unlikely that we would produce any novel results. Nonetheless, we took care of integrating the necessary semantics for a deduplication support in our code, structures and concurrency schemas, so that a future implementation will not require any modifications on our work.

Some optimizations include the following. If a chunk that needs COW, will be fully written in a request, then there is not point in copying the original contents and the overwriting them, so we just create the new chunk and directly write the new data. If a chunk is partially written, then we do not copy the unmodified regions from the original chunk separately, but we copy the whole chunk and then overwrite the desired region. This way, we save unnecessary I/Os. To that end, if more than one entries need to be updated on a parent node, we batch the updates in 1 I/O. For speed up, we keep the file descriptor of the root consistently open, and we use the `O_NOATIME` flag on the filesystem, to avoid metadata updates when just reading files. Finally, we want to preserve the locality within each chunk and node, and so we use the `posix_fallocate()` call to ensure that all data blocks on the file will be sequentially allocated. We also supply a preallocation feature so that first request will be faster and also so that locality of virtual sectors will be preserved on the physical storage.

0.4.1 Block driver and QEMU integration

The QEMU backend block driver (or format-specific block driver) that interacts with the files to service a guest request or offer snapshot/clone functionalities, was written in C, and was integrated in the block device model of QEMU.
According to this, alternating levels of devices and buses form a hierarchy that corresponds to the virtual hardware visible from the guest. The state of a virtual device is represented in QEMU by a generic `DeviceState` structure and an interface-specific structure (e.g. SCSIDevice). As soon as disk devices are recognized by the guest, a `BlockBackend` structure is created by the block backend layer, in order to connect the aforementioned structures (guest-point-of-view), with the format-specific QEMU block drivers that are able to simulate the operation of the disk, by consulting the nec-

essary host entities that store virtual disk's data and metadata (host-point-of-view). In between, a generic block drivers layer acts as an abstraction layer over different disk formats, by specifying a set of functions (API) that the format-specific block drivers can implement at will. Therefore, each guest request collected by the hypervisor, is being reduced to some block backend's layer function, which in turn calls a generic block drivers layer function, which after some preparations finally calls the format-specific block-driver's implementation. As for the chain of structures, the `BlockBackend` structure connected with the `DeviceState`, points to a generic `BlockDriverState` structure that is used to represent generic virtual disk information, among which is a private `BDRV<format>State` structure, used for format-specific data.

Our format-specific QEMU block driver implemented the following functions of the generic block drivers layer API: `bdrv_probe()`, which based on a magic number on the recipe headers deduces if the virtual disk is of dtree format, `bdrv_open()` that is called on virtual disk's mount and initializes the `BDRV<format>State` private structue based on the recipe header and user options, `bdvr_close()` to free memory upon the disk's unplugging, `bdrv_create()` to create the recipe header and root node for a new virtual disk based on user options (preallocation support), `bdrv_co_read()` and `bdrv_co_write()` to translate guest read and write requests to proper operation into host entities (in coroutine context), `bdrv_flush_to_disk()` to flush data buffered on the host page cache to the physical disk, `bdvr_get_info()` to return specific information, `bdvr_snapshot_create()`, `bdvr_snapshot_goto()`, `bdvr_snapshot_delete()` and `bdvr_snapshot_list()`, for snapshot creation, restoration, deletion and listing. For clone support, we implemented our own function, invoked externally, out of the QEMU block drivers context.

0.4.2 Caches and Flush

To service guest read requests, our driver invokes `read()` system calls on chunk/node files, that all invoke a DMA operation that transfers data from disk to the host page cache, before it is moved to the userspace buffer which simulates guest's memory. Data placement on the host page cache is desirable, since it may avoid disk access for future requests. However, on guest write requests, `write()` system calls just transfer data to the host page cache and then return; flush to disk will happen at an undeter-

mined later time. Although it seems that this violates guest semantics, who assumes that data are stored in a persistent way, even physical disks contain a small volatile disk cache for performance optimization (requests rescheduling, prefetching). Therefore, the host page cache can operate as a virtual disk cache, that like the actual disk cache, must be explicitly instructed to flush data in case persistent storage must be guaranteed. QEMU provides a list of caching modes, so that the user can control the involvement and the semantics of the host page cache for each virtual disk. In the default writeback mode, host page cache is enabled and explicit requests are periodically sent from the guest to flush data to physical disk. In unsafe mode, these flush requests are ignored. In writethrough mode, host page cache is enabled to speed up read requests, but all guest write requests must be also flushed to physical disk, before they are considered complete (`O_SYNC` semantics). In none mode, the qemu block driver should issue system calls with `O_DIRECT` semantics that bypass the host page cache. Finally, directsync mode combines `O_DIRECT` and `O_SYNC` semantics. Our driver supports only writeback, writethrough and unsafe mode, to avoid the strict requirements of `O_DIRECT`.

In order to explicitly flush data to disk, a userspace process calls one of `fsync()/fdatasync()` -for a file- or `sync()` -for the whole filesystem- system calls, that beyond creating I/Os to write dirty page cache data, produce a bio struct with the `REQ_FLUSH` or `REQ_FUA` flags set. This bio is handled by the block driver, which issues a special interface-specific command, in order to force the disk controller to flush data from the disk cache to the disk platters (e.g. for SCISI disk, the command is `SYNCHRONIZE CACHE`). Of course, the above are also true for a guest's userspace process or the guest kernel. As soon as the hypervisor collects the special disk cache flush command issued by the guest block driver, calls the `bdrv_flush_to_disk()` function of our specific backend driver, which must itself invoke the appropriate sync-family functions on the host, to ensure that virtual disk's data that reside on host page cache or physical disk cache, will be permanently stored. In writeback mode, `bdrv_flush_to_disk()` is called after every write request.

Our implementation of that function, chooses to use a `sync()` system call that flushes all filesystem's dirty data. This may infringe the principle of VM isolation, since a single VM flush will trigger flushes for all hosted VMs, but is the only viable solution. Flushing each chunk/node file separately, would require redundant metadata I/Os (e.g.

inode blocks will be updated in 131.000 I/Os instead of 1) and since the calls should be synchronous, scheduling could not be employed and the disk head would inefficiently move between each file's metadata, data and journal. Moreover, the overhead of so many system calls would not be negligible and memory would be consumed by keeping a possibly huge list of open files.

0.4.3 Concurrency and locks

Although a disk cannot be accessed simultaneously by two different computers without a special concurrency-aware protocol, the virtual disk's data and metadata, namely chunks, nodes and recipe-headers, can be accessed and modified from various other entities. Beyond the qemu-system process that emulates the VM and handles guest requests, these include one or more qemu-img processes that create, restore, delete and list snapshots, a qemu monitor process that saves the machine's state and thus creates a disk snapshot, a qemu guest agent process that may again create an online snapshot, a clone creation process, an offline deduplication process and a remote replication process. Consequently, locking mechanisms should be implemented to assure disk consistency.

Because all concurrent entities are processes and our shared data reside on files, we used the POSIX fcntl locks to ensure mutual exclusion. These are locks held by a process on a specific file (not file-descriptor), are advisory, meaning that they do not exclude any process not implementing them, can lock file regions, which will allow locking entries instead of whole nodes to increase parallelism, and can come as a read lock, to allow many readers but no writer, or as a write lock, to allow just one writer. A process blocks until it can be able to acquire the lock.

Chunks do not need to be protected, since a lock on their node entries will suffice to guarantee exclusive access. The centralized id provisioning file, is locked on disk creation, to ensure the exclusivity and atomicity of id acquisition and update.

Regarding nodes, a snapshot creation process locks both original and snapshot root node with a write lock. A snapshot restoration or clone creation process locks the snapshot node with a read lock and the clone/primary root node with write lock. All those processes lock all the entries of a node. Qemu-system responsible for servic-

ing guest requests, locks only the node entries that will be traversed, and holds the read or write lock depending on the request type, until DFS traversals of the subtrees will be complete. This means that a running request and a snapshot creation process cannot coexist, since the request should hold a lock on at least one entry of the root node, until it completes. This mutual exclusion, guarantees snapshot's data immutability. Otherwise, a running write request that has passed the root node processing phase, can be interleaved with snapshot creation, and if snapshot creation is completed first, the request will modify theoretically immutable data. A deduplication process, before updating node entries with fingerprints, acquires a write-lock on all its entries to prevent a request from reading/writing data to the node while the update is taking place. That is why a read-lock must be gained on an immutable node, before it is COWed. The mutual exclusion of running requests and snapshot creation can be achieved with a central lock on the recipe-header instead, but our implementation supports the coexistence of a replication or deduplication process on the primary virtual disk. These, just have to lock, using read and write locks respectively, the entries pointing to the subtree corresponding to the disk region that will be processed, which will guarantee the exclusion of any running or future request. However, locking only the entries that are traversed can support the parallel service of a request for another part of the disk.

Concerning recipe headers, to assure atomic creation by only one process, we use the `O_EXCL` flag on the `open()` system call that creates them. Locking the recipe header on each read/write is unnecessary, since an I/O of 512bytes is guaranteed to be atomic. If a read is attempted before the header's initialization, checking the magic number can hint the invalidity.

Online snapshots are not encouraged through the standard qemu monitor procedure which creates incremental images, but are supported through a normal qemu-img invocation. This process increases the value of the next snapshot number, which is necessary for constructing the id for a new COW node. Although it updates the respective field on the recipe header file, it cannot update the respective on-memory field of the `BDVRDtreeState`, which the requests consult in COW. Therefore, to ensure consistency, once every write request that demands COW, the recipe header must be read to acquire the next snapshot number. Considering that snapshot creation and running requests are mutually exclusive, one check per request is enough. Although this polling mechanism is not detrimental to performance, since the recipe header will probably

reside in host page cache, it can be averted by adopting to the qemu online snapshot model, so that the online snapshot creation will signal the qemu-system process that the recipe header has been modified, and the latter will be read only when necessary.

It is possible that more that one requests can be concurrently traversing the tree recipe, and fcntl locks do not ensure exclusion, since the requests are coroutine threads that all belong to the same process. However, in the scenario of two write requests both attempting to COW a node/chunk, because they read the read-only parent entry before it gets to be updated from the first creator, we have a race condition. The second creator will overwrite the whole node/chunk, without encompassing the modifications made by the first creator. To avert that, we can either consider this a rare scenario and force the second potential creator to restart the request, so that the first gets to complete the COW (our approach), or serialize the node/chunk creation requests by inserting them on a list (qcow2 approach). Alternatively, we could load all nodes' entries to a block driver cache and we could ensure atomic node/chunk creation, with a mutex at each entry. Because nodes are updated after the completion of their COW children, the virtual disk is also crash-consistent.

0.5 Experimental evaluation

We will measure system performance in terms of bandwidth for a bulk of requests issued and average latency of a request, under different configurations. These are determined by 4 parameters; the chunk size, the height of the tree recipe, the disk size and the state of the disk, namely whether we write on an empty disk (unallocated), on a allocated disk without performing any COW (allocated) or on a recently snapshoted disk so each write triggers a COW operation (COW). We use a different physical disk (HDD 5400 rpm, 8MB disk cache) for data and metadata, in order to get rid of the noise introduced by irrelevant I/Os produced by the host OS. We use a VirtIO interface for our virtual disk, a writeback caching mode to avoid constant disk flushing, and we take care emptying both host and guest inode, dentry and page caches before each run. We use a ext4 filesystem with indexing enabled for faster node/chunk lookup, noatime option to avoid unnecessary timestamp updates and writeback journaling to avoid a time-consuming fsck to restore metadata consistency in case of a crash.

0.5.1 Micro-Benchmarks

Micro benchmarks will allow the monitoring of the system's performance under specific I/O patterns over different configurations. We use the flexible IO tester workload generator, we issue 4KB I/Os, we bypass the guest page cache (direct I/O) and we explicitly flush data in the end.

In accordance with the theoretical model, the determinant performance factor is the number of files accessed, and secondarily the amount of data read/written. To extend the model, access on a file is even more costly as it entails extra I/Os for inode updates, lookup in directory data blocks, and possibly journal writes. Of course depending the access pattern, the host page cache may have a significant impact as well.

We will gradually narrow the window for the values of the varying parameters by omitting the inefficient ones, thus also dealing with the combinatoric explosion.

Chunk size

An increase on the chunk size, is expected to improve performance, for a constant disk and access pattern, as less files will be accessed. This is not true just for chunks, but also for nodes, which are normally reduced. Less chunks also mean less entries to be read and faster fingerprint index lookup in deduplication. We will explore the implication on both a flat and a 3-level tree recipe.

When we perform a sequential 1GB write, we observe no significant performance gain over 64KB in clean disk and over 128KB for COW disk, as opposed to the previous steep performance increase. In an allocated disk , both for read and write, we have a steady increase, more intensive under 16KB. Increasing the chunk sizes by a factor of two, may cut file accesses to half, but does not double performance, because the data that need to be written is constant and does play a role too. Sequential reads/writes, is the only case that the chunk size increase brings a moderate, instead of a significant, performance boost and that is mainly attributed to the lack of extra I/Os necessary for fie creation (clean/COW) or copying (COW). Needless to say that COW case is at least 1.5 times slower than the allocated case.

When we performed a 50M write on random sectors among a 10GB range on a clean disk, the results form a concave curve, with a peak at 16KB. This is the blending of

two conflicting factors. Firstly, for a bigger chunk size requests are distributed to less files that are more frequently accessed, and because those files' entries will be in cache after first access, the bigger the chunk size the more the metadata cache utilization. On the other side, the disk is clean and chunks are allocated on demand, but because the pattern is random, doubling the chunk size does not reduce by two the number of necessary chunk-files. Thus the product of number of chunks and chunk size, which constitutes the physical disk space occupied, increases with the chunk size. This physical region extension, harms bigger chunks, since it intensifies disk head movement.

To eliminate the impact of physical placement, we neglect random clean case, and we focus our random pattern experiments on a sequentially allocated disk (preallocated disk). Indeed, because of the higher cached metadata utilization for bigger chunks, both random read and writes on an allocated disk, feature a consistent performance increase. However in the COW case, the copying of a chunk that involves two extra I/Os analogous to the chunk size, counterweights the caching effect, leading to a concave curve, with a peak at 64KB.

The above results apply to a 3-level recipe as well, since nodes also dwindle with chunk size increase. Excluding sequential writes, the tree recipe does not introduce a substantial performance gap, comparing with its flat counterpart. This is because chunk operations dominate over the I/Os concerning the recipe. A taller tree will benefit more from a bigger chunk size, since the difference between 3-level and flat recipe on the 4KB - 64KB region lies among 15%-35%, whereas on the 128KB-1MB region is merely 10%.

As overall comments, we state that:

- there are not substantial performance improvements over 128KB
- performance is unacceptably low for 4KB and 8KB chunks
- chunk size greatly affects performance, except from sequential writes

For now on, we will focus on the region of 16KB-128KB, so that can still achieve high deduplication gains without sacrificing a great deal of speed.

Tree height

We expect that an increase of the tree height will degrade performance, because for a constant disk and chunk size, more nodes and thus more I/Os will be involved. For one thing, every new level added increases by one the path involved in each single

chunk access. This increases both file accessed and data blocks read/written. In addition, in the general case, a large request involves reading a lot of entries, and because a higher tree entails less entries per node, the required entries in the bottom level will be spread across more nodes, thus increasing file accesses. Indexing those increased leaf-nodes involves more upper-level entries, which may also be spread to more upper-level nodes. Our assumption that taller trees degrade performance holds true, and we will now explore the experimental results in more depth, to quantify the performance loss.

In the clean disk case, the chunk size has a greater impact on performance than the tree height, which is normal since the number of chunk-files that need to be created outweighs by far the number of nodes, and the creation time dominates the tree traversal time. It is indicative that the bandwidth for a 10-level tree with chunk size x, is better than that of a flat recipe with the previous chunk size $\frac{x}{2}$. Notably poor is the 16KB performance, 32KB could be tolerable and for all chunk sizes we experience a performance drop after 3 levels.

In the allocated case of sequential reads/writes, chunk size differences are mitigated, apart from 16KB which is still rather inferior. For each chunk size, we observe only a slight performance loss up to 3 levels, except for the 64KB drop from 2 to 3 levels. Excluding again 16KB, flat recipe and 2-level recipe exhibit very similar average latencies, which comes as no real surprise, since the 2-level root node is basically stored in RAM. This is not true for bandwidth of sequential writes for the following reason. Height increase leads to the distribution of entries to more files, which brings about more I/Os to journal and inode blocks, that additionally disturb the local movement of the disk head. This is also why performance degradation with height increase, is much more severe for sequential writes than reads.

In accordance to sequential patterns, random reads/writes experience a noteworthy performance loss above 3-level recipes and produce similar results between a flat and a 2-level recipe, due to root node caching. However, now for small chunk sizes (16KB, 32KB), 3-level recipe is noticeably worse than its 2-level counterpart, probably because the 3-level tree engages a lot of 2-level nodes that are accessed one or few times, thus reducing the cache efficiency.

In COW case, because the chunks are much more than the nodes, their COW time dominates, "hiding" the node COW time and other differences between a flat and a

2-level recipe, that again perform similarly. From 2-level to 5-level trees, performance is gradually deteriorating, with 3 levels being relatively close to 2, with the exception again of 16KB, 32KB for random read/writes.

To sum up:

- 2-level and flat recipe perform similarly (excluding 16KB chunks), because of root node caching
- performance degradation is generally tolerable up to 3-level recipes
- performance drop from 2 to 3 levels is detected mainly for random read/writes (allocated and COW) with 16KB and 32KB chunks, secondarily for sequential COW and finally for sequential writes for clean/allocated disk with 64KB chunks

Fueled by the first conclusion and the fact that the 2-level recipe can annihilate snapshot creation time, we strongly suggest its use over a flat recipe. The second conclusion sets an upper limit to the number of levels that do not severely degrade performance, so from now on, we will focus on comparing the 2 and 3 level recipes, particularly to find out if the latter can produce significant storage savings that can justify a slight performance loss.

Snaphot creation time

To solidly prove our claim of zero snapshot creation time for all tree recipes, we conducted measurements for a 1TB virtual disk. A flat recipe needs for 4KB to 128KB chunks, 255s, 130s, 70s, 33s, 17s and 9s respectively. On the contrary, a snapshot on a 2-level recipe takes 0.7-0.77s for all chunk sizes except 4KB, which demands 0.95s and a 3-level recipe demands 0.68-0.71s for all chunk sizes. Snapshot times of tree recipes on a 500GB disk are alike, fluctuating around 650ms. Analogous are the results for a clone creation operation, which discharged from invalidating source write permissions, produces slightly lower times. Even for large disks, we witness that creation time can be considered instant indeed, with hardly any deviations or height dependencies.

0.5.2 Metadata

We will now study the metadata storage overhead for a full disk, in respect with chunk size and tree height. A first dichotomy, divides metadata into physical, which are re-

quired for the mapping between entities and their physical storage, and logical, which reside on recipes are required to map virtual sectors to entities. In our case of file-entities, physical metadata consist of the inode size and possibly directory entries, which cumulative add up to 304bytes. However we will occasionally ignore those, since a better design could diminish that value to a negligible extent (e.g. 28bytes). Logical metadata consist of the node headers (16bytes) and the recipe entries (22bytes). Another classification separates the "raw" metadata, which comprise the recipe entries necessary to index the chunks and are solely dependent on the chunk size, and the tree-related metadata, which are composed from all the node headers and from the recipe entries of all but first level nodes.

Metadata calculation

"Raw" metadata are compulsory for a given disk size and chunk size, and tree height just determines the number of nodes that they will be distributed into. Because of the linear relation $chunk_entries = \frac{disk_size}{chunksize}$, there is a exponential decrease for each chunk size transition (considering chunks evenly spaced in a x-axis). For 4KB chunks, "raw" metadata constitute 0.5% of total virtual disk data, for 8KB 0.28% and for 16KB and above, constitute under 0.12%. Including physical metadata, we reach a 8% for 4KB and then as chunk size increases, the percentages face a constant reduction by approximately a power of two. A deduplication approach will hold about 28bytes for each fingerprint index entry, so these percentages drop by a factor of about 10. It is obvious that, especially excluding physical, "raw" metadata constitute a small fraction of the disk size, for all the chunk sizes greater than 8KB, which we have chosen.

Tree-related metadata are increased with taller trees for two reasons. Keeping the chunk size and the disk size constant, the number of chunks is unaffected and so are the "raw" metadata, namely the entries of the bottom level necessary for indexing the chunks. However because height increases, from $epn = \lceil \sqrt[h]{tc} \rceil$ we derive that each node will contain less entries, and so the first level will distribute its constant entries to more nodes. This increases the indexing needs of the second level, and so more entries must be added, thus inflating metadata. Recursively this holds true for all levels up to the root, which is a new node. Hence, the first reason is the upsurge in recipe entries for each level -except the first-, compared to the entries on that level for a shorter tree, plus of course the new root's extra entries. Moreover, because epn decreases, all levels

distribute their entries to more nodes than the corresponding level on a shorter tree, is it the first that has constant entries or an upper that has even more. This gives rise to the second reason for metadata escalation, which is that because the number of nodes rises, consequently so does the node headers' overhead.

2-level recipe metadata hardly differs from the flat one, since only a few-kilobye node is added, and rather limited is the 3-level recipe overhead as well. Beyond that point, we can mark a constant metadata increase with taller trees, that due to the mathematical anomalies of the ceiling function, cannot fit any consistent pattern and varies with disk size. However, for 5-level trees and below it is still the chunk size that remains the prominent factor for metadata storage costs. Note that we refer to full disk metadata.

Metadata savings

Due to the COW technique, only modified nodes are allocated after a snapshot. The smaller the nodes are, the more-fine grained is the entry-analysis and the higher are the prospects to isolate modified entries and copy just few unchanged ones. Because of the tree recipe, finer detection can lead to more effective subtree sharing as well. Although, as described, taller trees lead to smaller nodes, they also imply more nodes and so an increased metadata overhead due to extra node headers and entries. Furthermore, the I/O pattern may be scattered, not encouraging subtree sharing and modification accumulation to distinct nodes. Consequently, we must measure the metadata introduced from COW between two snapshots of a disk for different tree heights and chunk sizes, and deduce whether taller trees do actually store less metadata.

Finer-grained nodes, can also help a replication process save network bandwidth and time, again because the modified nodes to be sent will contain less unchanged entries and effective subtree sharing will instantly eliminate huge metadata transmission. Metadata modified between two snapshots correspond to metadata that need to be sent, so the effectiveness indicator is identical to the the storage savings case.

To that end, we utilized real-world I/O traces from SNIA IOTTA repository, for 36 volumes of 13 web and data servers, over 1 week. We then developed a program that simulated the node accesses for write I/Os, over different configurations (tree height, chunk size), thus measuring the amount of metadata modified.

Firstly, for most servers -especially data servers-, we observed that changes happened

mostly in the first 4 hours, and then the metadata modified remained rather stable for the next 3 days, which indicates that writes occurred repeatedly on the same data, placed on a locally confined disk region. Therefore, any timing for snapshot creation after 4 hours, will be equivalently representative. Henceforward we use a 7-day period.

Regarding the tree height, for all volumes a 3-level recipe is either extremely or slightly better than a 2-level recipe, but always capable of storing less metadata. Savings are outstanding particularly for project, web staging, user and media servers. 4-level recipes are only marginally superior than 3-level, and 5-level are either scarcely better than 4-level or sometimes even worst, engaging more metadata. The superiority of the 3-level recipe over its 2-level counterpart, is mitigated but still retained, even if we take into account the physical metadata involved in every node stored.

The metadata savings using 3 instead of 2 levels, are mitigated as chunk size increases, which indicates that especially for small chunk sizes, a 3-level recipe should be employed to avoid metadata explosion. This contradicts with the ominous random pattern performance results of a 3-level recipe for 16KB and 32KB chunks. Moreover, in volumes that contain data, the local nature of changes makes the use of a 3-level recipe more important than the choice of the chunk size, in terms of metadata storage overhead. In OS volumes the opposite is the case, because of scattered writes.

Now that we established that taller trees are more efficient at storing metadata changes, it is just a matter of how soon the metadata savings from a series of snapshots will recoup the initial full-disk storage overhead over a shorter tree. Studying all the combinations of different volumes and 16KB-128KB chunks, we conclude that a 3-level recipe seems to be the best solution. A 3-level over a 2-level recipe can always recoup the initial overhead in 1-9 snapshots (2.5 on average). A 4-level over a 3-level recipe is only more efficient on 75% of the volumes, requiring 31 snapshots on average, while a 5-level over a 4-level recipe is superior on 75% of the volumes requiring 93 snapshots on average. A 4-level over a 2-level recipe can recoup the initial overhead in 9.7 snapshots on average. These snapshot numbers drop even further by excluding physical metadata, to 1 snapshot for 3 against 2 levels and to 5.4 snapshots for 4 against 3 levels.

To recap, the tree-recipes can compensate for the small amount of their initial full disk tree-related metadata, after a series of very few snapshots, thus averting metadata explosion and assisting a replication process. The 3-level recipe can yield excep-

tional metadata savings comparing to a 2-level recipe, and although the chunk size is exponentially related to metadata, the overhead for 16KB chunks and above can be reclaimed through deduplication savings.

0.5.3 Macro-Benchmarks

We will now use workloads that simulate scenarios of real-world I/O patterns, starting with the Postmark benchmark that simulates a web/mail server, by issuing transactions to many small files. The dominant performance factor is the chunk size, with the 3-level recipe performing almost identically to the 2-level one for the same chunk size. This holds true for COW, allocated and warm cache cases, with COW case being quite more favored by big chunk sizes (64KB, 128KB) and allocated case by 128KB.

We also used the Pgbench benchmark to simulate PostgreSQL database operations, in an allocated disk. For 64KB and 128KB chunk sizes the 3-level recipe performance is close to the 2-level, which does not stand true for 16KB and 32KB, where there is a huge decline. This can be attributed to the fact that small chunk sizes struggle with random patterns. 128KB, 64KB, and even 32KB performance are rather close for 2-level recipes. From the database initialization times, we confirm that for big chunk sizes 2 and 3 levels perform alike, which is not again the case for small ones. Moreover, 32KB case does face a performance fall to the similar 64KB and 128KB ones, but not as sharp as the 16KB case.

Next we used the fileserver personality of the filebench benchmark, which creates and processes many files in a multi-threaded manner. While big chunk sizes are substantially faster than small ones, the differences in each group are negligible. Insignificant is also the performance drop from 3-level to 2-level recipes, regardless the chunk size.

The small differences between 2-level and 3-level recipes in macro-benchmarks, can be attributed to the host page cache, which after the first node access stores it and does not burden performance. It can also be explained because chunk operations dominate the bulk of I/Os. This paves the way for the use of 3-level recipes, that are much more appropriate for saving metadata.

0.5.4 Conclusion and future directions

Our tree recipe efficiently supports sparseness, deduplication (on metadata as well) and instant, autonomous and movable snapshots and clones, without imposing any architectural restrictions. The tree structure allows for a small chunk size, which is important to efficiently eliminate data redundancy, without compromising snapshot/clone creation time or leading to metadata explosion after a series of snapshots.

The power of the 2-level tree recipe is undeniable, since with a low performance and metadata overhead, enables instant snapshots and metadata savings for a series of snapshots. These savings can be even increased by a 3-level recipe, which after very few snapshots recoup the extra full-disk metadata overhead and substantially assist a replication process. These advantages combined with the insignificant or tolerable performance drop from the 2 levels, as the macro-benchmarks confirm, suggest the use of a 3-level recipe. Higher trees burden performance, providing only low-rate or even no metadata savings.

In our chunk size choice, for chunks of 16KB and above we should not take into account the metadata overhead, since it is limited and deduplication gains will compensate for it. The main tradeoff is the performance versus data savings since the smaller the chunk size, the lower the performance and the higher the storage savings from deduplication and COW sharing. We exclude chunk sizes greater than 128KB, since they deteriorate deduplication without offering a significant performance boost. Focusing on the 2 and 3 level cases, performance for 16KB is considerably worse than 32KB-128KB, especially for a clean disk. Moreover, the difference between 64KB and 128KB is particularly small, especially in terms of latency.

Overall, we conclude that a 3-level tree with 64KB chunk size can be an appropriate choice to strike a balance between sustainable performance and considerable storage savings. If we are eager to sacrifice significantly deduplication gains, 128KB chunks can be used for a slight performance boost and avoidance of some pathological 64KB scenarios. If we want to emphasize on the storage savings, 32KB chunks could be used, but it would be better to avoid 16KB, with which performance plummets.

The results produced can be generalized outside the context of our implementation, which treats entities as host files. An alternative implementation could avoid some

of the current drawbacks, such as the maximum entities limit imposed by the inode number limit, and the whole filesystem flushing -sync()- that creates performance dependencies between the I/O loads of different VMs. In addition, the on-demand chunk allocation entails a random physical placement of the chunks that may be far from retaining virtual disk's locality. As a result, guest patterns are distorted, performance predictability is undermined and guest optimizations for sequentiality are in vain. On the contrary, a distributed storage system in the place of the host filesystem and directly attached disk, will be unaffected by locality disturbance, as it is based on parallelism to deliver performance. We highlight that the main setback of our implementation is the excessive filesystem operations pertinent to file manipulation, such as inode updates, and that is why our results are inferior than qcow2 format, which engages only a single file. If we place our entities directly on raw storage or as objects of an object-storage, we will reap all the benefits described, without being slowed down by the host filesystem.

This thesis gives also rise to opportunities for future research. A deduplication system could be integrated so that deduplication savings for both data and metadata could be quantified, more multi-threaded benchmarks could be run to make sure that our locking mechanism is congestion-free, and on-demand chunk allocation instead of preallocation could be adopted in order to monitor the repercussions of the disturbance of virtual disk's locality on performance. Moreover the modestly sized recipes could be hosted in an SSD disk so that recipe traversal time would be minimized, or even better, SSDs could be used to host the hot data and metadata of primary virtual disks (and clones), whereas snapshots with their immutable data and metadata could be moved to HDDs. Because our test partition had only one virtual disk allocated when benchmarks were run, it would be interesting to study the effect of virtual disks and chunk/node aggregation on performance, since it is likely that lookup time and fragmentation induced by COW would both increase; if the effect is severe, the arrangement of files on directories should be readjusted. Finally, the use of a private cache in our specific block driver could be examined, in order to improve performance and implement a better mechanism for protection against simultaneous chunk/node allocations. Again we stress that our design would benefit the most from an alternative storage implementation, independent from the host filesystem.

Keywords: virtual disk, virtual disk format, snapshot, clone, deduplication, Copy-On-Write, storage virtualization, virtual machine, cloud, QEMU

1 Εισαγωγή

1.1 Διατύπωση του προβλήματος

Το πρόβλημα που εξετάζουμε είναι η υποστήριξη εικονικών δίσκων για εικονικές μηχανές, οι οποίοι θα μπορούν να υλοποιούν αποδοτικά σε χρόνο και χώρο λειτουργίες στιγμιοτύπων (snapshots), κλώνων (clones) και απαλοιφής διπλοτύπων (deduplication). Μέχρι στιγμής, ενώ υπάρχουν τεχνολογίες για υποστήριξη εικονικών δίσκων που υλοποιούν όλα τα παραπάνω ξεχωριστά, καμία δεν τα συνδυάζει ενιαία, εκμεταλλευόμενη τη σημασιολογία των απαιτούμενων λειτουργιών και προσφέροντας απαγκίστρωση από συγκεκριμένες αρχιτεκτονικές επιλογές.

1.2 Κίνητρα

Στα σύγχρονα και διαρκώς αναπτυσσόμενα περιβάλλοντα υπολογιστικού νέφους (cloud) που χρησιμοποιούν εικονικοποίηση, η παροχή εικονικών δίσκων που σέβονται τις απαιτήσεις απόδοσης και ταυτόχρονα προσφέρουν υψηλή εξοικονόμηση χώρου, είναι ζωτικής σημασίας. Με ολοένα και περισσότερα κέντρα IaaS με χιλιάδες υπολογιστές και συστοιχίες δίσκων, η εξοικονόμηση κόστους αποτελεί προφανώς διακαή πόθο. Η τεράστια δεξαμενή εικονικών μηχανημάτων στα περιβάλλοντα υπολογιστικού νέφους, δημιουργεί πλεονασμό ο οποίος θα ήταν ανόητο να μην εξαλειφθεί. Από την άλλη, αυτό οφείλει να μην γίνει θυσιάζοντας λειτουργίες του σύγχρονου storage όπως τα στιγμιότυπα και οι κλώνοι που αδιαμφισβήτητα θα παραμείνουν στις πρώτες γραμμές των απαιτήσεων, μιας που προσφέρουν τρομερή ευελιξία και χρηστικότητα.

1.3 Ελλείψεις

Απλές ιδέες για εξοικονόμηση χώρου, όπως τα αραιά αρχεία (sparse files), αποτυγχάνουν στο να παρέχουν μια κλιμακώσιμη λύση, αφού δεν έχουν δυνατότητες απαλοιφής πλεονασμού και αρκούνται στο να μην πιάνουν χώρο στον πραγματικό δίσκο για αδέσμευτες περιοχές του εικονικού δίσκου. Ενώ η τεχνική COW μπορεί να υποστηρίξει αποδοτικά στιγμιότυπα και κλώνους απαλείφοντας τον χρονικό πλεονασμό μεταξύ των διαφορετικών στιγμιοτύπων ενός εικονικού δίσκου, αφελείς υλοποιήσεις όπως εικονικοί δίσκοι πάνω από ένα σύστημα αρχείων BTRFS που μπορεί να υποστηρίξει στιγμιότυπα με COW, είναι ασύμφορες από πλευράς απόδοσης και επιβάλλουν περιορισμούς στην αρχιτεκτονική του συνολικού συστήματος. Το qcow2, που είναι η προτεινόμενη λύση για υποστήριξη στιγμιοτύπων σε εικονικούς δίσκους για QEMU, συνδυάζει και τα αραιά αρχεία και την τεχνική COW στα στιγμιότυπα και τους κλώνους, αλλά δεν υποστηρίζει την αυτονομία των στιγμιοτύπων και των κλώνων και ούτε επιτρέπει με έξυπνο και διαφανή τρόπο την συνεργασία με ένα σύστημα απαλοιφής διπλοτύπων. Αυτό γιατί τα δομικά κομμάτια στα οποία χωρίζει τον δίσκο για να εφαρμόσει την τεχνική COW (clusters), είναι τμήματα από bytes εσωτερικά σε ένα αρχείο και άρα δεν υποστηρίζουν ανεξαρτησία των στιγμιοτύπων, ενώ για να υπάρξει συνεργασία με ένα σύστημα απαλοιφής διπλοτύπων, χρειαζόμαστε ένα επιπλέον επίπεδο που θα εκμεταλλεύεται την σημασιολογική πληροφορία αυτών των τμημάτων και θα συνεργάζεται με την διαφανή διεπαφή ενός συστήματος απαλοιφής διπλοτύπων. Παράλληλα, η απόδοση επιβαρύνεται από πολλαπλές διασχίσεις δομών μεταδεδομένων και πλεοναστικές ενημερώσεις σε αυτές. Οι κλώνοι του qcow2 είναι και αυτοί εξαρτημένοι από το αρχικό στιγμιότυπο, οπότε έχουμε και πάλι προβλήματα αυτονομίας και πολλαπλών διασχίσεων.

1.4 Σχεδιασμός

Για να αντιμετωπίσουμε όλες τις παραπάνω ελλείψεις και να συνδυάσουμε αρμονικά τις λειτουργίες στιγμιοτύπων, κλώνων και απαλοιφής διπλοτύπων, προχωρήσαμε σε ένα απλό αλλά κομψό και διαφανή σχεδιασμό που χρησιμοποιεί όλες τις παραπάνω ιδέες. Αντίθετα με άλλες προσεγγίσεις που ταυτίζουν ένα εικονικό δίσκο με ένα αρχείο, εμείς τον ορίζουμε ως μια συλλογή από τεμάχια σταθερού μεγέθους. Δεσμεύ-

ονται όσα τεμάχια απαιτούνται με βάση τις περιοχές του εικονικού δίσκου που έχει γράψει η εικονική μηχανή, οπότε η υλοποίηση έχει την ιδιότητα του sparseness ακριβώς ανάλογη με αυτήν των αραιών αρχείων. Επίσης, όταν ο χρήστης πάρει ένα στιγμιότυπο του εικονικού δίσκου, όλα τα τεμάχια σημειώνονται ως μόνο για ανάγνωση, οπότε όταν ο χρήστης θελήσει να γράψει σε ένα τέτοιο τεμάχιο, εκείνη την στιγμή αυτό αντιγράφεται και γράφει στο νεοσύστατο. Κατ' αυτόν τον τρόπο διατηρούμε αμετάβλητα (immutable) όλα τα τεμάχια που ανήκουν σε στιγμιότυπα, εφαρμόζοντας την τεχνική COW και υποστηρίζοντας έτσι χωρικώς και χρονικώς αποδοτικά στιγμιότυπα και κλώνους. Παράλληλα, δίνουμε την δυνατότητα σε ένα σύστημα απαλοιφής διπλοτύπων να επεξεργαστεί τα immutable τεμάχια και να απαλείψει τον πλεονασμό μεταξύ και διαφορετικών εικονικών δίσκων. Η αντιστοίχιση μεταξύ περιοχών του εικονικού δίσκου και αρχείων που τον απαρτίζουν πραγματοποιείται μέσω μιας «συνταγής», η οποία στην απλή περίπτωση είναι ένας πίνακας που με γραμμικό τρόπο αντιστοιχίζει τομείς σε τεμάχια και στην γενική περίπτωση είναι ένα δέντρο κόμβων, με τους εσωτερικούς κόμβους να δείχνουν σε άλλους κόμβους και τους κόμβους φύλλα να δείχνουν σε τεμάχια. Με την δενδρική δομή, κατορθώνουμε να μειώσουμε το μέγεθος των μεταδεδομένων που πρέπει να αντιγράψουμε κατά την δημιουργία ενός κλώνου ή στιγμιοτύπου, και μπορούμε να επιτρέψουμε ακόμα μεγαλύτερο διαμοιρασμό κοινών δεδομένων και μεταδεδομένων. Ο παραπάνω σχεδιασμός υλοποιήθηκε με τα τεμάχια και τους κόμβους του δέντρου συνταγής να είναι αρχεία στο σύστημα αρχείων του φιλοξενούν μηχανήματος, και η λογική αντιστοίχισης και διαχείρισης στιγμιοτύπων και κλώνων ενσωματώθηκε στον κώδικα του QEMU, με την βοήθεια ενός οδηγού συσκευής μπλοκ (block driver).

1.5 Αποτελέσματα

Η παραπάνω σχεδίαση, μετρήθηκε με την βοήθεια micro-benchmarks, macro-benchmarks και ίχνη πραγματικών λειτουργιών Ε/Ε, έτσι ώστε να διερευνήσουμε τις παραμέτρους που θα οδηγήσουν σε μια μορφή εικονικού δίσκου που ενόσω θα εξοικονομεί χώρο θα σέβεται τις υψηλές απαιτήσεις για γρήγορη απόκριση. Καταφέραμε να καταλήξουμε σε ένα σύνολο παραμέτρων για το ύψος της δενδρικής δομής και το μέγεθος των τεμαχίων, που σε συνδυασμό με τις θεωρητικές παρατηρήσεις πάνω στην αποδοτικότητα της απαλοιφής διπλοτύπων, μπορούν να παρέχουν ένα αξιόλογο αποτέλεσμα

στα πλαίσια των παραπάνω στόχων. Η μελέτη μας αυτή, εγείρει επίσης ερωτήματα για την χρηστικότητα ενός συστήματος αρχείων σε ένα σύστημα παροχής εικονικών δίσκων σε περιβάλλοντα υπολογιστικού νέφους. Τα παραπάνω δημιουργούν λοιπόν πρόσφορο έδαφος για περαιτέρω έρευνα και υλοποιήσεις.

2 Υπόβαθρο

2.1 Αποθήκευση δεδομένων

2.1.1 Μέσα και τεχνολογίες αποθήκευσης

Η κύρια λειτουργία των ηλεκτρονικών υπολογιστών είναι η επεξεργασία δεδομένων. Για να εκτελέσει όμως υπολογισμούς στα δεδομένα, ο επεξεργαστής (CPU) χρησιμοποιεί μια ιεραρχία μέσων αποθήκευσης για να αποθηκεύει και να ανακαλεί τα δεδομένα αυτά. Στην κορυφή της ιεραρχίας βρίσκεται η κύρια μνήμη (RAM), η οποία αν και γρήγορη, είναι πτητική, δηλαδή μετά την διακοπή της τροφοδοσίας χάνει όλα τα δεδομένα της. Έτσι, για να την μόνιμη αποθήκευση δεδομένων που θα διατηρούνται στις περιόδους μη λειτουργίας του υπολογιστή, αναπτύχθηκαν τεχνολογίες μη-πτητικής μνήμης (Non-Volatile Memory). Τέτοιες τεχνολογίες περιλαμβάνουν την μαγνητική ταινία, τους μαγνητικούς δίσκους, τα οπτικά μέσα (CD, DVD), τις μνήμες flash, ακόμα και τις read-only μνήμες (ROM) [GK12].

Η πρώτη τέτοια τεχνολογία που χρησιμοποιήθηκε ως πρωτεύων χώρος αποθήκευσης (primary storage) ήταν η μαγνητική ταινία που αναπτύχθηκε το 1928 και χρησιμοποιήθηκε μέχρι την δεκαετία του 1950 οπότε και ήρθαν στο προσκήνιο οι μαγνητικοί δίσκου. Η αντικατάσταση αυτή έγινε επειδή, βασιζόμενη στα πρότυπα της καταγραφής και αναπαραγωγής μουσικής, η μαγνητική ταινία δεν επέτρεπε γρήγορη τυχαία προσπέλαση δεδομένων. Χρησιμοποιείται όμως ακόμα και σήμερα χαμηλότερα στην ιεραρχία, λόγω του χαμηλού κόστους της.

Οι μαγνητικοί δίσκοι ή αλλιώς σκληροί δίσκοι (HDD) αποτελούν την πλέον διαδεδο-

μένη επιλογή για τον πρωτεύοντα χώρο αποθήκευσης. Αποτελούνται από κάποιους κυκλικούς δίσκους (platters), καλυμμένους και στις δυο επιφάνειές τους με ένα μαγνητικό υλικό, οι οποίοι περιστρέφονται γύρω από ένα άξονα με σταθερό ρυθμό, από 5.400 έως 15.000 RPM. Οι κυκλικοί δίσκοι διαιρούνται λογικά σε ομόκεντρες κυκλικές τροχιές (tracks) οι οποίες υποδιαιρούνται σε τομείς (sectors). Κάθε δίσκος συνοδεύεται από μια μαγνητική κεφαλή (head) που χάρη σε ένα βραχίονα μπορεί να κινείται μεταξύ των τροχιών, έτσι ώστε σε συνδυασμό με την περιστροφή του κυκλικού δίσκου να διαβάζει ή να γράφει δεδομένα στους τομείς. Ο τομέας είναι η βασική δομική μονάδα ανάγνωσης και εγγραφής δεδομένων, έχει μέγεθος συνήθως 512byte και παρέχει ατομικότητα[1]. Η παραπάνω γεωμετρία, προωθεί την ανάλυση της ταχύτητας του δίσκου σε τρεις βασικές μετρικές: τον χρόνο αναζήτησης (seek time), που είναι ο χρόνος που θέλει η κεφαλή για να μεταβεί στην κατάλληλη τροχιά, την περιστροφική καθυστέρηση (rotational latency), που είναι ο χρόνος που κάνει ο δίσκος να περιστραφεί μέχρι η κεφαλή να βρεθεί στον κατάλληλο τομέα, και τον ρυθμό μεταφοράς που εκφράζει το πόσο γρήγορα μπορούν να διαβαστούν τα δεδομένα του τομέα[2]. Ο χρόνος αναζήτησης είναι τυπικά γύρω στα 4ms-12ms, η περιστροφική καθυστέρηση γύρω στα 3ms-5ms και οι ρυθμοί μεταφοράς είναι το πολύ μέχρι 250MB/s. Γι᾽ αυτό τον λόγο άλλωστε οι σκληροί δίσκοι έχουν τρομακτικά καλύτερη απόδοση σε σειριακές λειτουργίες Ε/Ε παρά σε τυχαίες. Οι προαναφερθείσες ταχύτητες είναι γύρω στις 6 τάξεις μεγέθους μικρότερες από αυτές της κύριας μνήμης και συνεχίζουν να αυξάνονται με χαμηλότερους ρυθμούς, οδηγώντας έτσι σε ένα μεγάλο χάσμα απόδοσης. Παρόλα αυτά οι σκληροί δίσκοι υποστηρίζουν υψηλή πυκνότητα δεδομένων με μεγάλες χωρητικότητες τάξης terabytes, προσφέρουν υψηλή αξιοπιστία και έχουν χαμηλό κόστος, οπότε συνεχίζουν να κυριαρχούν ως λύση στον τομέα της πρωτεύουσας αποθήκευσης.

Σε σχέση με τις παλαιότερες μαγνητικές ταινίες, οι σκληροί δίσκοι επιτρέπουν εύκολα τυχαίες προσπελάσεις τομέων, αλλά λόγω της μηχανικής κατασκευής τους, εισάγουν σε κάθε μια τέτοια προσπέλαση ένα κόστος αρκετών milliseconds. Αυτό το κόστος εξαλείφεται με την εισαγωγή των μνημών flash, στις οποίες η εγγραφή/διαγραφή πραγματοποιείται με την προσθήκη/αφαίρεση ηλεκτρονίων σε ένα solid-state cell που κατασκευάζεται από μια floating gate, ενώ η ανάγνωση πραγματοποιείται με την μέτρηση

[1] ο τομέας είτε γράφεται ολόκληρος είτε καθόλου

[2] υπάρχει και ο ρυθμός μεταφοράς από τον ελεγκτή του δίσκου στον ελεγκτή της διεπαφής του δίσκου, αλλά αυτός είναι συνήθως πολύ ψηλότερος και δεν αποτελεί bottleneck

της τάσης της πύλης. Η πιο ευρέως χρησιμοποιούμενη συνδεσμολογία μεταξύ των πυλών ακολουθεί των σχεδιασμό NAND και η προσπέλαση των δεδομένων γίνεται σε μπλοκ των 256KB-4MB. Οι μνήμη flash χρησιμοποιήθηκαν αρχικά σε κάρτες SD και usb flash drives, αλλά από τις αρχές του 2000 χρησιμοποιείται για την κατασκευή solid-state-drives (SSD), που έχουν μεγάλη χωρητικότητα και υπόκεινται στα πρότυπα των λειτουργιών εισόδου/εξόδου των κλασσικών σκληρών δίσκων, με σκοπό να μπορούν εύκολα να τους αντικαταστήσουν. Η απουσία μηχανικά κινούμενων κομματιών, δίνει στους SSD δίσκους χρόνο τυχαίας προσπέλασης από 20μs και ρυθμό μεταφοράς από 300MB/s έως κα 1500MB/s, ενώ τους κάνει και πιο ανθεκτικούς σε φυσικές δονήσεις. Παρόλα αυτά, είναι πιο ακριβοί από τους HDDs και έχουν μικρότερο προσδόκιμο χρόνο ζωής, αφού κάθε μπλοκ υποστηρίζει ένα πεπερασμένο αριθμό εγγραφών/διαγραφών.

Πέρα από τον πρωτεύοντα χώρο αποθήκευσης, για τον συνδυασμό της μόνιμης αποθήκευσης και της φορητότητας των δεδομένων, υπάρχουν και τα αφαιρούμενα μέσα (removable media). Στην κατηγορία αυτή, υπάγονται τόσο διαφορετικές τεχνολογίες αποθήκευσης, όπως οι οπτικοί δίσκοι, όσο και προϋπάρχουσες στην αφαιρούμενη εκδοχή τους. Οι οπτικοί δίσκοι, περιλαμβάνουν τα CD, τα DVD και τα Blue-ray disks και χρησιμοποιούν ακτίνες λειζερ για να εγγράψουν τα δεδομένα σε αύλακες μιας κυκλικής επιφάνειας. Οι δίσκοι αυτοί βγαίνουν σε 2 εκδοχές, ανάλογα με το εάν επιτρέπουν την εγγραφή μία φορά ή πολλές. Επίσης άλλα αφαιρούμενα μέσα είναι οι μαγνητοοπτικοί δίσκοι, οι μαγνητικές ταινίες, οι απαρχαιωμένες δισκέτες (floppy disks), τα USB flash drives, οι SD κάρτες αλλά ακόμα και HDDs ή SSDs, που αυτήν την φορά δεν βρίσκονται ενσωματωμένοι σε κάποιο δίαυλο στην μητρική πλακέτα του υπολογιστή αλλά αποτελούν αφαιρούμενους δίσκους που κατά απαίτηση εισάγονται σε εξωτερικές θύρες του υπολογιστή.

Τέλος, στον πάτο της ιεραρχίας υπάρχει και ο δευτερεύων χώρος αποθήκευσης, ο οποίος στην πράξη απαρτίζεται από αφαιρούμενα μέσα αποθήκευσης τοποθετημένα σε βιβλιοθήκες και ένα ρομποτικό μηχανισμό, ο οποίος κατ᾽ απαίτηση εντοπίζει τα κατάλληλα αφαιρούμενα μέσα και τα εισάγει σε ένα οδηγό για εγγραφή ή ανάγνωση ή και για μεταφορά των δεδομένων σε κάποιον πρωτεύοντα χώρο αποθήκευσης. Χρησιμοποιείται κυρίως για αντίγραφα ασφαλείας (backups) και αρχειοθέτηση (archiving) τεράστιων όγκων σπάνια προσπελάσιμων δεδομένων, καθώς μειώνει το κόστος ανά gigabyte αλλά είναι έως και 1000 φορές πιο αργός από τον πρωτεύοντα χώρο απο-

θήκευσης. Τυπικά παραδείγματα αποτελούν οι βιβλιοθήκες μαγνητικών ταινιών (tape libraries) και τα οπτικά τζούκμποξ (optical jukeboxes).

Αναφέρουμε εδώ ότι υπάρχει μια σύγχυση στην ονοματοδοσία της ιεραρχίας αποθήκευσης, καθώς κάποιες προσεγγίσεις ταυτίζουν την πρωτεύουσα αποθήκευση με τις πτητικές μνήμες, και ορίζουν ως τριτεύουσα και όχι ως δευτερεύουσα αποθήκευση τα backup και τα archival συστήματα. Επειδή όμως το πλαίσιο εργασίας μας επικεντρώνεται στην μόνιμη μη-πτητική αποθήκευση, ακολουθούμε την σύμβαση των συναφών μελετών, κατά την οποία θεωρούμε ότι ο όρος πρωτεύουσα ή πρωτογενής αποθήκευση (primary storage) αναφέρεται στις μη-πτητικές μνήμες που απαιτούν γρήγορη προσπέλαση και ο όρος δευτερεύουσα ή δευτερογενής αποθήκευση (secondary storage) αναφέρεται στις μνήμες που προορίζονται για μακροπρόθεσμη αποθήκευση χωρίς απαιτήσεις ταχύτητας. Όπου δεν αναφέρεται το αντίθετο, στο παρόν κείμενο επικεντρωνόμαστε στον πρωτεύοντα χώρο αποθήκευσης και σε δίσκους HDD.

2.1.2 Συστήματα αρχείων

Όπως προαναφέραμε, οι δίσκοι έχουν ως στοιχειώδη δομική μονάδα αποθήκευσης τον τομέα (sector) και η διευθυνσιοδότησή τους γίνεται με το σχήμα logical block addresing (LBA), με κάθε τομέα να αντιπροσωπεύεται από ένα μοναδικό ακέραιο αριθμό, που κρύβει τα γεωμετρικά χαρακτηριστικά της τοποθεσίας του τομέα. Συχνά μπορεί οι δίσκοι να χωρίζονται εσωτερικά σε διαμερίσεις (partitions), κάθε μία από τις οποίες αποτελείται από ένα συνεχόμενο αριθμό τέτοιων τομέων. Οι διαμερίσεις αυτές υπάρχουν σε επίπεδο υλικού, με τους τομείς που απαρτίζουν κάθε διαμέριση να αποθηκεύονται σε ένα πίνακα, τον πίνακα διαμερίσεων (partition table), ο οποίος είναι παραδοσιακά τοποθετημένος στον πρώτο τομέα του φυσικού δίσκου που ονομάζεται και Master Boot Record (MBR)[3]. Ένας δίσκος ή μια διαμέρισή του, που προορίζεται για αποθήκευση δεδομένων με την μορφή αρχείων ή για την εγκατάσταση ενός λειτουργικού συστήματος, οργανώνει τα δεδομένα στους τομείς με την βοήθεια ενός συστήματος αρχείων.

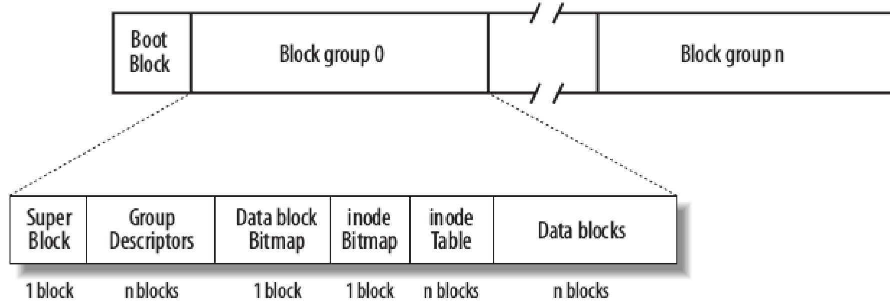
Ένα σύστημα αρχείων (filesystem) προσφέρει στους χρήστες την δυνατότητα να οργανώσουν τα δεδομένα τους μέσω εννοιών υψηλού επιπέδου, όπως τα αρχεία και οι

[3]διάδοχος αυτού του σχήματος είναι το πρότυπο GPT, που αποθηκεύει με διαφορετικό τρόπο τον πίνακα διαμερίσεων

κατάλογοι, αντί να τους υποχρεώνει να αλληλεπιδράσουν κατευθείαν με τους τομείς του δίσκου. Ένα αρχείο (file) είναι ένας συνεχής λογικός χώρος διευθύνσεων που αντιστοιχεί σε μια ακολουθία από bytes και ένας κατάλογος (directory) είναι μια συλλογή από αρχεία και άλλους καταλόγους. Έτσι το σύστημα αρχείων οργανώνει την αποθήκευση των δεδομένων, σε αρχεία τοποθετημένα σε ένα δέντρο καταλόγων. Τα αρχεία και οι κατάλογοι, πέρα από τα δεδομένα που περιέχουν, χαρακτηρίζονται και από κάποιες πληροφορίες όπως ο τύπος, τα δικαιώματα προσπέλασης, χρονοσφραγίδες, το μέγεθος κ.α. και οι οποίες αποτελούν τα μεταδεδομένα τους. Το σύστημα αρχείων μπορεί να διατηρεί και άλλου είδους μεταδεδομένα για δική του χρήση, όπως είναι γενικές πληροφορίες, πληροφορίες διαχείρισης του χώρου στον φυσικό δίσκο, αθροίσματα ελέγχου και οτιδήποτε άλλο μπορεί να του είναι χρήσιμο.

Ένα σύστημα αρχείων, για να αποθηκεύει και να διαχειρίζεται τα δεδομένα του, πρέπει να μπορεί να μεταχειρίζεται τον φυσικό χώρο αποθήκευσης. Για λόγους αφαίρεσης και απομόνωσης από το φυσικό μέσο, το σύστημα αρχείων θεωρεί ως στοιχειώδη μονάδα αποθήκευσης το μπλοκ (block), που στην περίπτωση των δίσκων αντιστοιχεί σε έναν αριθμό συνεχόμενων τομέων. Έτσι, βλέπει τον δίσκο ως ένα συνεχή λογικό χώρο διευθύνσεων που αντιστοιχεί σε μια ακολουθία από μπλοκ και στην πραγματικότητα μεταφράζεται με γραμμικό τρόπο σε μια ακολουθία από τομείς. Δεν αρκεί όμως να μπορεί να τοποθετεί τις πληροφορίες που θέλει σε μπλοκ, αλλά χρειάζεται να το κάνει με ένα συνεπή και οργανωμένο τρόπο. Έτσι οργανώνει τα δεδομένα και τα μεταδεδομένα ανάλογα με την σημασιολογία τους, σε συγκεκριμένες δομές που ονομάζονται δομές δεδομένων στο δίσκο (on-disk data structures). Οι δομές αυτές ορίζουν το πρότυπο με το οποίο αποθηκεύονται οι πληροφορίες σε κάθε μπλοκ και συνοδεύονται από μια συγκεκριμένη οργάνωση και χωροθέτηση τους στον δίσκο.

Ως παράδειγμα αναφοράς των δομών δεδομένων στο δίσκο, θα χρησιμοποιήσουμε την οικογένεια των συστημάτων αρχείων ext (ext2, ext3, ext4) [ed] [BC05], που χρησιμοποιείται σε πολλές διανομές Linux. Για να γίνουν κατανοητά τα παρακάτω, αναφέρουμε ότι αυτά τα συστήματα αρχείων αναθέτουν ένα μοναδικό αναγνωριστικό σε κάθε αρχείο, το οποίο είναι ένας ακέραιος αριθμός που ονομάζεται inode. Ένας αριθμός inode, συσχετίζεται με μια δομή inode, που περιγράφει χρήσιμα μεταδεδομένα για το εν λόγω αρχείο. Το ext4 χωρίζει τα διαθέσιμα μπλοκ σε n block groups μεγέθους 128MB, καθένα από τα οποία έχει τις εξής δομές:



- ένα αντίγραφο του superblock, που περιέχει γενικές πληροφορίες για όλο το σύστημα αρχείων, απαραίτητες για την προσάρτησή του από ένα λειτουργικό σύστημα
- n group descriptor blocks, με κάθε ένα μπλοκ να περιγράφει τις τοποθεσίες των επόμενων τριών δομών
- ένα data block bitmap, που περιγράφει ποια από τα μπλοκ δεδομένων σε αυτό το block group είναι δεσμευμένα και ποια διαθέσιμα
- ένα inode bitmap, που περιγράφει ποιοι αριθμοί inodes είναι δεσμευμένοι και ποιοι διαθέσιμοι
- ένα inode table, που περιέχει inode δομές για τους inode αριθμούς στους οποίους αναφέρεται ο εν λόγω πίνακας
- n data blocks, που αποθηκεύουν τα δεδομένα των αρχείων

Ένας κατάλογος αντιμετωπίζεται ως ένα ειδικό αρχείο και τα δεδομένα του, που δεν είναι τίποτα άλλο από τα ονόματα των αρχείων και των υποκαταλόγων του, αποθηκεύονται σε κανονικά μπλοκ δεδομένων (data blocks), που το σύστημα αρχείων προσπελάζει όταν χρειάζεται.

Πέρα από την οργάνωση των πληροφοριών σε δομές δεδομένων και την χωροθετήσή τους στον δίσκο, ένα σύστημα αρχείων πρέπει να ενσωματώνει και αλγορίθμους προσπέλασής και επεξεργασίας των παραπάνω δομών, ώστε να μπορεί να προσφέρει τις απαιτούμενες λειτουργίες υψηλότερου επιπέδου στα αρχεία και την δενδρική τους οργάνωση. Τέτοιες λειτουργίες, όπως η εγγραφή και η ανάγνωση bytes ενός αρχείου ή η δημιουργία και διαγραφή αρχείων και καταλόγων, ανάγονται εσωτερικά από το σύστημα αρχείων σε λειτουργίες επί των δομών του, όπως είναι η επίλυση ενός ονόματος αρχείου με βάση την ακολουθία των καταλόγων, η εύρεση του κατάλληλου

μπλοκ δεδομένων, η ενημέρωση των μεταδεδομένων ενός αρχείου και άλλες. Επιπροσθέτως, το σύστημα αρχείων πρέπει να διαχειρίζεται αποδοτικά τα μπλοκ του φυσικού αποθηκευτικού χώρου, οπότε πρέπει ενσωματώνει και ανάλογες συναρτήσεις για δέσμευση/αποδέσμευση μπλοκ, αποκερματισμό (defragmentation) των μπλοκ και άλλες.

Τέλος σημειώνουμε ότι για να εξυπηρετηθεί μια λειτουργία υψηλού επιπέδου, τα δεδομένα και τα μεταδεδομένα που εμπλέκονται σε αυτήν την λειτουργία, όπως αυτά εκφράζονται μέσω των on-data disk structures, πρέπει πρώτα να μεταφερθούν στην μνήμη, ώστε να είναι προσβάσιμα από τον κώδικα του συστήματος αρχείων. Γι' αυτό το λόγο οι δομές αυτές, αφού διαβαστούν από τον δίσκο, απεικονίζονται σε αντίστοιχες δομές στην μνήμη και διατηρούνται εκεί για όσο είναι εφικτό, ώστε μελλοντικά αιτήματα που επίσης τις χρειάζονται να αποφύγουν κοστοβόρες προσπελάσεις του δίσκου. Στο Linux, για αποθήκευση πληροφοριών σχετικές με τα περιεχόμενα καταλόγων υπάρχει η dentry cache, για αποθήκευση μεταδεδομένων αρχείων υπάρχει η inode cache και για αποθήκευση μπλοκ δεδομένων χρησιμοποιείται η page cache. Εφόσον όμως μελλοντικές αλλαγές στις δομές αυτές θα πραγματοποιηθούν στην μνήμη, ανακύπτει πρόβλημα ασυνέπειας με τις αντίστοιχες δομές του δίσκου, και έτσι αναπόφευκτα σε μια διακοπή τροφοδοσίας ο δίσκος θα έχει παρωχημένα δεδομένα. Για να αποφευχθεί αυτό, οι δομές στην δίσκο ανά διαστήματα «συγχρονίζονται», δηλαδή αποκτούν τα ίδια δεδομένα, με τις αντίστοιχες δομές στην μνήμη. Αυτό γίνεται είτε ανά συγκεκριμένα διαστήματα είτε με κάποια ρητή εντολή από την εφαρμογή που θέλει να είναι σίγουρη ότι τα δεδομένα της έχουν αποθηκευτεί με μόνιμο τρόπο. Στο Linux σε αυτήν την οικογένεια εντολών ανήκουν οι κλήσεις συστήματος `sync()`, `syncfs()`, `fsync()`, `fdatasync()`.

Μετάφραση μπλοκ αρχείου σε λογικό μπλοκ

Η μετάφραση από μπλοκ αρχείου (file block) σε λογικά μπλοκ (logical block) αποτελεί μια βασική εσωτερική λειτουργία ενός συστήματος αρχείων, η οποία πραγματοποιείται σε κάθε ανάγνωση και εγγραφή αρχείου. Παράλληλα η λειτουργία αυτή εισάγει και την ιδέα της μετάφρασης μεταξύ δυο διεπαφών μέσω μιας δομής αντιστοίχησης, ιδέα που θα συναντήσουμε επανειλημμένα ως βασική τεχνική για την υλοποίηση εικονικοποίησης.

Οι εφαρμογές χρήστη αντιμετωπίζουν ένα αρχείο ως μια γραμμική ακολουθία από

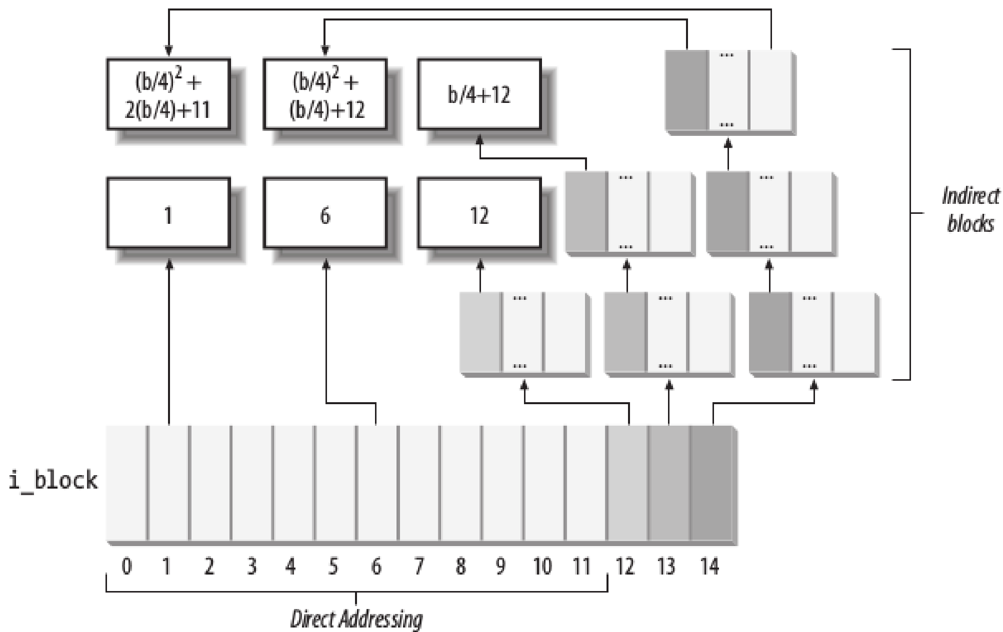
bytes τα οποία μπορούν να διαβάσουν και να γράψουν χρησιμοποιώντας κλήσεις συστήματος σε ένα συγκεκριμένο offset του αρχείου. Το σύστημα αρχείων χωρίζει αυτήν την ακολουθία με γραμμικό τρόπο, σε ομάδες από συγκεκριμένο αριθμό συνεχόμενων bytes, που ονομάζονται μπλοκ αρχείου (file block). Έτσι για παράδειγμα τα bytes 0-4095 αποτελούν το 1ο μπλοκ αρχείου, τα 4096-8191 το 2ο μπλοκ αρχείου και ούτω καθεξής. Επίσης, το σύστημα αρχείων αντιμετωπίζει μια συσκευή μπλοκ σαν ένα συνεχή λογικό χώρο διευθύνσεων που περιγράφουν μια ακολουθία από λογικά μπλοκ (logical blocks). Η ακολουθία των λογικών μπλοκ στην πραγματικότητα μεταφράζεται με γραμμικό τρόπο σε μια ακολουθία από τομείς του δίσκου. Τόσο τα μπλοκ αρχείου όσο και τα λογικά μπλοκ έχουν το ίδιο μέγεθος, το μέγεθος μπλοκ του συστήματος αρχείων. Συνεπώς, από την στιγμή που ο χρήστης επιδρά σε επίπεδο bytes και το σύστημα αρχείων σε επίπεδο λογικών μπλοκ, είναι απαραίτητη η ύπαρξη μιας δομής αντιστοίχησης η οποία θα μεταφράζει offsets του αρχείου σε λογικά μπλοκ.

Αυτή είναι μια διαδικασία δυο βημάτων, καθώς πρώτα ανακτάται ο αριθμός του μπλοκ αρχείου από το offset του αρχείου και στη συνέχεια από τον αριθμό του μπλοκ αρχείου εξάγεται ο αριθμός του λογικού μπλοκ. Είναι αρκετά εύκολο να βρεθεί ο αριθμός του μπλοκ αρχείου που περιέχει τον *ν-οστό* χαρακτήρα ενός αρχείου, απλά διαιρώντας το *ν* με το μέγεθος μπλοκ του συστήματος αρχείων και στρογγυλοποιώντας προς τον πλησιέστερο ακέραιο προς τα κάτω (ουσιαστικά το ακέραιο πηλίκο της διαίρεσης). Η μετάφραση από έναν αριθμό μπλοκ αρχείου σε έναν αριθμό λογικού μπλοκ δεν είναι τόσο απλή υπόθεση, αφού τα μπλοκ δεδομένων του αρχείου δεν είναι αναγκαστικά συνεχόμενα στο δίσκο. Επομένως, για να γίνει εφικτό το δεύτερο στάδιο μετάφρασης, το σύστημα αρχείων εισάγει μια δομή αντιστοίχισης, που αποθηκεύεται στα μεταδεδομένα του αρχείου.

Σαν παράδειγμα μιας τέτοιας δομής και διαδικασίας μετάφρασης θα πάρουμε και πάλι την οικογένεια των συστημάτων αρχείων ext. Στο ext2 και στο ext3 το inode ενός αρχείου περιέχει ένα πεδίο `i_block`, το οποίο είναι ένας πίνακας από δείκτες σε αριθμούς λογικών μπλοκ. Συνεπώς, ο αριθμός μπλοκ αρχείου (file block number) χρησιμοποιείται σαν δείκτης σε αυτόν τον πίνακα για να ανακτηθεί ο αριθμός λογικού μπλοκ (logical block number). Επειδή αυτός ο πίνακας θα αποκτούσε τεράστιο μέγεθος για να καλύψει περιπτώσεις μεγάλων αρχείων με πολλά μπλοκ αρχείου, χρησιμοποιείται μια δομή δενδρικής μορφής, που περιέχει μπλοκ έμμεσης δεικτοδότησης (indirect blocks), ώστε η αντιστοίχιση να γίνει πιο αποδοτική. Αυτά τα έμμεσα μπλοκ,

περιέχουν είτε δείκτες σε λογικά μπλοκ δεδομένων είτε δείκτες σε άλλα έμμεσα blocks. Η δομή αυτή παρουσιάζεται στο σχήμα 2.1.

Συγκεκριμένα, οι πρώτες 12 θέσεις του πίνακα περιέχουν τους αριθμούς λογικών μπλοκ, των μπλοκ δεδομένων που αντιστοιχούν στα πρώτα 12 μπλοκ αρχείου. Η 13η θέση περιέχει τον αριθμό λογικού μπλοκ ενός έμμεσου μπλοκ, το οποίο περιέχει ένα πίνακα δεύτερης τάξης με λογικούς αριθμούς μπλοκ δεδομένων. Αυτοί αντιστοιχούν στα μπλοκ αρχείου από 12 έως και $b/4 + 11$, όπου b είναι το μέγεθος μπλοκ του συστήματος αρχείων, ενώ η διαίρεση με το 4 πραγματοποιείται γιατί κάθε λογικό μπλοκ θεωρούμε ότι καταλαμβάνει 4 bytes. Επομένως, για αυτά τα μπλοκ αρχείου, ο κώδικας του συστήματος αρχείων πρέπει να συμβουλευτεί τον αρχικό πίνακα για να βρει ένα δείκτη στο έμμεσο μπλοκ, και στη συνέχεια πρέπει να συμβουλευτεί το έμμεσο μπλοκ για να βρει ένα δείκτη στο μπλοκ με τα πραγματικά δεδομένα. Κατά τον ίδιο τρόπο πραγματοποιείται διπλή και τριπλή έμμεση δεικτοδότηση, όπου απαιτείται οι διάσχιση τριών και τεσσάρων δεικτών αντίστοιχα, για να καταλήξουμε να ανακτήσουμε τον αριθμό λογικού μπλοκ που αντιστοιχεί στον αριθμό μπλοκ αρχείου. Η χρήση των 12 πρώτων θέσεων του πίνακα για άμεση δεικτοδότηση χρησιμοποιήθηκε για να ευνοήσει την ταχεία μετάφραση των πρώτων μπλοκ ενός αρχείου, μιας που τα περισσότερα αρχεία έχουν μικρό μέγεθος που καλύπτεται από αυτά τα 12 πρώτα μπλοκ.



Σχήμα 2.1: *Μετάφραση file block σε logical block στα ext2,ext3*

Το ext4 εισήγαγε την έννοια των extents που κάνουν την παραπάνω διαδικασία παρω-

χημένη. Ένα extent είναι απλώς ένα σύνολο από συνεχόμενα μπλοκ δεδομένων τόσο στον δίσκο όσο και στο αρχείο, δηλαδή αντιστοιχεί και σε συνεχόμενα μπλοκ αρχείου και και σε συνεχόμενα λογικά μπλοκ. Τα περισσότερα σύγχρονα συστήματα αρχείων, καταβάλλουν προσπάθειες για την δέσμευση συνεχόμενων λογικών μπλοκ και άρα συνεχόμενων τομέων του δίσκου για κάθε αρχείο, καθώς έτσι υπάρχει εκμετάλλευση της τοπικότητας, μείωση του κατακερματισμού και ταχύτερες λειτουργίες Ε/Ε. Ταυτόχρονα η χρήση extents μειώνει το μέγεθος μεταδεδομένων που απαιτούνται για την αντιστοίχιση, αφού ένα extent μπορεί να αντικαταστήσει ένα μεγάλο αριθμό από μπλοκ αρχείου. Αυτή η μείωση επιταχύνει ακόμα περισσότερο την προσπέλαση του αρχείου.

Στο ext4, η δενδρικής μορφής δομή αντιστοίχισης από αριθμό μπλοκ αρχείου σε αριθμό λογικού μπλοκ, αντικαταστάθηκε από ένα δέντρο δεικτοδότησης των extents (extent tree). Οι κόμβοι-φύλλα περιέχουν μια αντιστοίχιση ανάμεσα σε ένα εύρος συνεχόμενων μπλοκ αρχείου και ένα εύρος συνεχόμενων λογικών μπλοκ. Οι εσωτερικοί κόμβοι περιγράφουν το εύρος των μπλοκ αρχείου που καλύπτει το υποδέντρο τους, έτσι ώστε να επιτρέψουν την πλοήγηση μέχρι τους κατάλληλους τελικούς κόμβους-φύλλα. Ο κόμβος ρίζα του extent tree αποθηκεύεται στο πεδίο `i_block` και επιτρέπει την αποθήκευση των τεσσάρων πρώτων extents χωρίς την χρήση επιπλέον extent metadata blocks (κόμβων του δέντρου).

2.1.3 Στοίβα Ε/Ε

Σε αυτό το τμήμα, θα αναλύσουμε την διαδρομή των δεδομένων από μια εφαρμογή ενός υπολογιστικού συστήματος μέχρι την φυσική μη-πτητική μνήμη ή αντίστροφα. Για να γίνει αυτό εφικτό, χρειάζεται η μεσολάβηση του λειτουργικού συστήματος και συγκεκριμένα, διαφόρων αφαιρετικών στρωμάτων του, που παρουσιάζονται εκ των άνω προς τα κάτω. Χρησιμοποιούμε ως παράδειγμα μοντέλου στρωμάτων, αυτό της στοίβας Ε/Ε του Linux [BC05].

- Εφαρμογή
- VFS
- Σύστημα Αρχείων

- Κρυφή μνήμη σελίδων (Page Cache)
- Generic Block Layer
- Χρονοδρομολογητής Ε/Ε (I/O Scheduler)
- Οδηγός Συσκευής Μπλοκ (Block Device Driver)
- Σκληρός Δίσκος (HDD)

Εφαρμογή Στο υψηλότερο επίπεδο μια εφαρμογή, που τρέχει ως διεργασία στο λειτουργικό σύστημα, αλληλεπιδρά με ένα αρχείο, μέσω κλήσεων συστήματος του VFS όπως οι `open(), read(), write()`. Μια εφαρμογή χρήστη αντιλαμβάνεται ένα αρχείο ως μια γραμμική ακολουθία από bytes, έχοντας την δυνατότητα να προσπελάσει και να μεταβάλει κάθε ένα από αυτά.

VFS Το Virtual File System, αποτελεί ένα generic filesystem, δηλαδή ένα αφαιρετικό στρώμα μεταξύ των προγραμμάτων και των πραγματικών συστημάτων αρχείων, ενοποιώντας όλα τα συστήματα αρχείων κάτω από ένα κοινό πρότυπο και προφέροντας στο χρήστη μια ενιαία διεπαφή κλήσεων συστήματος. Για να το επιτύχει αυτό, υιοθετεί τις βασικές δομές και διεπαφές που διέπουν όλα τα συστήματα αρχείων, όπως για παράδειγμα αυτή των αρχείων και των λειτουργιών ανάγνωσης/εγγραφής σε αυτά, και καλεί συναρτήσεις των πραγματικών συστημάτων αρχείων για όσες λειτουργίες δεν είναι στοιχειώδεις. Οι βασικές δομές συγγενεύουν με αυτές των συστημάτων αρχείων ext που προαναφέραμε στην ενότητα 2.1.2. Καταφέρνει λοιπόν να επιτρέπει την υποστήριξη πολλών ετερογενών συστημάτων αρχείων για το Linux, και απεμπλέκει την εφαρμογή από τις λεπτομέρειες ενός συγκεκριμένου συστήματος αρχείων. Επίσης, δίνει την δυνατότητα για εικονικά συστήματα αρχείων που δεν βασίζονται σε δίσκους όπως το `/proc`, ενώ πραγματώνει και τη βασική αρχή του Unix «όλα είναι αρχεία», προσφέροντας κοινή διεπαφή για διαχείριση απλών αρχείων και ολόκληρων συσκευών.

Σύστημα Αρχείων Όπως αναφέρθηκε στην ενότητα 2.1.2, ένα σύστημα αρχείων οργανώνει τα δεδομένα και τα μεταδεδομένα στον δίσκο, προσφέροντας στις εφαρμογές τις υψηλού επιπέδου έννοιες των αρχείων και των καταλόγων. Κάθε συγκεκριμένο

σύστημα αρχείων συμβατό με τη στοίβα Ε/Ε του Linux, οφείλει να παρέχει συγκεκριμένες υλοποιήσεις για διάφορες συναρτήσεις της διεπαφής του VFS.

Page Cache Όπως αναφέρθηκε στην ενότητα 2.1.2, η page cache συνιστά μια write-back cache που αποτελείται από σελίδες μνήμης RAM (pages) τις οποίες το λειτουργικό χρησιμοποιεί για αποθήκευση μπλοκ δεδομένων του δίσκου. Οι σελίδες αυτές χρησιμοποιούνται για να επιταχύνουν τόσο μελλοντικές αναγνώσεις, όσο και εγγραφές. Για μια κλήση συστήματος `read()`, το σύστημα αρχείων τοποθετεί πρώτα σε σελίδες της page cache τα κατάλληλα μπλοκ και στην συνέχεια τα αντιγράφει από εκεί στον απομονωτή (buffer) του χώρου χρήστη. Για μια κλήση συστήματος `write()`, το σύστημα αρχείων αντιγράφει πρώτα σε σελίδες της page cache τα δεδομένα από το buffer του χώρου χρήστη και στη συνέχεια η κλήση συστήματος επιστρέφει ως ολοκληρωμένο. Μία σελίδα που περιέχει τα νέα δεδομένα μιας αίτησης εγγραφής θεωρείται βρώμικη (dirty), και κάποια στιγμή αργότερα θα συγχρονιστεί με τον δίσκο. Η αναβολή του συγχρονισμού εφαρμόζεται ώστε να εξοικονομηθούν εγγραφές στον δίσκο από τυχόν επιπλέον μεταβολές της ίδιας σελίδας στο κοντινό μέλλον. Η ποσότητα μνήμης που χρησιμοποιείται αφορά μνήμη που θα ήταν ειδάλλως ανενεργή και άρα η page cache δεν στερεί μνήμη από τις διεργασίες.
Μια εφαρμογή είναι δυνατόν να παρακάμψει την page cache, υποδεικνύοντας στην κλήση συστήματος `open()`, την σημαία `O_DIRECT`. Σύμφωνα με αυτήν, το σύστημα αρχείων επικοινωνεί κατευθείαν με το generic block layer, ενώ τα δεδομένα μεταφέρονται προς ή από τον δίσκο, κατευθείαν από το buffer του χώρου χρήστη, χωρίς πρώτα να αποθηκευτούν σε σελίδες της page cache (zero-copy). Αυτό μπορεί να είναι θεμιτό, γιατί η εφαρμογή εκτελεί η ίδια κάποιου είδους προσωρινή αποθήκευση και δεν θέλει να βασίζεται σε αυτή του πυρήνα.

Generic Block Layer Το επίπεδο αυτό, σε συνδυασμό με το επίπεδο του χρονοδρομολογητή Ε/Ε, προσφέρουν ένα ενιαίο τρόπο χειρισμού των λειτουργιών Ε/Ε για όλες τις μπλοκ συσκευές (block devices) του συστήματος, χωρίς να απαιτείται η γνώση συγκεκριμένων λεπτομερειών της κάθε μιας.
Η βασική δομή που διέπει αυτό το επίπεδο από την έκδοση του πυρήνα 2.6 και μετά είναι το `struct bio` (BIO), το οποίο περιγράφει μια τρέχουσα λειτουργία Ε/Ε. Κάθε τέτοια δομή περιέχει ως σημαντικότερα πεδία, ένα δείκτη στη συσκευή μπλοκ, έναν

αριθμό αρχικού τομέα της λειτουργίας Ε/Ε, το πλήθος των bytes που αφορούν την Ε/Ε, το εάν η λειτουργία αφορά ανάγνωση ή εγγραφή, ένα δείκτη σε μια συνάρτηση που πρέπει να κληθεί μετά την ολοκλήρωση της Ε/Ε και έναν πίνακα σε ένα ή περισσότερα τεμάχια (segments). Ένα segment συνιστά μια συνεχόμενη περιοχή κάποιας σελίδας μνήμης, και περιγράφεται από ένα `struct bio_vec`, που περιέχει τον δείκτη στο `struct page` αυτής της σελίδας, το offset σε αυτήν την σελίδα και το πλήθος bytes που καταλαμβάνει. Κατ' αυτόν τον τρόπο είναι δυνατόν μια λειτουργία Ε/Ε να πραγματοποιηθεί αναφερόμενη σε πολλές μη συνεχόμενες περιοχές της μνήμης, δίνοντας την δυνατότητα και για κλήσεις scatter-gather, δηλαδή μεταφορά σε διασκορπισμένους απομονωτές (buffers) του χώρου χρήστη. Η περιοχή του δίσκου όμως στην οποία αναφέρεται μια λειτουργία Ε/Ε περιορίζεται σε ένα αυστηρά συνεχόμενο αριθμό τομέων. Τα BIOs δημιουργούνται για κάθε λειτουργία Ε/Ε, τόσο για αυτές τύπου `O_DIRECT`, όσο και για αυτές που διέρχονται από την page cache. Επίσης, το επίπεδο αυτό είναι υπεύθυνο να ελέγξει εάν ο αριθμός του τομέα δεν υπερβαίνει τα επιτρεπτά όρια, να μετατρέψει τον αριθμό αυτό από σχετικό αριθμό με βάση την διαμέριση σε απόλυτο αριθμό τομέα του δίσκου, και να καλέσει την κατάλληλη συνάρτηση που με βάση το BIO θα δημιουργήσει μια αίτηση (request) στην ουρά της συσκευής (request queue).

Εδώ να αναφέρουμε ότι πάνω από το generic block layer υπάρχει προαιρετικά και ένα επιπλέον επίπεδο που μπορεί να υλοποιεί λογικές συσκευές μπλοκ, συνδυάζοντας πολλαπλούς φυσικούς δίσκους. Τα πιο αντιπροσωπευτικά παραδείγματα είναι ο Logical Volume Manager (LVM) και το software RAID (mdraid), τα οποία συνδυάζουν πολλούς φυσικούς δίσκους προβάλλοντας έναν ενιαίο όμως λογικό χώρο διευθύνσεων προς το σύστημα αρχείων. Έτσι υπάρχει η δυνατότητα για δυναμική διαχείριση του μεγέθους τον δίσκων, για μεγαλύτερη απόδοση με την κατανομή των δεδομένων μεταξύ διαφορετικών δίσκων και για υποστήριξη πλεονασμού για ανθεκτικότητα, με την εισαγωγή επιπλέον bits.

Χρονοδρομολογητής Ε/Ε Επειδή η προσπέλαση στον δίσκο είναι πολύ χρονοβόρα, το λειτουργικό δεν εξυπηρετεί αμέσως κάθε μία αίτηση Ε/Ε, αλλά την τοποθετεί σε μια ουρά της συσκευής με την ελπίδα ότι σύντομα θα έρθουν και άλλες αιτήσεις οι οποίες θα μπορέσουν να συγχωνευτούν σε μία αίτηση, από συνεχόμενους πάντα τομείς του δίσκου. Έτσι, οι συνεχόμενες προσπελάσεις που εκδίδονται με πολλαπλά BIO,

ουσιαστικά ενοποιούνται έξυπνα σε μια μόνο αίτηση προς τον δίσκο και άρα εξαλείφονται άσκοπες μετακινήσεις της κεφαλής του. Για αυτό και κάθε BIO εισάγεται σε ένα `struct request`, και συγκεκριμένα σε μια λίστα του, με την ελπίδα ότι και άλλα BIO θα προστεθούν αργότερα σε αυτήν τη λίστα. To `struct request` περιέχει και χρήσιμα στοιχεία για την κατάσταση της αίτησης. Ουσιαστικά είναι μια δομή που περιέχει αιτήσεις που προκύπτουν μετά από προσπάθεια βελτιστοποίησης και αντικατοπτρίζει τις αιτήσεις που τελικά θα αποσταλούν στον οδηγό συσκευής. Αντίθετα ένα BIO περιγράφει μια αίτηση όπως ακριβώς προέκυψε από τα ανώτερα στρώματα, όπως για παράδειγμα από την ανάγκη του συστήματος αρχείων να προσπελάσει ένα μπλοκ δεδομένων.

Κάθε συσκευή μπλοκ διαθέτει μια ουρά, τύπου `struct request_queue`, στην οποία εισάγονται τα `struct request` μετά την δημιουργία τους. Η ουρά αυτή περιέχει μια λίστα από αιτήσεις που αναμένουν να εξυπηρετηθούν, χρήσιμες πληροφορίες για την κατάσταση της συσκευής, αλλά και ένα δείκτη σε έναν δρομολογητή Ε/Ε. Ο δρομολογητής αυτός αρχικά αναλαμβάνει να ελέγξει εάν ένα νέο BIO που προορίζεται για αυτήν την συσκευή, μπορεί να προστεθεί σε κάποιο ήδη υπάρχον `struct request`. Στην περίπτωση που μπορεί, ανάλογα με το εάν το BIO τοποθετηθεί στο τέλος ή στην αρχή της ουράς του `struct request`, ελέγχει και εάν μπορεί να υπάρξει back-merge ή front-merge αντίστοιχα, δηλαδή συγχώνευση του `struct request` με κάποιο άλλο. Ο δρομολογητής αυτός είναι επίσης υπεύθυνος για την πολιτική με την οποία τα requests θα είναι ταξινομημένα στην ουρά αποστολής (dispatch queue), που είναι μια ουρά που περιέχει τις αιτήσεις της request queue, με την σειρά που αυτές θα εξυπηρετηθούν από τον οδηγό συσκευής. Στην πραγματικότητα, ο χρονοδρομολογητής Ε/Ε μπορεί να τοποθετεί τα requests σε περισσότερες ιδιωτικές ουρές, ώστε να υλοποιεί καλύτερα την επιθυμητή πολιτική.

Οι 4 βασικοί αλγόριθμοι χρονοδρομολογητών Ε/Ε είναι ο Noop elevator, ο CFQ, ο Deadline elevator και ο Anticipatory elevator. Γενικώς η πολιτική των αλγορίθμων, ακολουθεί το πρότυπο του ανελκυστήρα, όπου προτιμάται η κεφαλή του δίσκου να κινείται προς μια κατεύθυνση και όχι να μεταπηδά με τυχαίο τρόπο. Ο Noop (No operation) δεν ταξινομεί τις αιτήσεις, αλλά χρησιμοποιεί μια FIFO λογική σύμγωνα με την οποία τις τοποθετεί στο τέλος της ουράς και ο οδηγός συσκευής εξυπηρετεί αιτήσεις από την αρχή. Ο CFQ (Complete Fairness Queueing) προσπαθεί να εγγυηθεί δικαιοσύνη ανάμεσα στις διεργασίες, τοποθετώντας τις αιτήσεις σε 64 ουρές ανάλογα

με το hash του pid της διεργασίας που εξέδωσε την λειτουργία Ε/Ε. Η dispatch queue γεμίζει επιλέγοντας με τρόπο round-robin μια μη κενή ουρά και λαμβάνοντας τις n πρώτες αιτήσεις από αυτήν. Ο deadline χρησιμοποιεί ξεχωριστές ουρές για εγγραφές και αναγνώσεις, ταξινομημένες ανάλογα με τον αριθμό του τομέα κάθε αίτησης. Για να γεμίσει την ουρά αποστολής, προτιμά την ουρά των αναγνώσεων, εάν αυτή των εγγραφών δεν έχει απορριφθεί πολλές φορές, ενώ επίσης εφαρμόζει και μια πολιτική γήρανσης ώστε να μην υπάρχουν αιτήσεις που περιμένουν για πολλή ώρα. Ο Anticipatory, έχει και αυτός δυο ουρές, προτιμώντας αυτή των αναγνώσεων, ενώ πέρα από την πολιτική γήρανσης εφαρμόζει και ευριστικές τεχνικές βασισμένες σε στατιστικά.

Για να είναι δυνατή η συγχώνευση και η αναδιάταξη αιτήσεων από τον δρομολογητή Ε/Ε, ο οδηγός συσκευής δεν θα πρέπει να τις εξυπηρετεί αμέσως, αλλά θα πρέπει να παραμένει αδρανής για κάποιο διάστημα. Αυτό πραγματοποιείται με το plugging της συσκευής και συγκεκριμένα της αντίστοιχης ουράς, απενεργοποιώντας την κλήση του οδηγού ακόμα και αν υπάρχουν αιτήσεις που εκκρεμούν. Εάν περάσει ένα προκαθορισμένο χρονικό διάστημα ή μετά από ρητή απαίτηση του πυρήνα (π.χ. για εφαρμογή barriers), η συσκευή και η ουρά γίνονται unplugged, ο οδηγός συσκευής ενεργοποιείται και καλείται η συνάρτηση `request()` που αποτελεί την ρουτίνα στρατηγικής (strategy routine). Αυτή παίρνει την πρώτη αίτηση από την ουρά αποστολής για να την εξυπηρετήσει.

Επειδή μια διεργασία, δεν είναι δυνατόν να καταναλώνει χρόνο στη CPU περιμένοντας πότε τελικά θα εξυπηρετηθεί μια αίτηση, αφότου η αίτηση εισαχθεί στην ουρά της συσκευής, η διεργασία κοιμάται, περιμένοντας να την ξυπνήσει ο πυρήνας, όταν και αυτός ειδοποιηθεί για την ολοκλήρωση της αίτησης.

Πέρα από το παραπάνω μοντέλο δρομολόγησης, ένα BIO μπορεί εναλλακτικά είτε να κατευθυνθεί στο Linux multiqueue block I/O queuing mechanism (blk-mq), το οποίο εισήχθη πρόσφατα στο Linux Storage Stack για να υποβοηθήσει συσκευές flash υψηλής απόδοσης, είτε να παρακάμψει το επίπεδο του χρονοδρομολογητή Ε/Ε και να κατευθυνθεί κατευθείαν στον οδηγό της συσκευής μπλοκ που ενδέχεται να έχει την δικιά του ενσωματωμένη λογική δρομολόγησης.

Block Device Driver Ο οδηγός μιας μπλοκ συσκευής, όπως και κάθε άλλος οδηγός συσκευής, εξαρτάται από την συγκεκριμένη συσκευή και είναι υπεύθυνος ώστε

να μεταφράζει υψηλού επιπέδου λειτουργίες σε χαμηλού επιπέδου εντολές του πρωτοκόλλου της συσκευής. Συγκεκριμένα, οφείλει να κυρίως να εκδίδει τις αιτήσεις Ε/Ε των παραπάνω στρωμάτων προς τον δίσκο, κάτι που φέρνει εις πέρας χρησιμοποιώντας λειτουργίες DMA.

Το πρότυπο DMA παρέχει τη δυνατότητα σε υποσυστήματα του υλικού ενός υπολογιστή να έχουν πρόσβαση στην κύρια μνήμη για ανάγνωση ή εγγραφή δεδομένων, χωρίς να εμπλέκεται ο επεξεργαστής. Αυτό υλοποιείται με έναν εξειδικευμένο ελεγκτή, ο οποίος αφού λάβει τις παραμέτρους και την εντολή από τον επεξεργαστή, ξεκινάει την μεταφορά δεδομένων μεταξύ της μνήμης και της συσκευής εισόδου/εξόδου, ενώ την ίδια στιγμή ο επεξεργαστής ασχολείται με άλλες διεργασίες. Μόλις η μεταφορά ολοκληρωθεί, προκαλεί μια διακοπή ώστε να ειδοποιήσει τον επεξεργαστή για την ολοκλήρωσή της. Έτσι ο οδηγός συσκευής, καλεί την ρουτίνα στρατηγικής, λαμβάνει το επόμενο αίτημα Ε/Ε από την ουρά αποστολής, ξεκινάει μια διαδικασία DMA μεταξύ του ελεγκτή του δίσκου και των κατάλληλων θέσεων στην μνήμη και στη συνέχεια τερματίζει. Όταν η λειτουργία DMA ολοκληρωθεί, η διακοπή που προκαλείται οδηγεί στην κλήση και πάλι της ρουτίνας στρατηγικής του οδηγού για την εξυπηρέτηση του επόμενου αιτήματος.

Πέρα από την εξυπηρέτηση των αιτήσεων Ε/Ε με την έκδοση κατάλληλων εντολών χαμηλού επιπέδου, ο οδηγός αναλαμβάνει μετά από αίτημα των ανωτέρω στρωμάτων, να εκδίδει και διάφορες ειδικές εντολές χαμηλού επιπέδου συναφείς με την συσκευή ή την διεπαφή της, οι οποίες υλοποιούν εξειδικευμένες λειτουργίες.

Φυσικός δίσκος Στο χαμηλότερο επίπεδο, όπως κάθε άλλη συσκευή Ε/Ε, ο φυσικός δίσκος προσαρτάται σε κάποιο δίαυλο (bus) του συστήματος, μαζί με τον ελεγκτή του (disk controller). Ο επεξεργαστής επικοινωνεί με την συσκευή εισόδου-εξόδου, εκδίδοντας εντολές `IN` και `OUT` προς τις θύρες Ε/Ε (I/O ports), που είναι απλώς ειδικοί καταχωρητές του διαύλου, οι οποίοι ορίζουν και την διεύθυνση της συσκευής. Ανάμεσα στην θύρα Ε/Ε και τον ελεγκτή του δίσκου, μεσολαβεί ένα ειδικό κύκλωμα υλικού, που ονομάζεται διεπαφή Ε/Ε (I/O interface). Αυτό φροντίζει να ερμηνεύει τις τιμές των θυρών Ε/Ε και να τις μεταφράζει σε εντολές για τον ελεγκτή του δίσκου καθώς και να εντοπίζει τις αλλαγές στην κατάσταση του δίσκου και είτε να ενημερώνει τις θύρες Ε/Ε είτε να προκαλεί διακοπές εκ μέρους του. Οι πιο συνηθισμένες διεπαφές για δίσκους είναι οι PATA (IDE), Serial ATA και για υψηλού επιπέδου δίσκους τα

SCSI, Fibre Channel, Serial Attached SCSI (SAS). Ο ελεγκτής του δίσκου, που συνήθως είναι ενσωματωμένος στον ίδιο τον δίσκο, λαμβάνει τις εντολές της διεπαφής Ε/Ε, που συνήθως είναι του τύπου «εγγραφή ενός τομέα», και τις μετατρέπει σε ηλεκτρικά σήματα για το δίσκο, όπως το «τοποθέτηση της κεφαλής στην κατάλληλη τροχιά», «περιστροφή του δίσκου», «εγγραφή των δεδομένων». Αντίστοιχα για κάθε αλλαγή κατάστασης του δίσκου, μέσω της διεπαφής, ενημερώνει τις θύρες Ε/Ε ή προκαλεί μια διακοπή.
Πέρα από την μετάφραση των εντολών χαμηλού επιπέδου του οδηγού συσκευής, ο ελεγκτής μπορεί να ενσωματώνει και έξτρα λειτουργικότητα ώστε να εφαρμόζει την δικιά του αναδιάταξη των αιτημάτων για καλύτερη απόδοση. Επίσης, οι περισσότεροι σύγχρονοι δίσκοι διαθέτουν και μια κρυφή μνήμη (disk cache) ώστε να επιταχύνονται οι εγγραφές και οι μελλοντικές αναγνώσεις.

2.1.4 Εναλλακτικές αρχιτεκτονικές αποθήκευσης

Η παραδοσιακή χρήση μιας συσκευής αποθήκευσης που περιγράφηκε παραπάνω είναι αυτή του Direct Attached Storage (DAS), κατά την οποία η συσκευή συνδέεται κατευθείαν και με αποκλειστικό τρόπο σε κάποιο δίαυλο ενός υπολογιστή και είναι προσπελάσιμη μόνο από αυτόν. Η σύγχρονη τάση όμως ευνοεί τον δικτυωμένο αποθηκευτικό χώρο, όπου η παροχή είναι αποσυνδεδεμένη από την χρήση του αποθηκευτικού χώρου, κάτι που είναι επιθυμητό, καθώς οι πόροι είναι ευκολότερα διαμοιραζόμενοι, οι φυσικοί δίσκοι μπορούν να συγκεντρωθούν σε ένα χώρο για ευκολότερη διαχείριση και συντήρηση, ενώ καθίσταται εφικτή η λογική αφαίρεση των φυσικών δίσκων. Οι δύο βασικές κατηγορίες του storage networking είναι οι SAN και NAS.

Το μοντέλο Network Attached Storage (NAS) περιγράφει ένα δίκτυο αποθηκευτικών συσκευών που προσφέρουν προσπέλαση σε επίπεδο αρχείων. Οι συσκευές είναι δίσκοι προσαρτημένοι σε συγκεκριμένους υπολογιστές ή ειδικές συσκευές όπως και στο DAS πρότυπο, αλλά αυτή την φορά τα αρχεία τους μπορούν να είναι προσπελάσιμα από πολλούς ετερογενείς πελάτες μέσα από ένα τοπικό δίκτυο (LAN), ένα δίκτυο ευρείας περιοχής (WAN) ή και το Internet. Τα πιο συνηθισμένα πρωτόκολλα του μοντέλου NAS είναι το NFS και το CIFS που σχεδιάστηκαν για διαμοιρασμό αρχείων ανάμεσα σε δικτυακά συνδεδεμένους υπολογιστές. Το NFS είναι ένα πρωτόκολλο κατανεμημένου συστήματος αρχείων που προσφέρει στον χρήστη την δυνατότητα να προσαρτή-

σει ένα απομακρυσμένο σύστημα αρχείων και στην συνέχεια του προσφέρει την ψευδαίσθηση της προσπέλασης τοπικών αρχείων, ενώ στην πραγματικότητα προσπελάζει τα αρχεία ενός συστήματος αρχείων κάποιου NFS server που βρίσκεται στο δίκτυο. Δεν υπάρχει ειδική μέριμνα από το NFS ώστε ένα αρχείο να προστατεύεται από πολλαπλές εγγραφές διαφορετικών πελατών, αλλά υπάρχει η υποστήριξη για αυτό μέσω της χρήσης των μηχανισμών κλειδωμάτων αρχείων που προσφέρει το Linux.

Το μοντέλο Storage Area Network (SAN), περιγράφει ένα ειδικό δίκτυο αποθηκευτικών συσκευών που προσφέρουν προσπέλαση σε επίπεδο μπλοκ. Το SAN προσφέρει την ψευδαίσθηση τοπικά συνδεδεμένων συσκευών μπλοκ, αλλά σε αντίθεση με το NAS, η διαχείριση των δεδομένων και η τοποθέτηση ενός συστήματος αρχείων πάνω από αυτόν τον ακατέργαστο χώρο (raw storage) αφήνεται στην ευχέρεια του χρήστη. Ο όρος συνδέθηκε αρχικά με το Fibre Channel, το οποίο αποτελεί τόσο ένα δίκτυο υψηλής τεχνολογίας για διασύνδεση συσκευών αποθήκευσης μέσω οπτικών ινών, όσο και ένα πρωτόκολλο για την ενθυλάκωση SCSI εντολών και μεταφοράς τους προς τους δίσκους. Άλλες τεχνολογίες που αναπτύχθηκαν είναι οι iFCP και FCIP, οι οποίες ενθυλακώνουν πλαίσια Fibre Chanel σε πακέτα IP, ώστε να είναι δυνατή η σύνδεση δυο απομακρυσμένων δικτύων Fibre Chanel και η προβολή τους ως ένα ενιαίο σύστημα. Επίσης, το iSCSI ενθυλακώνει εντολές SCSI κατευθείαν σε πακέτα IP, προσφέροντας επικοινωνία σε SCSI μέσω δικτύων IP, ενώ πρωτόκολλα όπως το FCoE μπορούν να χρησιμοποιήσουν ένα δίκτυο Ethernet για την μεταφορά των Fiber Chanel. Συνεπώς, όλα τα πρωτόκολλα SAN σκοπό έχουν την μεταφορά εντολών SCSI από εξυπηρετητές σε δίσκους πάνω από κάποιο δίκτυο, που είτε είναι εξειδικευμένο, είτε χρησιμοποιεί πρωτόκολλα και υποδομές του Internet.

Τα δίκτυα SAN και NAS δίνουν την δυνατότητα εικονικοποίησης του storage, δηλαδή αφαίρεση των λογικών συσκευών από τις φυσικές. Αυτό, ειδικά σε μεγάλα κέντρα δεδομένων και περιβάλλοντα υπολογιστικού νέφους, παρέχει ευελιξία και αποδοτικότερη διαχείριση του διαθέσιμου χώρου, αφού κάθε ένας από τους υπολογιστές μιας μεγάλης ομάδας, μπορεί να δεσμεύει δυναμικά όσο χώρο χρειάζεται από μια πισίνα αποθηκευτικού χώρου ενός δικτύου συσκευών. Αντίθετα, σε προσεγγίσεις DAS, ο αναξιοποίητος χώρος ενός υπολογιστή δεν μπορεί να προσπελαστεί από άλλους και σπαταλιέται άσκοπα. Παράλληλα, η εικονικοποίηση μπορεί να παρέχει ανοχή σε σφάλματα δίσκων με την προσθήκη bit πλεονασμού ή/και αυξημένη απόδοση κατανέμοντας τα δεδομένα σε πολλούς δίσκους, αξιοποιώντας τεχνικές RAID. Αυτό πραγματοποιείται

με τον συνδυασμό άλλων δίσκων σε μια τοπική ή μη συστοιχία (disk array) και μπορεί να υλοποιηθεί και σε επίπεδο υλικού, χωρίς την μεσολάβηση λογισμικού του πελάτη, όπως το LVM και το raidmd που περιγράψαμε. Επίσης, αναδύονται δυνατότητες για αποδοτικό disaster recovery, αφού μέσω των SAN πρωτοκόλλων μπορούμε να έχουμε replication των δεδομένων σε ένα απομακρυσμένο δευτερεύον disk array, στην περίπτωση που το πρώτο καταστραφεί από φυσικά αίτια.

Μέχρι στιγμής είδαμε ότι το NAS και τα κατανεμημένα συστήματα αρχείων (distributed filesystems) προσφέρουν όχι μόνο προσπέλαση σε μη τοπικά αρχεία, αλλά και το διαμοιρασμό των αρχείων αυτών μεταξύ διαφορετικών υπολογιστών-πελατών. Στο μοντέλο SAN, οι δίσκοι μπορεί να είναι δικτυωμένοι προσφέροντας προσπέλαση επιπέδου μπλοκ, αλλά κάθε μπλοκ είναι συσχετισμένο και προσπελάσιμο από έναν μόνο υπολογιστή. Εάν θέλουμε τα μπλοκ κάποιων δίσκων να είναι μοιραζόμενα (shared disks) τότε οι υπολογιστές που διεκδικούν πρόσβαση σε αυτούς τους δίσκους, συνιστούν μια συστοιχία (cluster) και οφείλουν να χρησιμοποιήσουν ένα ενιαίο shared-disk filesystem ή αλλιώς clustered filesystem, πάνω από αυτούς τους δίσκους. Συγκεκριμένα αυτό το σύστημα αρχείων είναι υπεύθυνο να σειριοποιεί τις προσπελάσεις, έτσι ώστε να αποφεύγεται η αλλοίωση των δεδομένων (data corruption) κατά την ταυτόχρονη πρόσβαση των μπλοκ από διαφορετικούς κόμβους. Τέτοια συστήματα αρχείων είναι το GPFS, το OCFS2 και το GFS2.

Μια άλλη ενδιαφέρουσα αρχιτεκτονική αποθήκευσης είναι αυτή του object storage. Σύμφωνα με αυτήν τα δεδομένα αποθηκεύονται σε στοιχειώδεις μονάδες που αποκαλούνται αντικείμενα (objects). Τα αντικείμενα αυτά δεν είναι ιεραρχικά οργανωμένα όπως σε ένα σύστημα αρχείων, αλλά ισότιμα σε ένα επίπεδο σχήμα διευθυνσιοδότησης. Κάθε αντικείμενο περιέχει δεδομένα, μια μεταβλητή ποσότητα μεταδεδομένων που περιγράφουν τα δεδομένα και ένα καθολικά μοναδικό αναγνωριστικό. Αυτό το αναγνωριστικό χρησιμοποιείται από έναν χρήστη που θέλει να ανακαλέσει το αντικείμενο από την δεξαμενή των αντικειμένων, χωρίς να είναι γνωστή η φυσική διεύθυνση των δεδομένων, τα οποία μπορεί να είναι και διασκορπισμένα σε διαφορετικές συσκευές αποθήκευσης. Η προσέγγιση αυτή προσφέρει κλιμακωσιμότητα, απλότητα και εύκολη υποστήριξη κατανεμημένων περιβαλλόντων ενώ ο διαχωρισμός των μεταδεδομένων από τα δεδομένα και η ανεξαρτησία των αντικειμένων από τα χαρακτηριστικά του υποσυστήματος αποθήκευσης, μπορούν να προσφέρουν δυνατότητες για παραμετροποιήσιμη και αυτόματη διαχείριση της αποθήκευσής τους.

Μια εναλλακτική μέθοδος αποθήκευσης είναι αυτή του Content Adressable Storage (CAS) κατά την οποία τα δεδομένα ανακτώνται με βάση το περιεχόμενό τους και συγκεκριμένα με βάση κάποια τιμή κατακερματισμού (hash) που υπολογίζεται πάνω στα δεδομένα τους, και όχι με βάση την θέση τους στην δενδρική δομή ενός συστήματος αρχείων. Προφανώς τόσο στο CAS όσο και στο object storage απαιτείται ένα επίπεδο μετάφρασης για να ανακτηθούν οι τελικές διευθύνσεις τομέων του φυσικού δίσκου, όπου έχουν αποθηκευτεί τα δεδομένα. Η μέθοδος αποθήκευσης CAS χρησιμοποιείται για την εφαρμογή του απαλοιφής διπλοτύπων (deduplication), όπως θα περιγράψουμε στην ενότητα 3.1.3.

Μία τελευταία έννοια που αξίζει να παρουσιάσουμε είναι αυτή των αμετάβλητων δεδομένων (immutable). Σύμφωνα με αυτήν, τα δεδομένα άπαξ και θεωρηθούν immutable, τότε δεν μπορούν να υποστούν περαιτέρω εγγραφές και άρα να αλλάξουν στο μέλλον. Με το σημερινό χαμηλό κόστος των δίσκων είναι εφικτή η αποθήκευση του μεγάλου όγκου των imutable δεδομένων που συσσωρεύεται, έτσι ώστε να εκμεταλλευτούμε τις δυνατότητες προστασίας, διατήρησης ιστορικού και επίλυσης προβλημάτων ταυτοχρονισμού, που αυτή η τεχνολογία προσφέρει. Εφεξής χρησιμοποιούμε τον όρο immutable για να τονίσουμε την μονιμότητα της μη-μεταβλητότητας αυτών των δεδομένων, καθώς η ελληνική απόδοση των «αμετάβλητων δεδομένων» μπορεί να παρερμηνευθεί ως απουσία αλλαγής στα δεδομένα για ένα παρελθοντικό χρονικό διάστημα, χωρίς όμως να αποκλείεται η πιθανότητα μεταβολής τους στο μέλλον.

2.2 Εικονοποίηση και εικονικοί δίσκοι

2.2.1 Εικονικοποίηση

Με τον όρο εικονικοποίηση (virtualization), αναφερόμαστε συνήθως σε μεθόδους και τεχνικές για τη δημιουργία «εικονικών» αντικειμένων, αντίστοιχων των «φυσικών», τα οποία μπορούν να χρησιμοποιηθούν με ισοδύναμο τρόπο. Μια τέτοια εφαρμογή είναι η εικονικοποίηση του υλικού με στόχο την κατασκευή εικονικών μηχανών (virtual machines). Αυτό καθίσταται δυνατό με την προσομοίωση του υλικού (hardware) σε λογισμικό (software). Μέσα στην εικονική μηχανή που προκύπτει, ο χρήστης μπορεί να τρέχει το λειτουργικό σύστημα και εφαρμογές με τον ίδιο ακριβώς τρόπο όπως

και στο πραγματικό μηχάνημα. Ανάμεσα στα πλεονεκτήματα των εικονικών μηχανών έχουμε:

- ευελιξία με την ταυτόχρονη εκτέλεση πολλών διαφορετικών λειτουργικών συστημάτων στο ίδιο φυσικό σύστημα
- ασφάλεια, αφού κάθε εικονική μηχανή εκτελείται απομονωμένα από όλες τις υπόλοιπες και από το φυσικό υλικό
- φορητότητα, καθώς η εικονική μηχανή μπορεί να προσφέρει ένα μια αρχιτεκτονική συνόλου εντολών διαφορετική από αυτή του πραγματικού μηχανήματος και έτσι προγράμματα εκτελούνται σε διαφορετικό υλικό από αυτό για το οποίο έχουν γραφτεί/μεταγλωττιστεί
- αποδοτικότερη χρήση των φυσικών πόρων αφού μεγάλο πλήθος εικονικών μηχανών μπορεί να μοιράζεται περιορισμένους φυσικούς πόρους
- βελτιωμένη αξιοπιστία, αφού εικονική μηχανή μπορεί να μεταφερθεί σε διαφορετικό φυσικό σύστημα, ώστε να συνεχίσει τη λειτουργία χωρίς διακοπή, ξεπερνώντας προβλήματα του υλικού (π.χ. μια προγραμματισμένη εργασία συντήρησης)
- ολόκληρη η κατάσταση ενός μηχανήματος μπορεί να σωθεί και να μεταφερθεί ως απλό αρχείο
- επιτρέπει ανάπτυξη και έλεγχο λογισμικού σε απομονωμένα και και ετερογενή περιβάλλοντα

Στα αρνητικά συγκαταλέγονται η μειωμένη απόδοση, αφού δεν έχουμε άμεση αλληλεπίδραση με γρήγορο υλικό, ενώ η ύπαρξη πολλών εικονικών μηχανημάτων στο ίδιο φυσικό σύστημα μπορεί να οδηγήσει σε ασταθή απόδοση, αφού το φορτίο του ενός μηχανήματος πιθανόν να επηρεάζει την λειτουργία των υπολοίπων.

Το στρώμα λογισμικού που υλοποιεί το εικονικό υλικό πάνω από το φυσικό υλικό του host ονομάζεται hypervisor ή Virtual Machine Monitor, το φυσικό μηχάνημα πάνω στο οποίο τρέχει ο hypervisor ονομάζεται host και κάθε εικονική μηχανή ονομάζεται guest. Υπάρχουν οι Type-1 hypervisors που εγκαθίστανται κατευθείαν πάνω από το πραγματικό υλικο του host ώστε να υπάρχει γρήγορη αλληλεπίδραση με αυτό. Από

την άλλη, οι Type-2 hypervisors είναι λογισμικό διαρθρωμένο πάνω από το λειτουργικό σύστημα του host μηχανήματος και επικοινωνούν με αυτό για να έχουν πρόσβαση στο πραγματικό υλικό.

Η εικονικοποίηση μπορεί να διαιρεθεί σε πλήρη εικονικοποίηση (full virtualization) και παραεικονικοποίηση (paravirtualization). Κατά την πλήρη εικονικοποίηση το λειτουργικό σύστημα μπορεί να τρέξει μέσα στην εικονική μηχανή χωρίς καμία τροποποίηση, ακριβώς όπως τρέχει και στο πραγματικό μηχάνημα. Η εικονική μηχανή δεν ξέρει ότι τρέχει σε εικονικό περιβάλλον και ότι το υλικό της δεν είναι πραγματικό αλλά προσομοιώνεται μέσω κατάλληλου λογισμικού. Το κυριότερο πλεονέκτημα είναι η ευκολία στη δημιουργία εικονικών μηχανών, αλλά το γεγονός ότι υπάρχει αυξημένη ανάγκη για προσομοίωση υλικού μέσω λογισμικού καθιστά αυτήν την τεχνική αργή. Παράλληλα, άλλες τεχνικές όπως το page table shadowing στο λογισμικό είναι απαραίτητες ώστε να προστατεύουν σημαντικές δομές δεδομένων του λειτουργικού, όπως τα page table entries, από την επέμβαση του guest OS.

Αντίθετα, στην παραεικονικοποίηση, η εικονική μηχανή έχει επίγνωση ότι τρέχει σε εικονικό περιβάλλον που δεν υποστηρίζεται από πραγματικό υλικό και συνεργάζεται με τον host ώστε να επιτευχθεί καλύτερη απόδοση. Για το λόγο αυτό δεν μπορεί να χρησιμοποιηθεί ο ίδιος πυρήνας του λειτουργικού συστήματος με αυτόν που θα χρησιμοποιούνταν σε ένα πραγματικό μηχάνημα. Χρειάζεται να γίνουν κάποιες τροποποιήσεις ώστε ένα λειτουργικό σύστημα να τρέξει σε μία εικονική μηχανή που χρησιμοποιεί παραεικονικοποίηση. Ωστόσο η επίδοση είναι αρκετά καλύτερη σε σχέση με την πλήρη εικονικοποίηση, αφού λειτουργίες που είναι σημαντικά πιο χρονοβόρες σε ένα εικονικό περιβάλλον, δεν προσομοιώνονται αλλά εκτελούνται κατευθείαν από τον host μετά από την παρέμβαση του hypervisor. Τέτοιες κλήσεις ονομάζονται και hypercalls. Με αυτήν την τεχνική απλοποιείται επίσης και το λογισμικό του hypervisor.

Στην εικονικοποίηση, το λογισμικό δεν χρειάζεται να μεταφράζει κάθε εντολή, αλλά χρειάζεται να παρεμβαίνει για να συντηρήσει την ψευδαίσθηση μόνο όταν η εικονική μηχανή χρειάζεται να εκτελέσει μια προνομιούχο εντολή. Όσο οι διεργασίες στον guest εκτελούν μη προνομιούχες εντολές, ο hypervisor παραμένει αδρανής και οι εντολές εκτελούνται κατευθείαν στον φυσικό επεξεργαστή. Εντολές με προνομιούχα δικαιώματα όπως εντολές Ε/Ε και εξαιρέσεις, προκαλούν εξαιρέσεις στον host, ο οποίος τις χειρίζεται καλώντας τον hypervisor για να εξυπηρετήσει τα αιτήματα του guest.

Αυτός είτε τις διεκπεραιώνει σε λογισμικό είτε χρησιμοποιεί πραγματικές συσκευές του host.

Η αυξημένη ζήτηση για αποδοτική εκτέλεση εικονικών μηχανών, οδήγησαν στην ανάπτυξη ειδικών επεκτάσεων υλικού (π.χ. ειδικές εντολές στο σύνολο εντολών των σύγχρονων επεξεργαστών), έτσι ώστε το υλικό να υποστηρίζει αποδοτικότερα λειτουργίες εικονικοποίησης. Αρχίζοντας από το 2006, οι νεότεροι επεξεργαστές x86, τόσο από την AMD όσο και από την Intel, διαθέτουν τέτοιες επεκτάσεις, επιτρέποντας την εκτέλεση εικονικών μηχανών με απόδοση που προσεγγίζει αυτή του πραγματικού μηχανήματος. Προφανώς απαιτείται η εικονική μηχανή να εκτελείται σε φυσικό σύστημα με το ίδιο σύνολο εντολών.

Το παραπάνω έδωσε την δυνατότητα για ένα πλήρως εικονικοποιήσιμο σύστημα, όπου με χρήση επεκτάσεων υλικού για επιτάχυνση χρονοβόρων και δύσκολα εικονικοποιήσιμων λειτουργιών, το guest λειτουργικό σύστημα δεν χρειάζεται να μεταβληθεί και η απόδοση είναι αρκετά υψηλή. Από την στιγμή που σχεδόν όλοι οι πυρήνες λειτουργικών συστημάτων έχουν τρόπους για φόρτωση οδηγών συσκευών από τρίτους (third-party drivers), είναι ένα σχετικά προφανές βήμα η κατασκευή οδηγών που θα υποστηρίζουν διεπαφές παραεικονικοποίησης. Αυτό το μικρό βήμα από την πλήρη εικονικοποίηση στην παραεικονικοποίηση, δίνει το έναυσμα για την σύλληψη ενός φάσματος εικονικοποίησης. Σήμερα, τα κομμάτια που μπορούν να είναι πλήρως ή μερικώς εικονικοποιήσιμα είναι: δίσκοι και συσκευές δικτύου, διακοπές και χρονόμετρα, μέρη της πλατφόρμας όπως η μητρική πλακέτα, το BIOS και το boot firmware και τελικά οι προνομιούχες εντολές και τα page tables της μνήμης. Ανάλογα με την επιλογή πλήρους εικονικοποίησης ή παραεικονικοποίησης για καθένα από αυτά τα κομμάτια, έχουμε ένα ξεχωριστό σημείο στο φάσμα της εικονικοποίησης. Είναι λοιπόν ασφαλές να καταλήξουμε στο ότι ο όρος εικονικοποίηση αναφέρεται σε υποβοηθούμενη από το υλικό εικονικοποίηση, επιβάλλοντας την ιδέα της παραεικονικοποίησης για κάποια κομμάτια του guest λειτουργικού συστήματος.

Το μεγαλύτερο μέρος των πλατφορμών εικονικοποίησης τρέχουν πάνω από x86 αρχιτεκτονική. Παραδείγματα λογισμικού virtualization αποτελούν τα VMWare ESXi, Microsoft Hyper-V, Oracle VM VirtualBox, Xen, Parallels, και QEMU. Για τις ανάγκες μας, επιλέξαμε να χρησιμοποιήσουμε το QEMU-KVM, δηλαδή το λογισμικό προσομοίωσης του QEMU σε συνδυασμό με το υποσύστημα KVM του πυρήνα του Linux.

Το KVM (Kernel-based Virtual Machine) είναι ένα ανοικτού κώδικα (open source) υποσύστημα εικονικοποίησης που έχει ενσωματωθεί στον πηγαίο κώδικα του πυρήνα του Linux από την έκδοση 2.6.20, το οποίο προσφέρει έναν ενιαίο τρόπο για χρήση των επεκτάσεων εικονικοποίησης του επεξεργαστή. Αποτελείται από δύο modules, το `kvm.ko` και το `kvm_intel.ko` ή το `kvm_amd.ko` ανάλογα με το μοντέλο του επεξεργαστή του φυσικού μηχανήματος. Με την εισαγωγή ενός kernel module μπορεί να υπάρξουν πλήρως απομονωμένες εικονικές μηχανές που τρέχουν αμετάβλητες εικόνες λειτουργικών συστημάτων στον guest, μέσα σε ένα Linux host μηχάνημα.

Το QEMU (Quick EMUlator) είναι μία ανοικτού κώδικα πλατφόρμα για τη δημιουργία και διαχείριση εικονικών μηχανών. Το QEMU εκτελείται σε επίπεδο χρήστη και υποστηρίζει εικονικοποίηση είτε αμιγώς μέσω λογισμικού, είτε εικονικοποίηση μέσω υλικού. Στην πρώτη περίπτωση μπορεί να λειτουργήσει ως πλήρης προσομοιωτής χωρίς να απαιτεί ειδική υποστήριξη από το υλικό αλλά είναι πολύ αργό, ειδικά αν το σύνολο εντολών της εικονικής μηχανής διαφέρει από το σύνολο εντολών του φυσικού μηχανήματος. Στη δεύτερη περίπτωση, χρησιμοποιεί το KVM ώστε να έχει πρόσβαση στις επεκτάσεις εικονικοποίησης του επεξεργαστή, σε ένα ειδικό mode «KVM Hosting» . Τότε, η εικονική μηχανή είναι απαραίτητο να είναι της ίδιας αρχιτεκτονικής με το μηχάνημα στο οποίο εκτελείται. Τα λειτουργικά συστήματα του guest δεν χρειάζονται μεταβολή, ενώ ο QEMU εμπλέκεται στην προσομοίωση την συσκευών υλικού όπως δίσκοι, δίαυλοι, κάρτες γραφικών, δικτύου κτλ.

2.2.2 Υπολογιστικά περιβάλλοντα νέφους

Η εικονικοποίηση χρησιμοποιείται κατά κόρον στα υπολογιστικά περιβάλλοντα νέφους (cloud computing environments). Το cloud computing είναι ένα μοντέλο που επιτρέπει την βολική, κατά απαίτηση, δικτυακή πρόσβαση σε μια δεξαμενή από διαμοιραζόμενους υπολογιστικούς πόρους, όπως εξυπηρετητές δικτύου και συσκευές αποθήκευσης. Οι πόροι αυτοί μπορούν να δεσμεύονται και να απελευθερώνονται δυναμικά ανάλογα με τις απαιτήσεις, ταχύτητα και χωρίς ιδιαίτερη διαχειριστική προσπάθεια και ανθρώπινη αλληλεπίδραση. Έτσι μεγιστοποιείται η χρήση της υπολογιστικής ισχύος και μειώνεται το συνολικός κόστος για την διαχείριση των πόρων, καθώς χρησιμοποιείται λιγότερη ισχύς, ψύξη, χώρος σε rack κτλ για την συντήρηση του συστήματος.

Το cloud computing δεν είναι ταυτόσημο με την εικονικοποίηση αλλά είναι μια τεχνολογία που χρησιμοποιεί την εικονικοποίηση για να πετύχει ευκολότερα τους στόχους της. Η εικονικοποίηση επιβάλει μια λογική αφαίρεση και μια απομόνωση των πόρων υλικού από την παρεχόμενη αποθήκευση, δικτυακή συνδεσιμότητα κτλ. Το cloud αποφασίζει πως αυτοί οι εικονικοποιημένοι πόροι θα δεσμευτούν, θα διανεμηθούν και θα παρουσιαστούν. Η εικονικοποίηση δεν είναι απαραίτητη για την δημιουργία ενός περιβάλλοντος υπολογιστικού νέφους αλλά διευκολύνει την γρήγορη κλιμακωσιμότητα των εμπλεκόμενων πόρων και παράλληλα παρέχει μεγάλη ευελιξία και δυνατότητες για αυξημένη χρησιμοποίησή τους. Μπορεί να υποστηρίξει τον διαμοιρασμό πόρων, την απομόνωση των εικονικών μηχανών και την δίκαιη κατανομή φορτίου (load balancing) και για αυτό χρησιμοποιείται ευρέως σε περιβάλλοντα υπολογιστικού νέφους.

2.2.3 Εικονικοί Δίσκοι

Προείπαμε ότι οι εικονικές μηχανές προσομοιώνουν σε λογισμικό διάφορες συσκευές υλικού. Από το υλικό αυτό δεν θα μπορούσαν να λείπουν συσκευές μόνιμης αποθήκευσης δεδομένων, όπως οι δίσκοι, όπου η εικονική μηχανή θα μπορεί να αποθηκεύει το λειτουργικό σύστημα του guest αλλά και επιπλέον δεδομένα των χρηστών. Αυτό καθίσταται δυνατό μέσω εικονικών δίσκων (virtual disks) που προσομοιώνουν ένα φυσικό δίσκο για έναν guest, με τον τρόπο προσομοίωσης να ποικίλει. Στο πιο απλό σενάριο οι εικονικοί δίσκοι, είναι αρχεία στο σύστημα αρχείων του host που περιέχουν τα δεδομένα που θα είχε και ένας φυσικός δίσκος που θα χρησιμοποιούσε το guest λειτουργικό σύστημα, και προαιρετικά κάποια επιπλέον μεταδεδομένα. Ο hypervisor αναλαμβάνει να συντηρήσει στο guest λειτουργικό σύστημα την ψευδαίσθηση ότι αυτό μεταχειρίζεται έναν κανονικό φυσικό δίσκο. Για να το πετύχει αυτό, παρεμβάλλεται στις αιτήσεις Ε/Ε που εκτελεί ο guest, και αναλαμβάνει βρει τα κατάλληλα δεδομένα από το αρχείο του εικονικού δίσκου και να τα τοποθετήσει στην μνήμη του guest, προσομοιώνοντας ουσιαστικά μια λειτουργία DMA. Προφανώς σε κάθε φυσικό δίσκο που βλέπει ο guest αντιστοιχεί ένας εικονικός δίσκος, δηλαδή στην απλή περίπτωση ένα αρχείο στον host. Συνεπώς κάθε αίτηση Ε/Ε στο δίσκο του guest, μεταφράζεται διαφανώς σε μια λειτουργία επί του αρχείου εικονικού δίσκου στον host.

Οι φυσικοί δίσκοι δεικτοδοτούν, προσπελάζουν και μεταφέρουν τα δεδομένα που έχουν

αποθηκευμένα σε επίπεδο τομέων, ενώ τα αρχεία ενός συστήματος αρχείων δεικτοδοτούν και διαχειρίζονται τα δεδομένα τους σε επίπεδο bytes. Η βασική λειτουργία ενός εικονικού δίσκου είναι να καταφέρνει να παρέχει διεπαφή σε επίπεδο τομέων για την ικανοποίηση των αιτημάτων του guest, μέσα από την διευθυνσιοδοτούμενη κατά byte δομή του αρχείου. Αυτό το καταφέρνει οργανώνοντας τα bytes του σε ομάδες που ισοδυναμούν με το μέγεθος ενός τομέα και αντιστοιχίζοντας κάθε αριθμό τομέα σε μια τέτοια ομάδα bytes. Έτσι, για να βρεθούν τα περιεχόμενα ενός τομέα, αρκεί να βρεθεί το κατάλληλο offset στο αρχείο εικονικού δίσκου. Το offset αυτό προφανώς θα «δείχνει» στο πρώτο byte της ομάδας bytes που αντικατοπτρίζει τον αντίστοιχο τομέα. Για να καταστεί δυνατή αυτή η αντιστοίχιση, αναδύεται η ανάγκη για ένα στρώμα αντιστοίχισης (mapping layer) που θα μεταφράζει αριθμούς τομέων σε offsets ενός αρχείου.

Το στρώμα αντιστοίχισης στην απλή του περίπτωση δεν είναι τίποτα άλλο παρά μια απλή γραμμική αντιστοίχιση. Αν θεωρήσουμε ως μέγεθος τομέα τα 512 bytes, ο τομέας x απλώς αντιστοιχίζεται στο offset $512 * x$ του αρχείου εικονικού δίσκου. Γι' αυτήν τη μετάφραση δεν χρειάζεται να κρατήσουμε καν κάποια δομή αντιστοίχισης, αφού η αντιστοίχιση μπορεί να γίνει αμιγώς υπολογιστικά σε πραγματικό χρόνο. Πράγματι αυτή είναι και η προσέγγιση που εφαρμόζει το raw block format στον QEMU [4].

Σε μια πιο πλούσια προσέγγιση, στο αρχείο μπορεί να διατηρείται μια δομή αντιστοίχισης (π.χ. πίνακας, B-tree, hash buckets κτλ) που θα αντιστοιχίζει αριθμούς τομέα σε offsets του αρχείου εικονικού δίσκου. Αυτό συνήθως εφαρμόζεται όταν θέλουμε ο εικονικός δίσκος να προσφέρει επιμέρους δυνατότητες και χαρακτηριστικά, όπως στιγμιότυπα.

Ας δούμε λοιπόν την πλήρη διαδικασία που λαμβάνει χώρα, όταν ο guest εκτελεί μια αίτηση ανάγνωσης/εγγραφής. Όπως είδαμε πριν, το τελικό βήμα του guest είναι ο οδηγός συσκευής μπλοκ του λειτουργικού να στείλει στον ελεγκτή του δίσκου τις κατάλληλες εντολές, ανάλογα με την διεπαφή (interface), ώστε ο ελεγκτής να εκκινήσει την DMA διαδικασία αντιγραφής των κατάλληλων δεδομένων. Η αποστολή των εντολών αυτών γίνεται με προνομιούχες εντολές (π.χ. `IN`, `OUT` στο x86), οι οποίες επειδή προέρχονται από μια διεργασία χώρου χρήστη, και άρα δεν συμβαίνουν σε kernel mode, προκαλούν εξαίρεση στον host. Ο πυρήνας του host, ειδοποιεί τότε τον hypervisor,

[4] στην πραγματικότητα έχουμε $h + 512 * x$, αφού στο αρχείο προηγείται και μια επικεφαλίδα μεγέθους h

ο οποίος με κατάλληλες συναρτήσεις λαμβάνει τα δεδομένα της εντολής που ο guest ήθελε να στείλει στον ελεγκτή, ελεγκτής που είναι φυσικά εικονικός. Υλοποιώντας κατάλληλα στρώματα λογισμικού προσπαθεί να προσομοιώσει τη σημασιολογία της εντολής, αλληλεπιδρώντας αν χρειάζεται με το αρχείο εικονικού δίσκου. Έτσι μια αίτηση για ανάγνωση/εγγραφή κάποιων τομέων, αρχικά περνάει από το στρώμα της κατάλληλης διεπαφής το οποίο αποκωδικοποιεί την εντολή συμβουλευόμενο την σημασιολογία της διεπαφής και καταλήγει να κρατάει τα εξής: ένα αναγνωριστικό για την εικονική συσκευή στην οποία ο guest αναφέρεται, μια θέση μνήμης, έναν αριθμό τομέα, και το πλήθος των τομέων που οφείλει να γράψει αυτή η αίτηση. Στη συνέχεια, αφού διασχίσει άλλα πιθανώς generic στρώματα που ενοποιούν λειτουργίες στα διάφορα μοντέλα εικονικών συσκευών καλεί έναν οδηγό μπλοκ του εικονικού δίσκου (block driver). Προσοχή, αυτός ο οδηγός μπλοκ είναι διαφορετικός από τον οδηγό συσκευής μπλοκ που βρίσκεται στον πυρήνα ενός λειτουργικού συστήματος. Ο οδηγός μπλοκ του εικονικού δίσκου που είναι ενσωματωμένος στον κώδικα του hypervisor, υλοποιεί την αντιστοίχιση μεταξύ τομέα δίσκου και offset στο αρχείο εικονικού δίσκου. Για κάθε διαφορετική μορφή εικονικού δίσκου, υπάρχει και διαφορετικός οδηγός μπλοκ που ενσωματώνει την ανάλογη λογική. Αυτός λοιπόν, ανάλογα με τον αριθμό τομέα και το πλήθος των τομέων, επιτελεί την μετάφραση συμβουλευόμενος, αν χρειάζεται, την δομή αντιστοίχισης. Έπειτα, εκτελεί κλήσεις συστήματος της οικογένειας `read()/write()` του VFS, ώστε να μεταφέρει τα δεδομένα από το αρχείο στη θέση μνήμης, αν έχουμε μια αίτηση ανάγνωσης, ή το αντίστροφο, αν έχουμε μια αίτηση εγγραφής. Τέλος, ειδοποιεί τον guest ότι τα δεδομένα είναι έτοιμα, όπως θα έκανε και ένας κανονικός ελεγκτής.

Στην παραπάνω ανάλυση, υποθέσαμε σιωπηλά ότι ένα αρχείο εικονικού δίσκου είναι ένα κανονικό αρχείο στο σύστημα αρχείων του host. Στην γενική περίπτωση, το αρχείο μπορεί να βρίσκεται σε κάποιον απομακρυσμένο κόμβο σύμφωνα με τα πρότυπα του NAS ή να βρίσκεται σε κάποιο shared-disk filesystem. Πέρα από τα προηγούμενα, ένας εικονικός δίσκος μπορεί να μην υλοποιείται καν ως ένα αρχείο του host που παρέχει την γνωστή VFS διεπαφή προς τον οδηγό εικονικών συσκευών του hypervisor. Μπορεί να είναι ένα ένα ειδικό virtualized storage object με εξειδικευμένη διεπαφή και διευθυνσιοδότηση και το οποίο θα εφαρμόζει την δική του μέθοδο αντιστοίχισης για να παρέχει τα κατάλληλα δεδομένα στις αιτήσεις τομέων του guest. Σε όλες τις παραπάνω περιπτώσεις, η δομή αντιστοίχισης είναι μη στοιχειώδης και το ίδιο και

η διαδικασία της αντιστοίχισης τομέων με διευθύνσεις αποθήκευσης. Έτσι υπάρχει η ανάγκη για ανάπτυξη εξειδικευμένων οδηγών μπλοκ στον hypervisor, οι οποίοι θα αντιμετωπίζουν τις ανάγκες της εκάστοτε τεχνολογίας.

Στα περιβάλλοντα υπολογιστικού νέφους, είναι πολύ συχνό επίσης αντί ο εικονικός δίσκος να αντιστοιχίζεται σε ένα κανονικό αρχείο του συστήματος αρχείων, να αντιστοιχίζεται σε ένα αρχείο συσκευής μπλοκ (π.χ. `/dev/sda`), και άρα κατευθείαν σε μια συσκευή μπλοκ (block device) που είναι προσαρτημένη στον host. Η διεπαφή την οποία ο πυρήνας του Linux προσφέρει για τα αρχεία συσκευών μπλοκ, είναι η ίδια με αυτή ενός κανονικού αρχείου, δηλαδή μια γραμμική ακολουθία bytes. Τώρα όμως, αυτό το αρχείο δεν βασίζεται στις λειτουργίες του συστήματος αρχείων αλλά ο πυρήνας εσωτερικά αποθηκεύει τα δεδομένα των μπλοκ αρχείου (file blocks) κατευθείαν σε μπλοκ της υπάρχουσας πραγματικής συσκευής, με μια απλή γραμμική αντιστοίχιση. Με αυτόν τον τρόπο, μπορούμε να έχουμε αυξημένη απόδοση, αφού παρακάμπτεται το επίπεδο του συστήματος αρχείων, ενώ υπάρχει δυνατότητα να χρησιμοποιήσουμε τα πλεονεκτήματα που προσφέρει το μοντέλο SAN, αν ενοποιήσουμε πρώτα σε μια λογική συσκευή μπλοκ στον host διάφορα σύνολα από μπλοκ. Έτσι, ο εικονικός δίσκος στον guest ουσιαστικά αναφέρεται σε ένα σύνολο από μπλοκ που μπορεί να προέρχονται από διαφορετικούς φυσικούς δίσκους κατά το πρότυπο SAN. Αυτή η προσέγγιση δεν απαιτεί επίσης και καμία διαφοροποίηση στους οδηγούς μπλοκ του hypervisor, αφού συντηρείται το μοντέλο ενός αρχείου με μπλοκ αρχείου.

Οι εικονικοί δίσκοι είναι ένα παράδειγμα εικονικοποίησης αποθήκευσης (storage virtualization), λογικής αφαίρεσης δηλαδή, που απαγκιστρώνει την διεπαφή του storage από τις ρητές υποδομές υλικού της φυσικής αποθήκευσης.

Το είδος της δομής αντιστοίχισης, η λογική αντιστοίχισης και το σύνολο των χαρακτηριστικών που μπορεί να προσφέρει ένας εικονικός δίσκος συνθέτουν την μορφή αποθήκευσης (format) του εικονικού δίσκου. Υπάρχουν πολλές μορφές αποθήκευσης εικονικών δίσκων, όπως το qcow2 (QEMU), το vmdk (VMWare), το vdi (Orale VM Virtualbox), το vhd (Microsoft Virtual PC) και το hdd (Parallels).

VirtIO Η πλήρης εικονικοποίηση είναι ένα ελκυστικό χαρακτηριστικό, γιατί επιτρέπει την εκτέλεση οποιουδήποτε λειτουργικού συστήματος σε μια εικονική μηχανή, χωρίς αλλαγές. Δυστυχώς όμως, είναι αργή, γιατί ο hypervisor θα πρέπει να προσο-

μοιώνει πραγματικές συσκευές, όπως κάρτες δικτύου και σκληρούς δίσκους. Αυτή η προσομοίωση εισάγει πολυπλοκότητα και επιβάρυνση στην απόδοση. Το VirtIO [Rus08] είναι ένα πρότυπο για συσκευές δικτύου και δίσκους, όπου ο οδηγός συσκευής μπλοκ του guest γνωρίζει ότι τρέχει σε ένα εικονικό περιβάλλον και συνεργάζεται με τον hypervisor. Αυτό αυξάνει την απόδοση που λαμβάνει ο guest και συνδυάζει τα περισσότερα πλεονεκτήματα απόδοσης της παραεικονικοποίησης. Το VirtIO επιλέχθηκε ως η κύρια πλατφόρμα για εικονικοποίηση Ε/Ε στο KVM. Η υλοποίηση για τον host γίνεται στον χώρο χρήστη από το QEMU οπότε δεν χρειάζεται ειδικός οδηγός συσκευής για τον host.
Εσωτερικά, δουλεύει μέσω μοιραζόμενης μνήμης ανάμεσα στον host και στον guest. Ο δεύτερος χρησιμοποιεί αντί για κανονικούς οδηγούς συσκευής μπλοκ τους ειδικούς VirtIO οδηγούς συσκευής, οι οποίοι αντί να εκδίδουν εντολές προς τον εικονικό ελεγκτή της συσκευής προκαλώντας διακοπές, γράφουν τα δεδομένα σε ένα κυκλικό buffer (ring) και ενημερώνουν τον hypervisor στον host. Αυτός λαμβάνει από εκεί τα δεδομένα και συνεχίζει την διαδικασία που περιγράψαμε παραπάνω. Αντίστοιχα, όταν ο host έχει δεδομένα για τον guest, τα εισάγει στο κυκλικό buffer και τον ειδοποιεί. Αυτή η απλοποίηση στην μεταφορά των απαραίτητων δεδομένων από τον guest στο host και αντίστροφα οδηγεί σε μια σημαντική αύξηση της απόδοσης.

2.2.4 Στιγμιότυπα και κλώνοι

Στα υπολογιστικά συστήματα ένα στιγμιότυπο (snapshot) αντανακλά την κατάσταση του συστήματος μια δεδομένη χρονική στιγμή. Είναι ένα μόνο για ανάγνωση (read-only) αντίγραφο των δεδομένων, «παγωμένο» σε μια συγκεκριμένη χρονική στιγμή. Πιο συγκεκριμένα, τα στιγμιότυπα εικονικών μηχανών είναι στιγμιότυπα της κατάστασης, των δεδομένων του δίσκου και των ρυθμίσεων μιας εικονικής μηχανής σε μια δεδομένη χρονική στιγμή. Στην περίπτωσή μας περιοριζόμαστε στο να ταυτίσουμε ένα στιγμιότυπο με ένα αντίγραφο των δεδομένων ενός εικονικού δίσκου σε μια συγκεκριμένη χρονική στιγμή, το οποίο είναι μόνο για ανάγνωση. Άλλοι τύποι στιγμιοτύπων είναι στιγμιότυπα αρχείων, στιγμιότυπα συστημάτων αρχείων και στιγμιότυπα βάσεων δεδομένων. Ένα στιγμιότυπο επιτρέπει την επαναφορά της εικονικής μηχανής σε ένα σημείο στο παρελθόν, αναιρώντας τις αλλαγές που έλαβαν χώρα μετέπειτα από την δημιουργία του. Αυτό είναι χρήσιμο σαν μια backup τεχνική για το VM, για

παράδειγμα πριν από την διεξαγωγή μιας ριψοκίνδυνης λειτουργίας όπως ο έλεγχος για καινούργιο λογισμικό, η εξέταση ενός ιού ή η εφαρμογή ενός service pack. Πέρα από ένα σημείο επαναφοράς της εικονικής μηχανής, τα στιγμιότυπα μπορούν να χρησιμοποιηθούν και ως πηγή εξόρυξης δεδομένων (data mining). Ένα στιγμιότυπο μπορεί συνήθως να δημιουργηθεί ακαριαία και με την εικονική μηχανή σε λειτουργία. Ο κανονικός εικονικός δίσκος και τα δεδομένα του, συνεχίζουν να είναι διαθέσιμα στις εφαρμογές με την δυνατότητα μεταβολών.

Ένας κλώνος (clone) είναι ένα εγγράψιμο στιγμιότυπο, δηλαδή αντανακλά την κατάσταση του συστήματος μια δεδομένη χρονική στιγμή, η οποία στη συνέχεια δεν παραμένει μόνο για ανάγνωση, αλλά επιτρέπει και μεταβολές. Συνεπώς, στην περίπτωσή μας, ένας κλώνος είναι ένα αντίγραφο των δεδομένων ενός εικονικού δίσκου σε μια συγκεκριμένη χρονική στιγμή, που στη συνέχεια μπορεί να διαμορφωθεί ανεξάρτητα από τον αρχικό εικονικό δίσκο. Έτσι είναι εφικτό αρχίζοντας από μια κατάσταση του εικονικού δίσκου να δημιουργήσουμε πολλές ταυτόχρονα ζωντανές εκδοχές του, σε αντίθεση με τα στιγμιότυπα που επιτρέπουν την αποθήκευση διαφορετικών καταστάσεων του εικονικού δίσκου, με μία όμως μόνο ζωντανή εκδοχή του. Οι κλώνοι είναι χρήσιμοι σε μεγάλα cloud περιβάλλοντα εικονικοποίησης, για την γέννηση πολλών εικονικών μηχανών από ένα σύνολο πρότυπων εικονικών δίσκων με συγκεκριμένα λειτουργικά συστήματα και παραμετροποίηση.

Υπάρχουν αρκετές τεχνικές για την υλοποίηση στιγμιοτύπων και κλώνων. Μια απλοϊκή προσέγγιση θα ήταν να αντιγραφούν ρητά όλα τα μπλοκ δεδομένων του εικονικού δίσκου σε μια νέα τοποθεσία του φυσικού δίσκου (split mirror approach) αλλά στην πραγματικότητα χρησιμοποιούνται ευφυέστερες προσεγγίσεις, όπως το Copy-On-Write και η ελαφριά παραλλαγή του Redirect-On-Write.

3

Σχεδιασμός και Ανάλυση Μορφών Αποθήκευσης Εικονικών Δίσκων

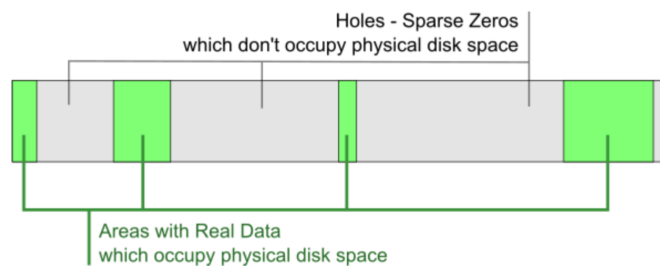
Σε αυτό το κεφάλαιο αρχικά θα διερευνήσουμε τις τεχνικές εξοικονόμησης χώρου που μπορούν να χρησιμοποιηθούν στα πλαίσια των εικονικών δίσκων και θα αναλύσουμε τις σχεδιαστικές επιλογές που ανακύπτουν. Παράλληλα θα εξερευνήσουμε υπάρχουσες λύσεις, σχεδιαστικά πρότυπα και μορφές δίσκων που μπορούν να προσφέρουν αποδοτική εξοικονόμηση χώρου με επιδόσεις πρωτογενούς αποθήκευσης. Στη συνέχεια υπό το πρίσμα του πολυχώρου σχεδιαστικών επιλογών, θα παραθέσουμε τα προβλήματα που αντιμετωπίζουν υπάρχοντες σχεδιασμοί και θα προχωρήσουμε στην σταδιακή παρουσίαση του δικού μας σχεδιασμού. Τέλος, θα προχωρήσουμε στην θεωρητική του αξιολόγηση και στην κατασκευή ενός μοντέλου απόδοσης.

3.1 Τεχνικές εξοικονόμησης αποθηκευτικού Χώρου

3.1.1 Αραιά αρχεία

Τα αραιά αρχεία (sparse files) είναι αρχεία των οποίων όσα μπλοκ περιέχουν μηδενικά, δεν αποθηκεύονται πραγματικά στον δίσκο. Αυτό σημαίνει ότι μπορεί να υφίσταται ένα αρχείο που φαινομενικά να καταλαμβάνει 10GB δηλαδή να είναι διευθυνσιοδοτήσιμα (addressable) 10GB, ενώ ο χώρος που καταλαμβάνει στον δίσκο να είναι μόλις 1GB, αφού τα υπόλοιπα 9GB να είναι μπλοκ που περιέχουν μηδενικά και δεν αποθηκεύονται. Αυτό το χαρακτηριστικό, μπορεί να αποβεί ιδιαίτερα χρήσιμο σε περι-

πτώσεις που ο δίσκος μπορεί να μην χρησιμοποιείται ποτέ ολόκληρος, κάτι που είναι αρκετά σύνηθες για εικονικούς δίσκους. Εάν μια εικονική μηχανή ποτέ δεν γεμίζει ολόκληρο τον εικονικό δίσκο, δηλαδή υπάρχουν περιοχές του που δεν έχουν τίποτα άλλο παρά μηδενικά, είναι εφικτή η εξοικονόμηση χώρου στον φυσικό δίσκο, χρησιμοποιώντας για την αναπαράσταση του εικονικού δίσκου, ένα αραιό αρχείο. Η διαφορά μεταξύ των δεδομένων που είναι διευθυνσιοδοτήσιμα, δηλαδή μπορούν να αναφερθούν, και αυτά που πραγματικά αποθηκεύονται φαίνεται στο σχήμα 3.1.



Σχήμα 3.1: *Αραιό αρχείο*

Τα αραιά αρχεία, δεν υποστηρίζονται από όλα τα συστήματα αρχείων. Για παράδειγμα, τα συστήματα αρχείων της οικογένειας FAT, το HPFS, το HFS+ και το UDF είναι συστήματα αρχείων που δεν τα υποστηρίζουν. Ένα σύστημα αρχείων που τα υποστηρίζει, δεσμεύει ένα μπλοκ δεδομένων μόνο όταν μιας διεργασία απαιτεί να γράψει δεδομένα σε αυτό. Πριν από αυτήν την αίτηση, το σύστημα αρχείων γνωρίζει ότι το μπλοκ είναι διευθυνσιοδοτήσιμο αλλά είναι μηδενικό, οπότε παρέχει διαφανώς ένα μπλοκ γεμάτο μηδενικά. Αυτό το κόλπο επιτυγχάνεται μέσω των μεταδεδομένων του αρχείου. Κατά την ανάγνωση αραιών αρχείων, το σύστημα αρχείων διαφανώς μετατρέπει τα μεταδεδομένα που αναπαριστούν τα άδεια μπλοκ σε «πραγματικά» μπλοκ γεμάτα μηδενικά, με την εφαρμογή να μην είναι ενήμερη για αυτήν την μετατροπή. Προφανώς, άπαξ και το εν λόγω μπλοκ αποκτήσει κάποια μη μηδενικά δεδομένα, τα δεδομένα γράφονται στον δίσκο με κατάλληλο τρόπο και συνάμα αυξάνεται και ο χώρος που καταλαμβάνει το αρχείο στο δίσκο. Ο κενός χώρος, δηλαδή τα μπλοκ γεμάτα μηδενικά, ανάμεσα στα δεσμευμένα μπλοκ είναι γνωστός και ως τρύπα αρχείου (file hole). Οι τρύπες αρχείων ουσιαστικά εκμεταλλεύονται τον διαχωρισμό του επιπέδου αρχείου από το επίπεδο του δίσκου, ή με άλλα λόγια εκμεταλλεύονται την δομή αντιστοίχισης ανάμεσα στα μπλοκ αρχείου (file blocks) και στα λογικά μπλοκ του δίσκου (logical blocks). Για να κάνουμε ορατό το μηχανισμό των αραιών αρχείων θα χρησιμοποιήσουμε ως παράδειγμα αναφοράς τις υλοποιήσεις των συστημάτων αρχείων ext2/ext3/ext4.

Στα συστήματα αρχείων ext, τα μεταδεδομένα ενός αρχείου αποθηκεύονται σε μια εγγραφή inode (inode entry), η οποία είναι μια δομή τόσο στον δίσκο όσο και στην κύρια μνήμη, που περιέχει διάφορες πληροφορίες για το αρχείο. Ανάμεσα σε αυτές, το `i_size` πεδίο περιέχει το ονομαστικό μέγεθος του αρχείου (nominal file size) συμπεριλαμβανομένων και των τρυπών, ενώ το `i_blocks` πεδίο περιέχει τον αριθμό των μπλοκ δεδομένων που πράγματι δεσμεύονται στο δίσκο για αυτό το αρχείο. Συνεπώς το ονομαστικό μέγεθος του αρχείου είναι ανεξάρτητο από το πραγματικό μέγεθος που αυτό καταλαμβάνει στον δίσκο. Αυτός ο διαχωρισμός μπορεί να γίνει ορατός και σε μια εφαρμογή επιπέδου χρήστη μέσω της κλήσης συστήματος `stat()`, το οποίο επιστρέφει μια δομή που ανάμεσα σε άλλα διαθέτει το πεδίο `st_size`, που περιέχει την τιμή του `i_size`, και το πεδίο `st_blocks` που περιέχει την τιμή του `i_blocks`.

Τώρα που εδραιώσαμε αυτόν τον διαχωρισμό ανάμεσα στο ονομαστικό μέγεθος ενός αρχείου και τα μπλοκ και άρα τον χώρο που πραγματικά καταλαμβάνει αυτό το αρχείο στον δίσκο, θα επικεντρωθούμε στον μηχανισμό που φέρνει εις πέρας την ιδέα των αραιών αρχείων. Όπως έχουμε περιγράψει προηγουμένως, η μετάφραση μεταξύ ενός file offset και ενός αριθμού λογικού μπλοκ, πραγματοποιείται χάρη σε μια δομή αντιστοίχισης, τοποθετημένη στο inode του αρχείου.

Στην περίπτωση των ext2 και ext3 που διατηρούν την παλιά δομή δεικτών, το σύστημα αρχείων αντί να δεσμεύει μπλοκ γεμάτα μηδενικά, προτιμά να χρησιμοποιεί την τιμή 0 σε όλους τους δείκτες της δομής που αντιστοιχούν σε μπλοκ αρχείου γεμάτα μηδενικά. Για παράδειγμα, εάν υπάρχει μια τρύπα στο πρώτο μπλοκ αρχείου, ο πρώτος δείκτης του `i_block` πίνακα θα περιέχει την τιμή 0 και όχι την διεύθυνση ενός μπλοκ δεδομένων που περιέχει μηδενικά. Αυτή η τεχνική ακολουθείται και για τους δείκτες των έμμεσων μπλοκ έτσι ώστε να σηματοδοτήσουν ότι δεν αναφέρονται σε πραγματικά μπλοκ του δίσκου.

Εάν θέλουμε να είμαστε διεξοδικοί, στον κώδικα του ext4 ο οποίος υποστηρίζει την παλιά δομή δεικτών, η μετάφραση από τον αριθμό μπλοκ αρχείου στον αριθμό λογικού μπλοκ, πραγματοποιείται μέσω της συνάρτησης `ext4_map_blocks()`, η οποία καλεί τις `ext4_block_to_path()` και `ex4_get_branch()`. Αυτές οι συναρτήσεις λειτουργούν σε μια δομή `struct ext4_map_blocks`, η οποία αποθηκεύει την αντιστοίχιση. Μέσα σε αυτήν την δομή υπάρχει το bitfield πεδίο `m_flags`, του οποίου η σημαία `EXT4_MAP_MAPPED`, περιγράφει εάν η αντιστοίχιση είναι πραγματική ή εάν

αντανακλά μια τρύπα. Συνεπώς, εάν ένας μηδενικός δείκτης βρεθεί σε κάποιο σημείο της διάσχισης των δεικτών της δομής που βρίσκεται αποθηκευμένη στο `i_block`, η `ex4_get_branch()` επιστρέφει χωρίς να γεμίσει την δομή `ext4_map_blocks` και χωρίς να «σηκώσει» την σημαία `EXT4_MAP_MAPPED`.

Στην περίπτωση του ext4 όμως, χρησιμοποιείται συνήθως ένα extent tree αντί για την παλιά ιεραρχική δομή δεικτών και έτσι οι τρύπες ανακύπτουν φυσικά από την απουσία κόμβων-φύλλων που θα έπρεπε να περιγράφουν αυτήν την περιοχή των μπλοκ αρχείου. Για παράδειγμα, εάν υπάρχει μια τρύπα που αρχίζει από το 50-οστό μπλοκ αρχείου και εκτείνεται για 10 μπλοκ αρχείου, τότε θα υπάρχει ένας κόμβος-φύλλο που θα περιγράφει τα extents που αντιστοιχούν στα μπλοκ αρχείου 0-49, ένας κόμβος φύλλο που θα περιγράφει τα extents που αντιστοιχούν στα μπλοκ αρχείου 60 έως το τέλος, αλλά δεν θα υπάρχει κόμβος φύλλο που θα περιέχει πληροφορίες για την αντιστοίχιση των μπλοκ αρχείου 50-59.

Άλλα συστήματα αρχείων μπορεί να χρησιμοποιούν τους δικούς τους αλγορίθμους για την υποστήριξη αραιών αρχείων, αλλά γενικά αυτή η λειτουργικότητα υλοποιείται μέσω τεχνασμάτων στο επίπεδο αντιστοίχισης ανάμεσα στα μπλοκ αρχείου και τα λογικά μπλοκ, το οποίο επίπεδο ενυπάρχει στα μεταδεδομένα του αρχείου.

Ο χώρος δεσμεύεται μόνο όταν πραγματικά χρειάζεται και συνεπώς μεγάλα αρχεία μπορούν να δημιουργηθούν ακόμα και όταν δεν υπάρχει επαρκής ελεύθερος χώρος στο σύστημα αρχείων. Επίσης τα αραιά αρχεία μειώνουν και το χρόνο της δημιουργίας ενός τέτοιου μεγάλου αρχείου, αφού το σύστημα αρχείων δεν χρειάζεται ούτε να δεσμεύσει μπλοκ ούτε να γράψει μηδενικά σε όλο το χώρο που παραλείπεται.
Από την άλλη, τα αραιά αρχεία είναι επιρρεπή στον κατακερματισμό (fragmentation), αφού συνεχόμενα μπλοκ αρχείου δεσμεύονται σταδιακά και συνήθως όχι σε συνεχόμενες περιοχές της μνήμης. Επίσης, αναφορές για τον ελεύθερο χώρο στο δίσκο μπορεί να είναι ανακριβείς χάρη σε αποκλίσεις ανάμεσα σε εργαλεία που χρησιμοποιούν τα δεσμευμένα μπλοκ και αυτά που χρησιμοποιούν το ονομαστικό μέγεθος ενός αρχείου. Ακόμη, τα αραιά αρχεία δεν υποστηρίζονται από όλες τις εφαρμογές αντιγράφων ασφαλείας (backup). Αυτό σημαίνει ότι κάθε εφαρμογή backup ή αντιγραφής, η οποία δεν αναγνωρίζει και δεν εκμεταλλεύεται την σημασιολογία των αραιών αρχείων, υπάρχει περίπτωση να αντιγράψει ολόκληρο το αρχείο στο δίσκο, μαζί με τα μηδενικά μπλοκ, και άρα να σπαταλήσει ανούσια, πολύ χώρο στον δίσκο προορισμού.

Ως παράδειγμα μιας εφαρμογής που υποστηρίζει αραιά αρχεία αναφέρουμε το GNU cp. Κατά προεπιλογή ή με βάση την επιλογή `--sparse=auto`, τα αραιά αρχεία ανιχνεύονται από μια ευριστική τεχνική και το αντίστοιχο αρχείο προορισμού γίνεται και αυτό αραιό. Με την επιλογή `--sparse=always` δημιουργείται ένα αραιό αρχείο προορισμού, όποτε το αρχείο πηγή περιέχει μια αρκούντως μεγάλη ακολουθία από μηδενικά bytes, ενώ η επιλογή `--sparse=never` αποτρέπει την δημιουργία αραιών αρχείων. Οι GNU tar και cpio υποστηρίζουν αραιά αρχεία.

3.1.2 COW, στιγμιότυπα και BTRFS

Η βασική αρχή της τεχνικής Copy-On-Write είναι ότι όταν πολλοί δράστες χρησιμοποιούν στην αρχή πανομοιότυπα αντίγραφα κάποιων δεδομένων, τα οποία περιστασιακά μπορεί να χρειαστεί να υποστούν τοπικές μεταβολές από τον καθένα, τότε είναι απαραίτητο να δημιουργηθεί εξ αρχής ένα ξεχωριστό αντίγραφο αυτών των δεδομένων για κάθε τέτοιο δράστη. Αντ' αυτού, είναι δυνατόν όλοι οι δράστες να έχουν πρόσβαση σε ένα στιγμιότυπο των δεδομένων μέσω κάποιου δείκτη, με την προϋπόθεση πως την πρώτη φορά που χρειαστεί κάποιος να μεταβάλλει τα δεδομένα, και μόνο τότε, πρέπει πρώτα να δημιουργηθεί ένα τοπικό αντίγραφο, το οποίο θα υποστεί τις μεταβολές και με το οποίο θα συνεχίσουν να δουλεύουν. Τα αρχικά δεδομένα παραμένουν αμετάβλητα και είτε συλλέγονται ως σκουπίδια όταν κανένας δράστης δεν δείχνει σε αυτά, είτε παραμένουν immutable για πάντα.
Ο διαμοιρασμός οδηγεί στην εξοικονόμηση των πόρων και άρα η τεχνική COW ενδείκνυται σε περιπτώσεις που υπάρχουν πολλές διαφορετικές διεργασίες που χρησιμοποιούν τον ίδιο πόρο και κάθε μία τους έχει μια μικρή πιθανότητα να χρειαστεί να τον μεταβάλει. Το όνομα Copy-On-Write δόθηκε στην τεχνική αυτή, γιατί κάθε φορά που ένας δράστης επιχειρεί να μεταβάλλει την μοιραζόμενη πληροφορία, πρέπει πρώτα να δημιουργήσει ένα ξεχωριστό, ιδιωτικό αντίγραφό της, ώστε οι αλλαγές να μην γίνουν ορατές και σε άλλους δράστες.

Μια ευρέως διαδεδομένη εφαρμογή του παραπάνω μηχανισμού είναι η κλήση συστήματος `fork()` στα σύγχρονα συστήματα Unix. Αυτή η κλήση δημιουργεί ένα αντίγραφο της τρέχουσας διεργασίας, με την διαφορά ότι η νέα διεργασία δεν λαμβάνει στην πραγματικότητα το δικό της αντίγραφο για κάθε σελίδα μνήμης, μέχρι είτε η διεργασία πατέρας είτε η διεργασία παιδί, να επιχειρήσουν μια εγγραφή σε μια τέτοια

σελίδα. Τότε και μόνο τότε η σελίδα αντιγράφεται.

Στο πεδίο του ενδιαφέροντός μας, η τεχνική COW είναι μια από τις βασικές πολιτικές ενημέρωσης όταν μεταβάλλονται δεδομένα σε μπλοκ του δίσκου. Το μπλοκ αρχικά διαβάζεται από τον δίσκο και μεταφέρεται στην κύρια μνήμη, υφίσταται αλλαγές και στην συνέχεια γράφεται στον δίσκο σε μια καινούργια τοποθεσία, αφήνοντας τα δεδομένα του αρχικού μπλοκ αμετάβλητα. Έτσι η τεχνική COW χρησιμοποιείται ευρέως σε συστήματα αποθήκευσης για τους παρακάτω λόγους [CWTX14]:

- Προστασία δεδομένων: συστήματα αρχείων όπως το WAFL, ZFS και το BTRFS, χρησιμοποιούν την πολιτική ενημέρωσης COW, ώστε να αποτρέψουν πιθανή απώλεια δεδομένων σε περιπτώσεις αστοχίας (system crash). Επίσης η πολιτική αυτή αξιοποιείται στα παραπάνω συστήματα αρχείων για την παροχή στιγμιοτύπων.
- Αύξηση της απόδοσης: log-structure συστήματα αρχείων όπως το LFS χρησιμοποιούν την πολιτική ενημέρωσης COW για να μετατρέψουν ένα μοτίβο πολλών μικρών τυχαίων εγγραφών σε μια μεγάλη σειριακή εγγραφή, που εκμεταλλεύεται την καλύτερη απόδοση του δίσκου στις σειριακές λειτουργίες Ε/Ε.
- Ενημέρωση δεδομένων σε ειδικά μέσα: μέσα Write-Once-Read-Many όπως οι οπτικοί δίσκοι, χρησιμοποιούν την τεχνική COW ώστε να υλοποιήσουν τυχαίες εγγραφές. Flash-memory συστήματα αρχείων όπως το CFFS, FlashFS και JFFS χρησιμοποιούν την τεχνική COW για να βελτιστοποιήσουν τις λειτουργίες ενημέρωσης, βελτιώνοντας την απόδοση εγγραφής και μετριάζοντας την φθορά.

Πέρα από την COW, η άλλη πολιτική ενημέρωσης είναι η Update-In-Place (UIP), κατά την οποία το μπλοκ διαβάζεται και μεταφέρεται στην κύρια μνήμη, υφίσταται αλλαγές και στη συνέχεια γράφεται στον δίσκο στην αρχική του τοποθεσία, πάνω από τα παλιά δεδομένα.

Ένα μειονέκτημα του COW είναι πως η ανάγκη του να γράφει πάντα σε νέα σημεία, τείνει να διασκορπίζει τα δεδομένα σε όλο το δίσκο, προκαλώντας κατακερματισμό (fragmentation). Ο κατακερματισμός, συνίσταται στην ύπαρξη δεδομένων που δεν είναι αποθηκευμένα σε συνεχόμενες φυσικές θέσεις του δίσκου, παρόλο που σε ανώτερο επίπεδο είναι συνεχόμενα. Ο κατακερματισμός είναι καταστροφικός για την απόδοση όταν υπάρχει σε περιβάλλοντα εικονικοποίησης, γιατί αυξάνει την καθυστέρηση ολοκλήρωσης μιας λειτουργίας Ε/Ε (latency). Από την στιγμή που τα δεδομένα είναι πιο διάσπαρτα, απαιτείται περισσότερη ώρα για την εύρεση και προσπέλαση ολόκληρων

αρχείων και συνεπώς η τεχνική COW δεν συνίσταται για εικονικοποίηση ή κάθε φορτίο που έχει μικρά και τυχαία μοτίβα εγγραφών.

Επίσης, η τεχνική COW εισάγει μια ανεπιθύμητη επιπλοκή αναδρομικής ενημέρωσης. Αν θεωρήσουμε ότι το σύστημα αρχείων διατηρεί τα μεταδεδομένα του σε ένα μεγάλο B-tree όπως το BTRFS, τότε όταν ένα μπλοκ μεταδεδομένων που βρίσκεται στα φύλλα μεταβάλλεται, πρέπει να μεταβληθεί και ο πατρικός του κόμβος. Αυτό γιατί, το COW θα γράψει το νέο μπλοκ σε μια νέα τοποθεσία στο δίσκο και ο πατέρας του πρέπει να ενημερώσει τον δείκτη του με αυτήν την τοποθεσία. Αυτή όμως η διαδικασία ενημέρωσης αν εφαρμοστεί αναδρομικά για κάθε κόμβο μέχρι την ρίζα του δέντρου, εισάγει παρενέργειες όπως επιπρόσθετες εγγραφές (additional writes) [CNF$^+$09], αλλαγή του μοτίβου λειτουργιών Ε/Ε (I/O pattern alternation) [NP02], και μείωση της απόδοσης (performance degradation) [WKC07]. Τα flash-memory μέσα μάλιστα, θα υποφέρουν ακόμα περισσότερο από τις παρενέργειες των αυξημένων εγγραφών, αφού υποστηρίζουν περιορισμένο αριθμό εγγραφών.

COW για στιγμιότυπα

Η τεχνική COW μπορεί να παρέχει την δυνατότητα για ένα μόνο για ανάγνωση στιγμιότυπο κάποιων δεδομένων σε μια δεδομένη χρονική στιγμή, που μπορεί να συνυπάρχει με μια ζωντανή εγγράψιμη εκδοχή των ίδιων δεδομένων, χωρίς να καταναλώνει περισσότερο χώρο απ᾽ όσο χρειάζεται. Δίχως το COW, στιγμιότυπα μπορούν να ληφθούν με την split-mirror τεχνική, κατά την οποία δημιουργείται ένα off-line αντίγραφο όλων των μπλοκ σε μια νέα τοποθεσία στον δίσκο. Αυτό όμως προϋποθέτει εκ των προτέρων προγραμματισμό, υψηλή επιβάρυνση σε χώρο αποθήκευσης και δεν μπορεί να υποστηρίξει on-line εφαρμογές, εκτός και αν αυτές μπλοκάρουν.

Έτσι λοιπόν, καταφεύγουμε στο COW, ως έναν πιο έξυπνο μηχανισμό για την υλοποίηση στιγμιοτύπων. Υποθέτουμε ότι έχουμε ένα τυπικό περιβάλλον υπολογιστικού νέφους, όπου στον χώρο αποθήκευσης δεσμεύουμε έναν αρχικό τόμο (source volume) για την αναπαράσταση του εικονικού δίσκου μιας εικονικής μηχανής. Παρουσιάζουμε τις δυο πιο διαδεδομένες προσεγγίσεις:

Copy-On-Write Κατά την στιγμή δημιουργίας του snasphot, ένας μικρός τόμος δεσμεύεται ως snasphot volume, αντίστοιχα με το source volume. Με την πρώτη αίτηση

εγγραφής σε ένα μπλοκ δεδομένων μετά την λήψη του στιγμιοτύπου, τα αρχικά δεδομένα του μπλοκ αντιγράφονται από το source volume στο snapshot volume και μετά την αντιγραφή η εγγραφή των νέων δεδομένων πραγματοποιείται στο source volume. Αυτό έχει ως αποτέλεσμα η εικόνα των δεδομένων κατά την στιγμή της δημιουργίας του στιγμιοτύπου να διατηρείται στον συνδυασμό του source volume και του snapshot volume. Τόσο οι λειτουργίες ανάγνωσης, όσο και οι λειτουργίες εγγραφής μετά την πρώτη μεταβολή ενός μπλοκ, πραγματοποιούνται στο source volume. Αν μια διεργασία επιχειρήσει να προσπελάσει το snasphot κάποια στιγμή στο μέλλον, το κάνει χάρη σε ειδικά μεταδεδομένα που διατηρούνται ώστε να παρακολουθούν τα μπλοκ που έχουν αλλάξει από την στιγμή λήψης του στιγμιοτύπου.

Redirect-On-Write Μια άλλη εναλλακτική είναι η πολιτική Redirect-On-Write, που αποτελεί ελαφρά παραλλαγή της Copy-On-Write. Η τελευταία, απαιτεί 3 λειτουργίες Ε/Ε για κάθε πρώτη εγγραφή σε ένα μπλοκ μετά το στιγμιότυπο: μια για ανάγνωση του αρχικού μπλοκ από το source volume, μια για εγγραφή του αρχικού μπλοκ στο snapshot volume και μια για εγγραφή των νέων δεδομένων στο source volume. Αυτές οι λειτουργίες Ε/Ε πραγματοποιούνται στο κρίσιμο μονοπάτι, κάτι που ενδέχεται να επηρεάσει αρνητικά την απόδοση της εφαρμογής. Για να ξεπεραστεί αυτό, η Redirect-On-Write αφήνει το αρχικό μπλοκ αμετάβλητο στο source volume και γράφει τα νέα δεδομένα στο snapshot volume. Συνεπώς, αν ένα μπλοκ πρέπει να μεταβληθεί, το σύστημα αποθήκευσης απλώς ανακατευθύνει τους δείκτες αυτού του μπλοκ σε ένα άλλο μπλοκ και γράφει τα δεδομένα εκεί. Έτσι εξαλείφονται οι επιπλέον λειτουργίες Ε/Ε του Copy-On-Write. Μετά την λήψη του στιγμιοτύπου, κάθε αίτηση εγγραφής πραγματοποιείται στο snapshot volume, ενώ οι αιτήσεις ανάγνωσης μπορεί να χρειαστεί να προσπελάσουν το source volume ή το snapshot volume, ανάλογα με το εάν το μπλοκ έχει μεταβληθεί ή όχι από την στιγμή δημιουργίας του στιγμιοτύπου. Το σύστημα και πάλι διατηρεί μεταδεδομένα ώστε να μπορεί να κατευθύνει τις αιτήσεις στα κατάλληλα μπλοκ. Εδώ, η εικόνα των δεδομένων κατά την λήψη του στιγμιοτύπου βρίσκεται από αυτήν την στιγμή και μετά στο source volume.

Με την τεχνική COW, η ανάγνωση του στιγμιοτύπου εμπλέκει μια διαδικασία εύρεσης της θέσης κάθε μπλοκ (source ή snapshot volume), γεγονός που μπορεί να εισάγει μια υπολογιστική επιβάρυνση, η οποία δεν υπάρχει στην ROW. Από την άλλη, κατά την διάρκεια ζωής ενός συστήματος αρχείων που χρησιμοποιεί την ROW, θα υπάρ-

ξει ένας κατακερματισμός, αφού πολλά μπλοκ δεδομένων δεν θα είναι πλέον συνεχόμενα και το σύστημα αρχείων θα πρέπει να ασχολείται με την εύρεση των διάσπαρτων μπλοκ. Σε γενικές γραμμές, το Copy-On-Write αποδίδει καλά σε εφαρμογές με πολλές αναγνώσεις (read-intensive), ενώ το Redirect-On-Write σε εφαρμογές με πολλές εγγραφές (write-intensive) [XL+06].

Κάποιες άλλες παραλλαγές των παραπάνω είναι:

- Copy-on-write με αντιγραφή στο παρασκήνιο (background copy): χρησιμοποιεί copy-on-write για ακαριαία δημιουργία του στιγμιοτύπου και στη συνέχεια προαιρετικά εκκινεί μια διαδικασία αντιγραφής στο παρασκήνιο, η οποία αντιγράφει τα μπλοκ των δεδομένων από το source volume στο snapshot volume. Έτσι αυτό το αντίγραφο μπορεί να χρησιμοποιηθεί ως κλώνος του στιγμιοτύπου.
- Log-structure file architecture: αυτή η λύση χρησιμοποιεί αρχεία καταγραφής (log-files) ώστε να παρακολουθήσει τις εγγραφές στο αρχικό volume. Όταν τα δεδομένα πρέπει να επανακτηθούν, γίνεται επαναφορά (rollback) των συναλλαγών που έχουν καταγραφεί στο αρχείο καταγραφής. Κάθε αίτηση εγγραφής στο αρχικό volume καταγράφεται στο αρχείο καταγραφής όπως σε μια σχεσιακή βάση δεδομένων.
- Συνεχόμενη προστασία δεδομένων (Continuous data protection - CDP): δημιουργεί ένα στιγμιότυπο κάθε φορά που πραγματοποιείται μια αλλαγή στον αρχικό δίσκο

BTRFS

Τώρα θα εξετάσουμε το BTRFS σύστημα αρχείων, το οποίο χρησιμοποιεί κατά κόρον την τεχνική COW και εγγενώς υποστηρίζει στιγμιότυπα και κλώνους. Ο λόγος που το παρουσιάζουμε είναι διττός. Αρχικά για να εισάγουμε κάποιες βασικές έννοιες γύρω από το COW σε B-trees, έννοιες που θα χρησιμοποιήσουμε και στον δικό μας σχεδιασμό. Το BTRFS αποτελεί άλλωστε και το βασικό σχεδιαστικό πρότυπο, στα ίχνη του οποίου βαδίζει ο μετέπειτα δικός μας σχεδιασμός. Επίσης, μέσω του BTRFS θα παρουσιάσουμε αργότερα μια κομψή προσέγγιση που μπορεί να εκμεταλλευτεί τις εγγενείς δυνατότητές του συστήματος αρχείων για στιγμιότυπα και κλώνους ώστε να λειτουργήσει ως μια λύση που θα εκπληρώσει μερικώς τους σκοπούς του προβλήματος που θέσαμε, δηλαδή αποδοτικούς σε χώρο και χρόνο εικονικούς δίσκους με υποστήριξη στιγμιοτύπων, κλώνων και απαλοιφής διπλοτύπων.

Το BTRFS είναι ένα σύστημα αρχείων βασισμένο στην πολιτική ενημέρωσης COW,

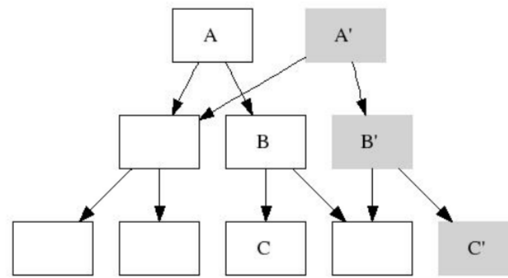
που χρησιμοποιεί ως βασική δομή του ειδικά προσαρμοσμένα B-trees. Αυτά τα B-trees, είναι σχεδόν όπως τα κανονικά B-trees με την διαφορά πως έχει αφαιρεθεί η αλυσίδα δεικτών στα φύλλα και έτσι ένα φύλλο δεν έχει δείκτη στο δεξιό του φύλλο [Rod08]. Εκτός από στιγμιότυπα και κλώνους σε επίπεδο αρχείων, το BTRFS μπορεί να παρέχει στιγμιότυπα ολόκληρων κομματιών του συστήματος αρχείων, εξοικονομώντας έτσι τόσο δεδομένα όσο και μεταδεδομένα. Γι' αυτό το σκοπό εισάγει την έννοια του subvolume, το οποίο από την σκοπιά του χρήστη είναι απλώς ένας ειδικός κατάλογος που επιτρέπει την λήψη ενός στιγμιοτύπου για όλα τα δεδομένα που αναδρομικά περιέχει[1].

Η διάρθρωση των δομών του συστήματος αρχείων στον δίσκο (on-disk layout) είναι ένα δέντρο από τα προαναφερθέντα ελαφρώς παραλλαγμένα B-trees, με την πολιτική Copy-On-Write ως τον μηχανισμό ενημέρωσης. Υπάρχουν πολλά τέτοια δέντρα: ένα για κάθε subvolume για αποθήκευση των μεταδεδομένων αρχείων και καταλόγων, ένα για αποθήκευση των extent checksums, ένα που λειτουργεί ως extent free-space και reference-couner map, ένα για προσπέλαση όλων των αυτών δέντρων, και κάποια ακόμη. Οι κόμβοι των δέντρων αποτελούνται από πολλαπλά items. Τα items των εσωτερικών κόμβων βοηθούν στην πλοήγηση προς τα φύλλα, τα οποία έχουν items που περιγράφουν διάφορα είδη μεταδεδομένων (αρχείου, καταλόγου, δείκτη σε δεδομένα αρχείου και άλλα). Για κάθε τέτοιο είδος μεταδεδομένων υπάρχει και ένας ξεχωριστός τύπος item [btra].

Το BTRFS, αντί να διατηρεί ένα journal που θα καταγράφει τις αλλαγές στα μπλοκ, για να διατηρήσει την ακεραιότητα των δεδομένων, τα γράφει σε μια νέα τοποθεσία στο δίσκο και στη συνέχεια ενημερώνει τους κατάλληλους δείκτες με αυτήν. Βασιζόμενο στην έννοια του COW, κάθε φορά που ένα μπλοκ πρέπει να υποστεί μεταβολή, ολόκληρο το μπλοκ διαβάζεται στη μνήμη, υφίσταται μεταβολές και αργότερα γράφεται στο δίσκο σε μια νέα τοποθεσία. Οι αλλαγές δεν γράφονται στον δίσκο τουλάχιστον μέχρι να ολοκληρωθεί η λειτουργία Ε/Ε και συνήθως για πολλά δευτερόλεπτα. Αυτή η διαδικασία COW που πραγματοποιείται σε ένα μπλοκ ονομάζεται και shadowing.

Όταν μια σελίδα γίνεται shadowed, η τοποθεσία της στο δίσκο αλλάζει, οπότε δημιουργεί την ανάγκη ενημέρωσης του άμεσου προγόνου της στο δέντρο, ο οποίος έχει ένα δείκτη σε αυτήν, με την νέα τοποθεσία. Με άλλα λόγια, αν θέλουμε να ενη-

[1]πέρα από εμφωλευμένα subvolumes



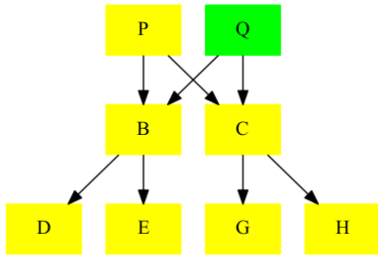
Σχήμα 3.2: *shadowing*

μερώσουμε ένα extent, το item που δείχνει στην διεύθυνση του extent πρέπει επίσης να ενημερωθεί και άρα και αυτός ο κόμβος μεταδεδομένων που περιέχει το item πρέπει να γίνει shadowed. Επειδή στο BTRFS κάθε μεταβολή σε μπλοκ προκαλεί COW, το shadowing εξαπλώνεται μέχρι τον κόμβο ρίζα του συστήματος αρχείων (recursive updates).

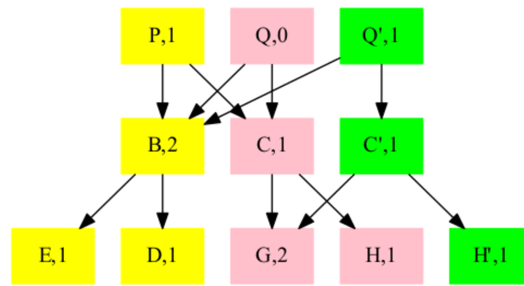
Για παράδειγμα, αν πρέπει να ενημερωθεί το μπλοκ C, τα A, B, C γίνονται shadow με αυτήν την σειρά σε νέα μπλοκ του δίσκου και όταν ολοκληρωθεί η διαδικασία, η νέα ρίζα του δέντρου είναι η A' και τα A, B, C αποδεσμεύονται. Τα παραπάνω φαίνονται στο σχήμα 3.2

Ισχυριστήκαμε νωρίτερα ότι το BTRFS μπορεί να δημιουργήσει ένα στιγμιότυπο για ένα subvolume, διατηρώντας έτσι όχι μόνο τα δεδομένα αρχείων αλλά και την ιεραρχική δομή των καταλόγων και τα μεταδεδομένα των αρχείων. Για να το επιτύχει αυτό, πρέπει να δημιουργήσει ένα στιγμιότυπο για το δέντρο που περιέχει τα items του subvolume. Αυτό μπορεί να πραγματοποιηθεί αποδοτικά με την χρήση των ειδικά προσαρμοσμένων για COW B-trees, καθώς οι κόμβοι μπορούν να είναι μοιραζόμενοι ανάμεσα σε πολλαπλά στιγμιότυπα. Πράγματι, ολόκληρα υποδέντρα μπορούν να είναι μοιραζόμενα ανάμεσα στα στιγμιότυπα. Εάν θελήσουμε λοιπόν να πάρουμε ένα στιγμιότυπο για ένα subvolume, αρκεί να αντιγράψουμε μόνο την ρίζα του αρχικού subvolume. Αυτή θα αντιγραφεί με COW και όλοι οι άλλοι κόμβοι στο δέντρο θα είναι μοιραζόμενοι, όπως φαίνεται στο σχήμα 3.3.

Αν τώρα πρέπει να αλλάξουν τα περιεχόμενα του λόγω αλλαγής στο subvolume Q, πώς θα συμβεί αυτό, χωρίς να αλλοιωθούν τα περιεχόμενα του για το P; Προφανώς με COW. Αρχίζοντας από τον Q όλοι οι κόμβοι στην πορεία προς τον θα γίνουν shadowed (Q, C, H) και τώρα θα υπάρχουν 2 εκδοχές. Ο κόμβος C θα ανήκει στο subvolume P ενώ ο shadowed C' στο Q. Παρομοίως ο H θα ανήκει στο P ενώ ο



Σχήμα 3.3: *δημιουργία στιγμιοτύπου*



Σχήμα 3.4: *μεταβολή περιεχομένων*

shadowed H' στο Q. Ο κόμβος B με τους υποκόμβους του είναι ακόμα μοιραζόμενος. Τα παραπάνω γίνονται ευδιάκριτα στο σχήμα 3.4

Ο Q θα πρέπει να αποδεσμευτεί, γιατί τώρα έχουμε το shadow του Q'. Δεν θα πρέπει να συμβεί όμως το ίδιο με τον C, γιατί ο C πρέπει να είναι προσπελάσιμος από το subvolume P. Προκειμένου να διαχωρίσουμε τους μοιραζόμενους κόμβους από τους μη, ώστε να αποδεσμεύουμε τα άχρηστα μπλοκ μετά το COW, θα χρησιμοποιήσουμε μετρητές αναφορών (reference counters) στα μπλοκ. Διατηρούμε έναν μετρητή αναφορών για κάθε μπλοκ που υποδεικνύει πόσοι δείκτες υπάρχουν σε αυτό το μπλοκ. Όταν ο μετρητής αναφορών φτάσει στο 0, τότε το μπλοκ είναι άχρηστο και αποδεσμεύεται. Εδώ το BTRFS εφαρμόζει τον παραπάνω κανόνα με lazy τρόπο, δηλαδή αυξάνει μόνο τους μετρητές αναφορών των παιδιών του shadowed κόμβου και όχι τους μετρητές αναφορών όλων των κόμβων στο υποδέντρο από κάτω του. Αυτό γιατί ο μετρητής αναφορών δηλώνει πόσοι δείκτες υπάρχουν σε ένα μπλοκ και όχι πόσα subvolumes μοιράζονται το εν λόγω μπλοκ. Έτσι εξοικονομείται και χρόνος καθώς δεν χρειάζεται να προσπελάσουμε επιπλέον κόμβους. Ο μετρητής αναφορών ενός μπλοκ λοιπόν υποδηλώνει ότι είναι μοιραζόμενο όχι μόνο το δεδομένο μπλοκ αλλά και όλο το υποδέντρο από κάτω του. Αυτή είναι μια σημαντική ιδέα που θα χρησιμοποιήσουμε και στην υλοποίησή μας. Για πληρότητα, αναφέρουμε ότι ο μετρητής αναφορών δεν περιέχεται στο ίδιο το μπλοκ όπως υπονοεί το σχήμα 3.4, αλλά σε ένα item σε ένα ξεχωριστό B-tree, το extent allocation tree.

Η υλοποίηση του COW στο BTRFS, που στην πραγματικότητα είναι πιο κοντά στην έννοια του ROW, μπορεί να επιδράσει αρνητικά στην απόδοση, όταν έχουμε πολλές, μικρές, τυχαίες εγγραφές σε μεγάλα αρχεία, διότι θα οδηγήσει στον κατακερματισμό των αρχείων αυτών. Τέτοιες περιπτώσεις είναι μεγάλα αρχεία βάσεων δεδομένων και αρχεία εικονικών δίσκων.

3.1.3 Απαλοιφή διπλοτύπων

Η απαλοιφή διπλοτύπων είναι μια τεχνολογία που μπορεί να χρησιμοποιηθεί σε ένα σύστημα αποθήκευσης για να μειώσει την ποσότητα του αποθηκευτικού χώρου που απαιτείται για ένα αριθμό αρχείων, με τον χωρισμό των δεδομένων σε τεμάχια (chunks), ταυτοποίηση των διπλότυπων τεμαχίων, και αποθήκευση μόνο ενός αντιγράφου από κάθε τεμάχιο. Με αυτόν τον τρόπο αποθηκεύονται λιγότερα δεδομένα και μειώνεται το κόστος. Για παράδειγμα ένα σύστημα αποθήκευσης ηλεκτρονικών μηνυμάτων, μπορεί να περιλαμβάνει 100 αντίγραφα της ίδιας επισύναψης. Με την απαλοιφή διπλοτύπων, αποθηκεύεται πραγματικά μόνο ένα αντίγραφο και τα υπόλοιπα απλώς αναφέρονται σε αυτό για να ανακτήσουν τα δεδομένα.
Η αυτόματη απαλοιφή των διπλότυπων δεδομένων, δηλαδή των κομματιών δεδομένων που εμφανίζονται πάνω από μία φορά, είναι ευρέως διαδεδομένη σε backup συστήματα, καθώς οι διαφορετικές εκδόσεις των backup δεδομένων εμφανίζουν μια μεγάλη ποσότητα πλεονασμού. Στα σύγχρονα περιβάλλοντα εικονικοποίησης και μαζικής αποθήκευσης, η απαλοιφή διπλοτύπων μπορεί επίσης να χρησιμοποιηθεί και για πρωτεύουσα αποθήκευση, ώστε να εξαλείψει διπλότυπα τεμάχια ανάμεσα σε διαφορετικές εικονικές μηχανές ή και ανάμεσα σε στιγμιότυπα των εικονικών μηχανών.

Ο έξτρα αποθηκευτικός χώρος που εξοικονομείται από την απαλοιφή διπλοτύπων, μειώνει το κόστος της υποδομής και μπορεί να χρησιμοποιηθεί για επιπλέον υλοποιήσεις RAID που αυξάνουν την αξιοπιστία. Επίσης, δεν είναι απαραίτητο οι δίσκοι που περιέχουν τα λειτουργικά συστήματα των εικονικών μηχανών να είναι «gold», δηλαδή να περιορίζονται σε κάποιες προεπιλεγμένες εικόνες. Αρκεί να χρησιμοποιηθεί η απαλοιφή διπλοτύπων ώστε να ανιχνευτούν τα πανομοιότυπα μπλοκ του δίσκου ανάμεσα στις εικόνες. Επιπλέον, μπορεί να αυξήσει την διεκπεραιωτικότητα των εγγραφών από την στιγμή που μπορεί να οδηγήσει σε λιγότερες εγγραφές, ενώ μπορεί να μειώσει και την κατανάλωση του εύρους ζώνης του δικτύου αφού δεν θα χρειαστεί να στέλνονται τα διπλότυπα δεδομένα. Μπορεί επίσης να εφαρμοστεί σε δίσκους SSD ώστε να μειώσει τις εγγραφές και να αυξήσει το προσδόκιμο ζωής τους.

Από την άλλη, διαταράσσει το σειριακό μοτίβο και την τοπικότητα της αποθήκευσης δεδομένων, που έρχεται σε σύγκρουση με πολλές βελτιστοποιήσεις που γίνονται ώστε αυτή να επιτευχθεί, μιας και ειδικά σε HDDs η σειριακή προσπέλαση των δεδομένων είναι τάξεις μεγέθους ταχύτερη από την τυχαία. Επιπροσθέτως, η απαλοιφή

διπλοτύπων εισάγει μια επιπλέον επιβάρυνση στη διαδικασία μιας εγγραφής και ένα επίπεδο μετάφρασης μεταξύ της λογικής αφαίρεσης των τεμαχίων και της πραγματικής τοποθέτησής τους στο φυσικό μέσο. Αυτό το επιπλέον επίπεδο δεν κοστίζει μόνο σε όρους αποθηκευτικού χώρου αλλά υποβαθμίζει την απόδοση μιας λειτουργίας Ε/Ε και εισάγει πολυπλοκότητα.

Αρχικά θεωρούμε ότι το σύστημα αποθήκευσης μας προορίζεται για αποθήκευση λογικών τόμων (volumes), ώστε να επικεντρωθούμε στην σημασιολογία που ομαδοποιεί τα δεδομένα μας σε μια λογική οντότητα και να ανεξαρτητοποιηθούμε από τα φυσικά μέσα αποθήκευσης. Ένας τόμος μπορεί να αποτελεί ένα αντίγραφο ασφαλείας, έναν εικονικό δίσκο, ένα στιγμιότυπο ενός συστήματος αρχείων, και γενικότερα οποιαδήποτε λογικά συναφή συλλογή δεδομένων αξιόλογου μεγέθους που εσωτερικά μπορεί να περιέχει αρχεία.
Υπάρχουν δύο είδη απαλοιφής διπλοτύπων για τα δεδομένα που αποθηκεύονται στο σύστημά μας. Η εσωτερική απαλοιφή διπλοτύπων (intra-deduplication), που εκμεταλλεύεται τον πλεονασμό ανάμεσα σε δεδομένα του ίδιου τόμου, και η εξωτερική απαλοιφή διπλοτύπων (inter-deduplication), που εκμεταλλεύεται τον πλεονασμό ανάμεσα σε διαφορετικούς τόμους. Η εσωτερική απαλοιφή διπλοτύπων, μπορεί να υποδιαιρεθεί σε χρονική και χωρική. Στην χρονική, απαλείφεται ο πλεονασμός ανάμεσα σε δυο διαφορετικές χρονικές εκδόσεις του τόμου, δηλαδή δυο στιγμιότυπα του τόμου που αναφέρονται στα περιεχόμενά του σε δυο διακριτές χρονικές στιγμές. Στη χωρική, εξαλείφεται ο πλεονασμός ανάμεσα σε κομμάτια δεδομένων (τεμάχια) της ίδιας χρονικής έκδοσης. Τέλος, ένα σύστημα αποθήκευσης μπορεί να συνδυάσει τον εσωτερικό με τον εξωτερικό πλεονασμό (inter-intra deduplication) ώστε να εκμεταλλευτεί όλους τους παραπάνω τύπους πλεονασμού για την δεξαμενή τομέων που διαθέτει.

Η αποτελεσματικότητα της απαλοιφής διπλοτύπων ποσοτικοποιείται από τον αντίστοιχο λόγο (deduplication ratio) ο οποίος ορίζεται ως το κέρδος σε αποθηκευτικό χώρο, δηλαδή η ποσότητα των διπλοτύπων που εξαλείφονται προς το συνολικό μέγεθος των δεδομένων που θα αποθηκεύονταν χωρίς την χρήση της τεχνικής.

$$\text{deduplication ratio} = 1 - \frac{\text{bytes stored after deduplication}}{\text{bytes stored originally}}$$

Πηγές του πλεονασμού

Στα εικονικοποιημένα περιβάλλοντα υπολογιστικού νέφους, υπάρχουν πολλές πηγές για πλεονασμό μεταξύ των εικόνων (images) των VMs. Αρχικά υπάρχουν εικόνες με τα ίδια ή παρεμφερή λειτουργικά συστήματα και από αυτά υπάρχουν πολλά που περιέχουν τις ίδιες ή παρεμφερείς εφαρμογές. Ακόμα και οι ιδιωτικές εικόνες των χρηστών είναι συνήθως ελαφρώς παραλλαγμένες δημόσιες εικόνες με κωδικούς, κλειδιά ασφαλείας και αρχεία ρυθμίσεων, ενώ συνήθως οι εικονικές μηχανές κλωνοποιούνται από ένα μικρό αριθμό προεπιλεγμένων εικόνων και έκτοτε δεν αποκλίνουν πολύ. Επίσης, υπάρχει όφελος από τις ενημερώσεις λογισμικού (software patches), ενώ πλεονασμός μπορεί να ανιχνευτεί ακόμα και ανάμεσα σε ίδια αρχεία ανεξάρτητων χρηστών μιας υποδομής υπολογιστικού νέφους, όπως πολυμέσα και δημοφιλή αρχεία στο Internet. Σύμφωνα με το [JM09], κατάλληλοι υποψήφιοι για απαλοιφή διπλοτύπων είναι παρόμοιες εκδόσεις πυρήνα, συνεχόμενες ή μη εκδόσεις ενός λειτουργικού συστήματος (distributions), διαφορετικές μορφοποιήσεις του ίδιου εικονικού δίσκου για διαχείριση από διαφορετικούς hypervisors και πακέτα λογισμικού με πολλές εξαρτήσεις. Τα κέρδη από απαλοιφή του πλεονασμού μπορούν να αυξηθούν με την χρήση έξυπνων μεθόδων που μπορούν να προβλέψουν τα σημεία ύπαρξης πλεονασμού ανάλογα με τις επεκτάσεις των αρχείων και το κομμάτι του συστήματος αρχείων. Ως δείγμα του μείζονος ρόλου που κατέχει η απαλοιφή διπλοτύπων σε περιβάλλοντα υπολογιστικού νέφους, αναφέρουμε ότι ο χώρος που εξοικονομείται βελτιώνεται σχεδόν γραμμικά σε σχέση με την λογαριθμική αύξηση των τόμων που προστίθενται [JPZ+11]. Μάλιστα η συνάθροιση πολλών συστημάτων αρχείων σε μια ενιαία δεξαμενή για απαλοιφή διπλοτύπων, προσφέρει περισσότερα οφέλη απ' ό,τι η κατάλληλη ρύθμιση των παραμέτρων της διαδικασίας της απαλοιφής, όπως είναι το μέγεθος του τεμαχίου ή ο αλγόριθμος τεμαχισμού [MB12].

Βασική διαδικασία

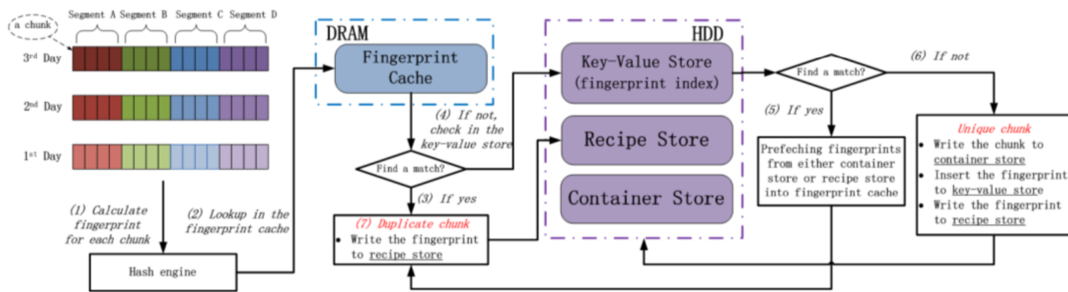
Τώρα θα διατυπώσουμε τα βασικά βήματα μια κλασσικής διαδικασίας απαλοιφής διπλοτύπων (base process), παρουσιάζοντας παράλληλα τα κύρια αρχιτεκτονικά συστατικά ενός συστήματος αποθήκευσης που καλείται να την υποστηρίζει.

Υποθέτουμε ότι έχουμε ένα σύστημα που δέχεται αιτήσεις για αποθήκευση μιας έκδοσης ενός τόμου. Αρχικά τα δεδομένα του τόμου χωρίζονται σε μη επικαλυπτόμενα τεμάχια (chunks) σταθερού ή μεταβλητού μεγέθους. Στη συνέχεια, υπολογίζεται το

αποτύπωμα (fingerprint) κάθε τεμαχίου, χρησιμοποιώντας μια κρυπτογραφική σύνοψη (cryptographic digest). Ένα αποτύπωμα λοιπόν είναι μια τιμή κατακερματισμού (hash) των δεδομένων του τεμαχίου, για παράδειγμα με βάση τους αλγορίθμους κατακερματισμού SHA-1 ή MD5, και αποτελεί το μοναδικό αναγνωριστικό του τεμαχίου. Δυο τεμάχια με πανομοιότυπα αποτυπώματα θεωρούνται διπλότυπα, χωρίς να χρειαστεί μια byte προς byte σύγκριση των περιεχομένων τους. Η πιθανότητα συγκρούσεων (hash collisions) είναι αρκετά μικρότερη από αυτήν των αστοχιών υλικού, οπότε σε πραγματικά συστήματα θεωρείται αποδεκτή.
Ένα ευρετήριο αποτυπωμάτων (fingerprint index) αντιστοιχεί τα αποτυπώματα των αποθηκευμένων τεμαχίων στις φυσικές διευθύνσεις τους. Συνεπώς, μόλις ένα τεμάχιο αποκτήσει το αποτύπωμά του, πραγματοποιείται μια αναζήτηση σε αυτό το ευρετήριο ώστε να διαπιστωθεί εάν το τεμάχιο είναι διπλότυπο και είναι ήδη αποθηκευμένο στο σύστημα. Όσα τεμάχια έχουν μοναδικά αποτυπώματα που δεν υπάρχουν στο ευρετήριο, πρέπει να αποθηκευτούν. Για αυτό το σκοπό τοποθετούνται σε οργανωμένες περιοχές του δίσκου που ονομάζονται containers και οι οποίες έχουν σταθερό μέγεθος (τυπικά 4MB) και ακολουθούν ένα log μοντέλο, όπου τα δεδομένα τοποθετούνται σειριακά μέχρι να γεμίσει το container. Στη συνέχεια, το αποτύπωμα μαζί με την διεύθυνση του container και το offset του τεμαχίου σε αυτό, προστίθενται στο ευρετήριο αποτυπωμάτων.
Ο τόμος όμως πρέπει να είναι ανακτήσιμος, οπότε για να είναι δυνατή η ανακατασκευή του από τα δεδομένα των τεμαχίων που τον αποτελούσαν, υπάρχει ένα αρχείο συνταγή (recipe), το οποίο περιέχει την ακολουθία των αποτυπωμάτων των τεμαχίων που συνθέτουν τον τόμο. Επομένως, είτε ένα τεμάχιο είναι διπλότυπο είτε όχι, το αποτύπωμά του αποθηκεύεται στην συνταγή.

Ως βελτιστοποίηση της παραπάνω διαδικασίας, μπορεί να χρησιμοποιηθεί και μια κρυφή μνήμη (fingerprint cache) όπου θα φορτώνονται προσωρινά εγγραφές από το ευρετήριο αποτυπωμάτων, ώστε να αποφευχθούν χρονοβόρες προσπελάσεις στο δίσκο, μιας και το κανονικό ευρετήριο αποτυπωμάτων είναι συνήθως πολύ μεγάλο για να αποθηκευτεί στην RAM. Μόλις βρεθεί ένα διπλότυπο, φορτώνονται κάποιες εγγραφές από το ευρετήριο αποτυπωμάτων, με την ελπίδα ότι τα νέα τεμάχια θα τις χρησιμοποιήσουν και έτσι θα επιταχυνθούν οι μελλοντικές προσπελάσεις. Η ποσότητα και το είδος των εγγραφών που φορτώνονται αναλύονται στη συνέχεια.

Για να ανακτηθεί ολικώς ή μερικώς ο τόμος, μπορούμε να συμβουλευτούμε την συ-



Σχήμα 3.5: *Base deduplication process*

νταγή που θα μας προσφέρει τα κατάλληλα αποτυπώματα και στη συνέχεια να ψάξουμε στο ευρετήριο αποτυπωμάτων ώστε να βρούμε τις φυσικές διευθύνσεις των containers και να ανακτήσουμε τα πραγματικά δεδομένα. Η παραπάνω διαδικασία, συνοψίζεται στο σχήμα 3.5 και ευθυγραμμίζεται με το πρότυπο του Content Adressable Storage που περιγράφτηκε στην 2.1.4.

Όπως προκύπτει από τα παραπάνω, τα βασικά αρχιτεκτονικά συστατικά είναι τα εξής:

- container store: η περιοχή του φυσικού δίσκου όπου αποθηκεύονται τα τεμάχια
- fingerprint index: ευρετήριο για αναζήτηση υπαρχόντων τεμαχίων με βάση τα αποτυπώματά τους και επίσης δομή αντιστοίχισης από τα αποτυπώματα στα container. Μπορεί να αναφερθεί επίσης ως και key-value store όπου το κλειδί είναι το αποτύπωμα και η τιμή η φυσική διεύθυνση
- recipe store: μια αποθήκη που για κάθε τόμο (και έκδοσή του) περιέχει την ακολουθία των αποτυπωμάτων των τεμαχίων που τον συνθέτουν
- fingerprint cache: μια κρυφή μνήμη στην οποία φορτώνονται προσωρινά εγγραφές του fingerprint index και προσπαθεί να εκμεταλλευτεί μοτίβα προσπελάσεων ώστε να αποφευχθούν κοστοβόρες αναζητήσεις στο fingerprint index.

Από τα παραπάνω συστατικά στοιχεία, το container store και το recipe store αποθηκεύονται στον δίσκο. Το ευρετήριο αποτυπωμάτων είναι συνήθως υπερβολικά μεγάλο για την RAM, οπότε μεταφέρεται και αυτό στο δίσκο, ενώ η fingerprint cache υπάρχει στην μνήμη. Το container store περιέχει τα δεδομένα του τόμου, ενώ τα fingerprint index και recipe store περιέχουν τα μεταδεδομένα που είναι απαραίτητα στο σύστημα απαλοιφής διπλοτύπων.

Παράμετροι απαλοιφής διπλοτύπων

Μπορούμε να καταρτίσουμε μια λίστα με τις βασικές παραμέτρους της διαδικασίας, βασιζόμενοι στις κύριες σχεδιαστικές επιλογές που καλείται να κάνει ένα σύστημα απαλοιφής διπλοτύπων και είναι κοινές για όλα αυτά τα συστήματα [PP12]. Αυτές συνοπτικά είναι οι εξής:

- μέγεθος των τεμαχίων
- αλγόριθμος τεμαχισμού (granularity): ολόκληρα αρχεία, μπλοκ σταθερού μεγέθους, μπλοκ μεταβλητού μεγέθους
- τοποθεσία τεμαχισμού και κατακερματισμού (place of chunking & hashing): στην πλευρά του πελάτη ή σε αυτή του εξυπηρετητή
- inline vs offline (χρόνος που πραγματοποιείται η απαλοιφή): κατά την διάρκεια της εγγραφής των δεδομένων ή σαν διεργασία παρασκηνίου
- πιστοποίηση της ισότητας ανάμεσα στο αποθηκευμένο διπλότυπο και στα δεδομένα του τεμαχίου: αν πραγματοποιηθεί ή οχι πλήρης σύγκριση των περιεχομένων τους ώστε να αποτραπούν συγκρούσεις hash
- ακρίβεια της ανίχνευσης διπλότυπων (indexing): αξιόπιστη εύρεση όλων των διπλοτύπων ή προσεγγιστική τεχνική που ανταλλάσσει ακρίβεια για αυξημένη απόδοση (fingerprint index: exact or near-exact)
- πολιτική προφόρτωσης fingerprint cache (prefetching policy): λογική τοπικότητα ή φυσική τοπικότητα
- εύρος της απαλοιφής διπλοτύπων: ολόκληρο το σύστημα (global deduplication), ή περιορισμός στα δεδομένα ενός συγκεκριμένου κόμβου (local deduplication)
- επίπεδο ασφάλειας: (καθόλου κρυπτογράφηση, κρυπτογράφηση ανά χρήστη, convergent κρυπτογράφηση)

Στη συνέχεια θα εξετάσουμε σύντομα τις επιδράσεις κάθε μιας από αυτές τις παραμέτρους σε ένα σύστημα απαλοιφής διπλοτύπων. Μπορούμε να αξιολογήσουμε αυτές τις επιδράσεις με βάση τα εξής κριτήρια: εξοικονόμηση αποθήκευτικού χώρου, απόδοση λειτουργιών Ε/Ε (ο χρόνος που απαιτείται ώστε να εκπληρωθεί κατά μέσο όρο

μια τέτοια λειτουργία), χρόνος ανάκτησης του τόμου (restore time), χρόνος που καταναλώνει ο επεξεργαστής, ανάγκες μνήμης, κίνηση δικτύου και διασφάλιση ιδιωτικότητας.

Μέγεθος τεμαχίου

Μικρότερο μέγεθος τεμαχίου οδηγεί σε πιο αποδοτική απαλοιφή διπλοτύπων. Όσο μικρότερο είναι το μέγεθος, τόσο αυξάνεται το επίπεδο λεπτομέρειας και άρα τόσο αυξάνονται οι πιθανότητες να εμφανιστούν όμοιες ακολουθίες δεδομένων. Όταν υπάρχουν μεγάλες ακολουθίες από αμετάβλητα δεδομένα μεταξύ δυο εκδόσεων ενός τόμου, τότε το μικρότερο μέγεθος δεν θα επιδράσει σημαντικά στην απόδοση, αφού ο πλεονασμός θα είναι ανιχνεύσιμος και από το μεγαλύτερο μέγεθος. Όταν συχνές αλλαγές συμβαίνουν σε περιοχές μικρότερες από το μέγεθος του μικρότερου τεμαχίου, πάλι δεν θα υπάρχει διαφορά, αφού και τα δύο μεγέθη θα αδυνατούν να ανακαλύψουν τον πλεονασμό. Όταν όμως οι αλλαγές είναι σποραδικές ή συμβαίνουν ανά ένα συγκεκριμένο αριθμό δεδομένων, τότε το μικρότερο μέγεθος θα βοηθήσει να απομονωθούν τα κομμάτια που άλλαξαν από τα κομμάτια που παρέμειναν αμετάβλητα. Επίσης, ένα μικρότερο τεμάχιο χρησιμοποιεί αποδοτικότερα το εύρος ζώνης του δικτύου, όταν αυτό αποστέλλεται στο σύστημα από κάποιον απομακρυσμένο κόμβο.
Από την άλλη, μικρότερα τεμάχια συνεπάγονται και περισσότερα αποτυπώματα για την ίδια ποσότητα δεδομένων, οπότε όχι μόνο έχουμε μεγαλύτερη επιβάρυνση από μεταδεδομένα καθώς θα έχουμε μεγαλύτερες συνταγές και μεγαλύτερα ευρετήρια αποτυπωμάτων, αλλά το τελευταίο συνεπάγεται και αύξηση του χρόνου αναζήτησης στο ευρετήριο και άρα μείωση της απόδοσης μιας λειτουργίας Ε/Ε. Επιπροσθέτως, μικρότερα τεμάχια περιορίζουν τις δυνατότητες για αποδοτική συμπίεση (compression) μετά την απαλοιφή των διπλότυπων και την αποθήκευση του τεμαχίου, ενώ μπορεί να οδηγήσουν και σε αυξημένο κατακερματισμό (fragmentation) καθώς διαταράσσεται ακόμα εντονότερα η τοπικότητα των δεδομένων του αρχικού τόμου.
Αποτελέσματα ερευνών πάνω στο θέμα [NKO$^+$06],[JM09],[JPZ$^+$11],[MB12],[WDQ$^+$12] και οι επιλογές διαφόρων ανεπτυγμένων συστημάτων απαλοιφής διπλοτύπων όπως παρουσιάζονται στον πίνακα 3.1, συγκλίνουν σε μια περιοχή μεγέθους μεταξύ 2KB και 16KB, εάν θέλουμε να έχουμε αποδοτική απαλοιφή διπλοτύπων. Τα οφέλη εξοικονόμησης χώρου για αυτά τα μεγέθη συνήθως υποσκελίζουν την επιβάρυνση από τα μεταδεδομένα και είναι πιο σημαντικά από οφέλη παραδοσιακής συμπίεσης. Επίσης συμφωνούν στο ότι το deduplication ratio μειώνεται γρήγορα με την αύξηση του μεγέ-

θους του τεμαχίου. Παρόλα αυτά ακόμα και με μεγαλύτερα μεγέθη τεμαχίων, όπως τα 128KB, είναι δυνατόν να εξοικονομηθεί ένα αξιόλογο κομμάτι αποθηκευτικού χώρου, ενώ εάν συνυπολογιστεί και η επιβάρυνση σε μεταδεδομένα στο deduplication ratio, υπάρχουν περιπτώσεις που τεμάχια 32KB αποδίδουν εξίσου καλά ή και καλύτερα από τα μικρότερά τους [SKM+16].

Αλγόριθμος τεμαχισμού

Υπάρχουν τρεις βασικές προσεγγίσεις: τεμαχισμός σε επίπεδο αρχείων (whole file chunking), τεμαχισμός σταθερού μεγέθους (fixed sized chunking) και τεμαχισμός μεταβλητού μεγέθους (variable sized chunking ή content defined chunking). Στην πρώτη προσέγγιση, τα τεμάχια συμπίπτουν με τα αρχεία που υπάρχουν στον τόμο, και είναι εφικτή μόνο στην περίπτωση που ο τόμος αυτός έχει κάποιο σύστημα αρχείων. Στον τεμαχισμό σταθερού μεγέθους, τα δεδομένα του τόμου διαιρούνται σε τεμάχια με σταθερά σύνορα, δηλαδή ένα νέο τεμάχιο ορίζεται για παράδειγμα κάθε 4KB. Στον μεταβλητό τεμαχισμό, η επιλογή των τεμαχίων δεν βασίζεται σε κάποιο συγκεκριμένο offset αλλά σε ιδιότητες των δεδομένων. Ο αλγόριθμος λειτουργεί ολισθαίνοντας ένα παράθυρο N-byte πάνω από τα δεδομένα και υπολογίζοντας το Rabin Fingerprint [Rab81] για κάθε παράθυρο, που δεν είναι τίποτα άλλο παρά μια τιμή κατακερματισμού. Όταν τα χαμηλότερα M bits της τιμής του Rabin Fingerprint είναι μηδενικά, το σύστημα θεωρεί αυτά τα N bytes ως σημείο διάσπασης, ολοκληρώνει τα όρια του τρέχοντος τεμαχίου, και από το επόμενο byte αρχίζει την ίδια διαδικασία για την εύρεση των ορίων του επόμενου τεμαχίου. Από την στιγμή που η έξοδος του Rabin Fingerprint είναι ψευδο-τυχαία, η πιθανότητα κάποιων N bytes να αποτελούν όριο διάσπασης είναι 2^{-M}. Θα μπορούσε να χρησιμοποιηθεί οποιαδήποτε συνάρτηση κατακερματισμού, αλλά το Rabin Fingerprint αποτελεί αποδοτική επιλογή κατά την διαδικασία της ολίσθησης που περιγράψαμε, αφού ο υπολογισμός της τιμής κατακερματισμού για μια περιοχή B μπορεί να επαναχρησιμοποιήσει κάποιους από τους υπολογισμούς που έγιναν για μια περιοχή A, με την οποία η B επικαλύπτεται.

Η προσέγγιση του μεταβλητού τεμαχισμού είναι συνήθως πιο αποδοτική από πλευράς εξοικονόμησης αποθηκευτικού χώρου αφού, αντιμετωπίζει το πρόβλημα ολίσθησης συνόρων (boundary shifting problem). Στον σταθερό τεμαχισμό, μια απλή εισαγωγή bytes στην αρχή της ροής των δεδομένων θα προκαλέσει την αλλαγή των περιεχομένων κάθε επόμενου τεμαχίου και άρα θα μεταβληθούν όλα τα αποτυπώματα των τεμαχίων. Αντίθετα, στον μεταβλητό τεμαχισμό, τα όρια των τεμαχίων δεν είναι στα-

θερά αλλά εξαρτώνται από μοτίβα του περιεχομένου, και έτσι η εισαγωγή των δεδομένων πιθανότατα δεν θα αλλάξει το μοτίβο διάσπασης πολλών τεμαχίων. Από την άλλη ο μεταβλητός τεμαχισμός απαιτεί επιπλέον υπολογισμούς για τις τιμές κατακερματισμού και καταναλώνει δυνητικά πολύτιμους πόρους της μνήμης και του επεξεργαστή, και υποβαθμίζει την απόδοση εάν πραγματοποιείται στο μονοπάτι εγγραφής (write path). Ενώ οι περισσότερες έρευνες συμφωνούν ότι ο μεταβλητός τεμαχισμός φέρνει καλύτερα αποτελέσματα, υπάρχουν αναφορές σύμφωνα με τις οποίες ο σταθερός τεμαχισμός είναι εφικτό να ανιχνεύσει ένα σημαντικό βαθμό πλεονασμού [JM09] [JPZ$^+$11].

Τοποθεσία τεμαχισμού και κατακερματισμού

Σε περίπτωση που για την απαλοιφή διπλοτύπων έχουμε πελάτες που στέλνουν τα δεδομένα στον deduplication εξυπηρετητή μέσω ενός δικτύου, τότε ο τεμαχισμός και ο υπολογισμός των τιμών κατακερματισμού μπορούν να πραγματοποιηθούν είτε πριν την αποστολή, δηλαδή στον πελάτη, είτε μετά την αποστολή, δηλαδή στον εξυπηρετητή. Εάν τεμαχιστούν πριν την μεταφορά τους στο δίκτου τότε μπορούν αρχικά να σταλούν μόνο τα αποτυπώματά τους στον εξυπηρετητή, ώστε αυτός να ελέγξει εάν αποτελούν διπλότυπα και αν ναι, να μην χρειαστεί να αποσταλούν τα δεδομένα του τεμαχίου, αφού αυτά θα είναι ήδη αποθηκευμένα. Προφανώς, ο τεμαχισμός στον πελάτη μπορεί να εξοικονομήσει κίνηση στο δίκτυο και πόρους του εξυπηρετητή αλλά είναι επιβαρυντικός για τον πελάτη και για κάποιες εφαρμογές αυτή η επιβάρυνση ενδέχεται να μην είναι αποδεκτή [MCM01].

Inline ή offline

Εδώ προσδιορίζεται ο χρόνος στον οποίο πραγματοποιείται η ανίχνευση και η απαλοιφή των διπλότυπων δεδομένων. Στην inline προσέγγιση, η απαλοιφή πραγματοποιείται στο κρίσμο μονοπάτι εγγραφής (write critical path) και απαιτεί τον τεμαχισμό, τον υπολογισμό των τιμών κατακερματισμού, την αναζήτηση στο ευρετήριο αποτυπωμάτων, την αποθήκευση στο φυσικό δίσκο και την ενημέρωση της συνταγής να συμβούν πριν ολοκληρωθεί μια αίτηση εγγραφής. Στην offline προσέγγιση, η απαλοιφή διπλοτύπων διεξάγεται με την μορφή μιας διεργασίας παρασκηνίου και τα δεδομένα πρώτα αποθηκεύονται σε κάποια προσωρινή τοποθεσία στο δίσκο και υπόκεινται επεξεργασία από το σύστημα απαλοιφής κάποια πιο βολική στιγμή αργότερα και έξω από το κρίσιμο μονοπάτι εγγραφής.

Η inline προσέγγιση αυξάνει την διεκπεραιωτικότητα των εγγραφών (write throughput), καθώς οι εγγραφές για διπλότυπα τεμάχια δεν απαιτούν επιπλέον εγγραφές μετά την αποθήκευση του πρώτου τέτοιου τεμαχίου. Επίσης, απαιτείται έστω και προσωρινά λιγότερος αποθηκευτικός χώρος από την offline που πρέπει να τα τοποθετήσει σε ένα προσωρινό χώρο του δίσκου μέχρι να τρέξει η διεργασία παρασκηνίου, ενώ αποφεύγει και μετέπειτα αναγνώσεις από αυτόν τον προσωρινό χώρο. Επιπροσθέτως επειδή για την αξιοπιστία των δεδομένων χρησιμοποιούνται συχνά RAID τεχνικές, στην inline προσέγγιση, η εξάλειψη μερικών εγγραφών σημαίνει συνήθως και αποφυγή του επαναϋπολογισμού του parity και εξάλειψη των εγγραφών των parity bits. Επιπλέον, η inline μπορεί να συνδυαστεί κομψά με τον τεμαχισμό στον πελάτη ώστε να μειωθεί η κίνηση στο δίκτυο και η διεκπαραιωσιμότητα του δίσκου, κάτι που δεν είναι εφικτό στην online προσέγγιση, καθώς η αναζήτηση στο ευρετήριο δεν γίνεται ακαριαία.
Παρόλα αυτά η καθυστέρηση (latency) μιας αίτησης εγγραφής αυξάνεται καθώς όλη η επεξεργασία πρέπει να πραγματοποιηθεί στο μονοπάτι εγγραφής. Αυτή η αύξηση προκύπτει κυρίως εξαιτίας της αναζήτησης στο ευρετήριο αποτυπωμάτων και επειδή αυτή απαιτεί τυχαίες λειτουργίες Ε/Ε στον δίσκο, μπορεί να είναι μη αποδεκτή, ειδικά για εγγραφές σε πρωτεύουσα αποθήκευση. Υπάρχουν προτάσεις ώστε να μειωθεί αυτή η επίδραση με βελτιστοποιήσεις που εκμεταλλεύονται την τοπικότητα [ZLP08].
Η offline απαλοιφή διπλοτύπων, προφέρει ακόμη ευελιξία στην επιλογή μιας βολικής στιγμής για την διεξαγωγή της, όπου το σύστημα θα έχει χαμηλό φορτίο και δεν θα στερεί πολύτιμους πόρους από άλλες λειτουργίες. Τέλος σε file servers, τα περισσότερα αρχεία δεν προσπελάζονται 24 ώρες μετά την άφιξή τους, οπότε μια προσέγγιση που απαλείφει μόνο παλαιότερα αρχεία μπορεί να αποφύγει την αύξηση της καθυστέρησης και την επιβάρυνση από αλλεπάλληλες εγγραφές για τα συχνά προσπελάσιμα αρχεία του εξυπηρετητή [ESKK+12].

Ταυτοποίηση της ισότητας των περιεχομένων των τεμαχίων

Δυο τεμάχια μπορούν να θεωρηθούν ίσα απλώς με σύγκριση των αποτυπωμάτων τους, με εμπιστοσύνη στην χαμηλή πιθανότητα να υπάρξει μια σύγκρουση (hash collision) ή εναλλακτικά το σύστημα μπορεί να επιβάλει μια πλήρη byte προς byte σύγκριση των περιεχομένων τους ώστε να αποφευχθεί μια περίπτωση σύγκρουσης και συνάμα μια αλλοίωση δεδομένων. Τα περισσότερα συστήματα στην πραγματικότητα εμπιστεύονται την σύγκριση απλώς των τιμών κατακερματισμού των τεμαχίων, μιας που η πιθανότητα συγκρούσεων είναι πολύ μικρότερη από αυτήν των αστοχιών υλικού. Συ-

γκεκριμένα, η πιθανότητα συγκρούσεων για ένα 160-bit hash σε ένα αποθηκευτικό σύστημα κλίμακας exabyte είναι περίπου $7{\times}10^{-18}$. Συγκρούσεις έχουν αναφερθεί και παρόλο που η επιτηδευμένη πλαστογράφηση δύο τεμαχίων με ίδιο αποτύπωμα και διαφορετικά, νοηματικά ευσταθή δεδομένα αποτελεί κενό ασφαλείας, κάτι τέτοιο δεν είναι πρακτικά εφαρμόσιμο. Έτσι με την απλή σύγκριση αποτυπωμάτων, εξοικονομείται χρόνος του επεξεργαστή, δεν επηρεάζεται η απόδοση των λειτουργιών Ε/Ε, ειδικά για τις inline προσεγγίσεις, ενώ μόνο έτσι μπορεί να υποστηριχθεί και ο τεμαχισμός στην πλευρά του πελάτη ώστε να μειωθεί η κίνηση στο δίκτυο.

Ακρίβεια ανίχνευσης των τεμαχίων - Ευρετήριο αποτυπωμάτων

Το ευρετήριο αποτυπωμάτων μπορεί να χρησιμοποιηθεί είτε για exact deduplication, όπου όλα τα διπλότυπα τεμάχια εξαλείφονται ώστε να επιτύχουμε υψηλότερα deduplication ratio, είτε για near-exact deduplication όπου απαλείφονται μόνο κάποια από τα διπλότυπα τεμάχια, ώστε να μειώσουμε το μέγεθος του ευρετηρίου για να χωράει στην κύρια μνήμη και έτσι να επιτύχουμε καλύτερη απόδοση .

Στην exact προσέγγιση, για να αποτρέπονται, όσο είναι δυνατό, λειτουργίες Ε/Ε προς το πλήρες ευρετήριο που βρίσκεται στο δίσκο χρησιμοποιείται μια κρυφή μνήμη ευρετηρίου που προφορτώνει αποτυπώματα και συνήθως αυτή συνδυάζεται και με ένα Bloom filter στην κύρια μνήμη. Το Bloom filter είναι μια αποδοτική πιθανοτική δομή δεδομένων που αποφεύγει τις αναζητήσεις για κάποια από τα μη διπλότυπα τεμάχια [Blo70]. Ο κατακερματισμός (fragmentation) μπορεί να κάνει την κρυφή μνήμη μη αποδοτική και άρα να αυξήσει τις αναζητήσεις στο κύριο ευρετήριο με την πάροδο του χρόνου

Στην near-exact προσέγγιση, υπάρχει ένα μερικό ευρετήριο αποτυπωμάτων (partial fingerprint index) στο οποίο αποθηκεύονται με δειγματοληπτικό τρόπο κάποια μόνο αποτυπώματα (features). Έτσι κι εδώ είναι απαραίτητη μια κρυφή μνήμη ώστε να διατηρηθεί ένα υψηλό deduplication ratio, μιας και η μονάδα προφόρτωσης θα περιέχει αποτυπώματα που υπάρχουν στην συνταγή αλλά όχι και στο ευρετήριο αποτυπωμάτων. Αυτή η προσέγγιση γενικώς επιφέρει μια αύξηση στο συνολικό κόστος του συστήματος αλλά μπορεί να αποβεί χρήσιμη για μεγάλα συστήματα αποθήκευσης που θέλουν να μειώσουν την καθυστέρηση των εγγραφών (write latency) [FFH+15]

Πολιτική προφόρτωσης κρυφής μνήμης αποτυπωμάτων

Η κρυφή μνήμη αποτυπωμάτων (fingerprint cache) είναι απαραίτητη τόσο στο exact

deduplication για να μειώσει τις αναζητήσεις στο ευρετήριο αποτυπωμάτων όσο και στο near exact deduplication για να διατηρήσει ένα υψηλό deduplication ratio. Ανάλογα με την δομική μονάδα προφόρτωσης της κρυφής μνήμης, έχουμε εκμετάλλευση είτε της λογικής είτε της φυσικής τοπικότητας. Στην λογική τοπικότητα, προφορτώνεται μια ακολουθία αποτυπωμάτων από μια συνταγή, η οποία ονομάζεται και segment. Στην φυσική τοπικότητα προφορτώνεται μια ακολουθία αποτυπωμάτων από ένα container, η οποία αντικατοπτρίζει την φυσική διάταξη των τεμαχίων στο container. Καταλληλότερη για την πρόβλεψη των αποτυπωμάτων που ακολουθούν είναι η λογική τοπικότητα, και η φυσική την προσεγγίζει έως το σημείο που παρεισφρύει ο κατακερματισμός. Τότε η πολτική προφόρτωσης δεν καθίσταται αποδοτική, με αποτέλεσμα είτε να αυξάνονται οι αναζητήσεις στο ευρετήριο αποτυπωμάτων για exact deduplication, είτε να μειώνεται το deduplication ratio για near exact deduplication. Στην λογική τοπικότητα βέβαια, παρόλο που προφορτώνεται η πιο πρόσφατη σειρά των αποτυπωμάτων στον τόμο, δημιουργείται πρόβλημα με την επιλογή του κατάλληλου segment από ένα αποτύπωμα. Αυτό γιατί τα διπλότυπα τεμάχια έχουν το ίδιο αποτύπωμα αλλά ανήκουν εν γένει σε διαφορετικά segments, δηλαδή εμφανίζονται σε διαφορετικές θέσεις στην ακολουθία αποτυπωμάτων της συνταγής, οπότε η διαδικασία επιλογής του κατάλληλου segment από ένα αποτύπωμα είναι μη στοιχειώδης. Θα αποφύγουμε να αναλύσουμε με μεγαλύτερη λεπτομέρεια τον υποχώρο των δυνατών επιλογών της ακρίβειας ανίχνευσης και της πολιτικής προφόρτωσης, και για περισσότερα παραπέμπουμε στο [FFH+15]. Επιπλέον, υπάρχουν και οι κρυφές μνήμες που βασίζονται στην χρονική τοπικότητα και υλοποιούν μια πολιτική αντικατάστασης LRU, υποθέτοντας ότι ένα αποτύπωμα επανεμφανίζεται αρκετές φορές σε ένα περιορισμένο χρονικό παράθυρο.

Η βασική διαδικασία (base procedure) δεν αποδίδει καλά για εικονικούς δίσκους, όπου υπάρχει μεγάλη αυτοαναφορά ή αλλιώς εσωτερικός χωρικός πλεονασμός[FFH+15]. Αυτού του είδους ο πλεονασμός αποδυναμώνει τις παραπάνω πολιτικές προφόρτωσης, αφού ένα αποτύπωμα μπορεί να αναφέρεται σε πολλά segments και άρα αυξάνει την πιθανότητα να μην φορτωθεί το σωστό segment. Η λύση είναι ένας εναλλακτικός μηχανισμός απαλοιφής διπλοτύπων που ονομάζεται ανίχνευση ομοιότητας (similarity detection), κατά τον οποίο φορτώνεται ένας αριθμός από υποψήφια segments και όχι μόνο ένα για κάθε αποτύπωμα, ώστε να αυξηθούν οι πιθανότητες για cache hit.

Εύρος απαλοιφής διπλοτύπων

Για να είναι κλιμακώσιμο ένα σύστημα αποθήκευσης, συνήθως είναι κατανεμημένο σε διάφορους κόμβους, οπότε μπορεί να υπάρξει διαφοροποίηση στο εύρος των κόμβων που ανιχνεύονται τα διπλότυπα. Στο τοπικό deduplication κάθε κόμβος απαλείφει διπλότυπα ανεξάρτητα από τους άλλους, διατηρώντας δικό του ευρετήριο αποτυπωμάτων και συνεπώς όμοια τεμάχια μεταξύ δυο διαφορετικών κόμβων δεν εξαλείφονται. Κάποια συστήματα χρησιμοποιούν ευφυείς μηχανισμούς που αντιστοιχίζουν παρόμοια αρχεία ή παρόμοιες ομάδες τεμαχίων στον ίδιο κόμβο ώστε να αυξηθεί η συνολική εξοικονόμηση χώρου σε όλο το cluster. Στην καθολική προσέγγιση, τα διπλότυπα τεμάχια απαλείφονται καθολικά για όλο το cluster, αλλά αυτό απαιτεί ένα ευρετήριο που θα είναι προσβάσιμο από όλους τους κόμβους. Η καθολική απαλοιφή διπλοτύπων προσφέρει υψηλότερα κέρδη σε αποθηκευτικό χώρο αλλά η χρήση κεντρικοποιημένων ευρετηρίων παρουσιάζει προβλήματα κλιμακωσιμότητας και αντοχής σε σφάλματα (fault tolerance), ενώ άλλες λύσεις με αποκεντρικοποιημένα ευρετήρια οδηγούν σε αύξηση του χρόνου για αναζητήσεις και ενημερώσεις στο ευρετήριο. Επίσης υπάρχει και η κεντρικοποιημένη προσέγγιση, με συστήματα που είναι κατανεμημένα αλλά πραγματοποιούν απαλοιφή διπλοτύπων σε ένα μόνο κόμβο.

Ασφάλεια

Η πρώτη προσέγγιση είναι η αυστηρή κρυπτογράφηση με μια δεξαμενή τεμαχίων και ένα κλειδί ανά χρήστη. Αν κάποιος αποκτήσει πρόσβαση στα δεδομένα του δίσκου, ή ακόμη και στα αποτυπώματα των τεμαχίων, δεν θα μπορεί να τα διαβάσει αφού θα είναι κρυπτογραφημένα. Η πληροφορία του αποτυπώματος κάθε τεμαχίου δεν τον βοηθάει, αφού έχουμε μια μονόδρομη κρυπτογραφική συνάρτηση κατακερματισμού (one-way cryptographic hash function) που πρακτικά είναι αδύνατον να αναστραφεί. Σε αυτήν την περίπτωση, βέβαια, η απαλοιφή διπλοτύπων αναγκαστικά περιορίζεται μεταξύ των τεμαχίων κάθε χρήστη, δηλαδή εξαλείφεται η δυνατότητα για inter-deduplication, αφού ίδια τεμάχια διαφορετικών χρηστών, θα κρυπτογραφηθούν με διαφορετικό κλειδί και άρα τελικώς θα διαφέρουν.

Η εναλλακτική συμβιβαστική επιλογή, σε περίπτωση που απαιτείται ένας βαθμός ασφάλειας είναι η convergent encryption. Η κρυπτογράφηση κάθε τεμαχίου γίνεται με κλειδί την SHA-1 τιμή κατακερματισμού του τεμαχίου, οπότε ταυτόσημα τεμάχια παράγουν ταυτόσημα κρυπτο-τεμάχια. Στη συνέχεια πρέπει να μην είναι δυνατό ο εισβολέας να έχει πρόσβαση στο αποτύπωμα κάθε τεμαχίου. Γι᾽ αυτό η συνταγή κάθε χρήστη κρυ-

πτογραφείται με ένα ιδιωτικό κλειδί, ξεχωριστό για κάθε χρήστη. Αν κάποιος αποκτήσει πρόσβαση στα κρυπτογραφημένα δεδομένα του δίσκου, τότε δεν θα μπορεί να τα διαβάσει, αφού κάθε τεμάχιο θα είναι κρυπτογραφημένο με κλειδί την τιμή κατακερματισμού των δεδομένων του και θα είναι αδύνατο να βρει ποια είναι αυτή η τιμή. Αυτή η τακτική όμως είναι ευάλωτη σε επιθέσεις επιβεβαίωσης της πληροφορίας (confirmation of a file attack), όπου ο εισβολέας μπορεί να επιβεβαιώσει εάν ένα συγκεκριμένο τεμάχιο υπάρχει στο chunk store, όπως για παράδειγμα ένα αρχείο που παραβιάζει πνευματικά δικαιώματα. Επίσης είναι ευάλωτο στην επίθεση «εύρεσης των υπόλοιπων πληροφοριών» (learn the remaining information attack) όπου εάν υπάρχει ένα γενικό πρότυπο δημόσιου εγγράφου, με κάποια bits μόνο να διαφέρουν ανά περίπτωση, όντας αυτά που περιέχουν την χρήσιμη και μυστική πληροφορία, (π.χ. φόρμες τραπεζών), είναι εφικτό ο επιτιθέμενος να δοκιμάσει όλους τους συνδυασμούς των λίγων αυτών bits.
Καταλήγοντας, η convergent encryption παρέχει ασφάλεια ως προς τους εισβολείς που θέλουν το περιεχόμενο των δεδομένων ενός χρήστη αλλά όχι και σε αυτούς που που θέλουν να δουν εάν απλώς υπάρχουν κάποια δεδομένα ή σε ειδικές περιπτώσεις πρότυπων εγγράφων. Παρέχει όμως inter-deduplication που είναι πολύ σημαντικό σε περιβάλλοντα υπολογιστικού νέφους και επομένως η αυστηρή κρυπτογράφηση των δεδομένων ανά χρήστη αξίζει μόνο αν η απαίτηση για ασφάλεια είναι πολύ υψηλή. Ενδεικτικά η υλοποίηση αυστηρής κρυπτογράφησης απαιτεί 1.5 φορές των αποθηκευτικό χώρο ενός συστήματος που εφαρμόζει την πιο χαλαρή convergent encryption [NKO+06].

Πρωτεύουσα αποθήκευση

Αφού εξερευνήσαμε τον υποχώρο παραμέτρων μιας γενικής διαδικασίας απαλοιφής διπλοτύπων, μπορούμε να εστιάσουμε στην απαλοιφή διπλοτύπων για πρωτεύουσα αποθήκευση. Τα backup και archival συστήματα θεωρούν ότι τα δεδομένα είναι immutable, και ανταλλάσσουν latency για deduplication ratio και throughput, χρησιμοποιώντας κυρίως inline προσεγγίσεις. Το deduplication ratio τέτοιων συστημάτων κυμαινεται από 83% σε 90% [MB12]. Στην πρωτεύουσα αποθήκευση όμως, τα δεδομένα μπορούν να μεταβληθούν και η χαμηλή καθυστέρηση των λειτουργιών Ε/Ε είναι κρίσιμο ζήτημα, οπότε το ποσοστό των offline προσεγγίσεων αυξάνεται και το deduplication ratio είναι γύρω στο 68% [MB12]. Εάν επικεντρωθούμε σε γενικού σκοπού εικονι-

κούς δίσκους, για μεγάλα clusters το deduplication ratio μπορεί να αυξηθεί έως και 80%. [JM09].

Η παραπάνω διαφοροποίηση στα κέρδη αποθηκευτικού χώρου ανάμεσα στο backup και στο primary storage συμβαίνει διότι ο εγγενής πλεονασμός στο τελευταίο είναι χαμηλότερος απ' ότι στα backup φορτία. Διαφορετικές backup εκδόσεις του ίδιου τόμου παρουσιάζουν σημαντικότερο πλεονασμό συγκριτικά με τον χωρικό πλεονασμό ενός συστήματος αρχείων. Παρόλα αυτά, όπως προαναφέρουμε, σε περιβάλλοντα υπολογιστικού νέφους, οι εικονικοί δίσκοι μπορούν να αποτελέσουν μια σπουδαία πηγή πλεονασμού, αλλά πρέπει να λαμβάνουμε υπ' όψιν μας και τις αυστηρές απαιτήσεις για χαμηλό latency και το γεγονός ότι τα δεδομένα μπορούν να μεταβληθούν, χρησιμοποιώντας ίσως τεχνικές Copy-On-Write για να αποτρέψουμε ενημερώσεις σε μοιραζόμενα δεδομένα. Επίσης οι αιτήσεις ανάγνωσης είναι πολύ συχνές και πρέπει να εξυπηρετούνται πιο αποδοτικά απ' ότι οι αιτήσεις επαναφοράς (restore) του backup και archival storage.

Δεδομένης της ανάγκης για χαμηλή καθυστέρηση και μειωμένη επιβάρυνση στο κρίσιμο μονοπάτι εγγραφής, η offline προσέγγιση μοιάζει να είναι πιο βολική, αν και μπορεί να εισάγει επιπλέον προσωρινό χώρο και αναγνώσεις από αυτόν, να εγείρει ζητήματα ταυτοχρονισμού και να αυξάνει την πολυπλοκότητα του συστήματος. Έτσι, προκύπτουν και inline προσεγγίσεις με βελτιστοποιήσεις ώστε να μειωθεί το latency μιας αίτησης Ε/Ε. Τρόποι για να αντιμετωπίσουμε την υψηλή καθυστέρηση είναι οι SSDs, υλοποιήσεις απαλοιφής διπλοτύπων βασισμένες στο υλικό, RAM Bloof filters [Blo70], διαμοιρασμός των δεδομένων σε πολλούς δίσκους (disk stripping) που θα επιτρέψει παράλληλες αναζητήσεις, κρυφές μνήμες που εκμεταλλεύονται την τοπικότητα και ευρετήρια αποτυπωμάτων που δεν εφαρμόζουν exact deduplication ώστε να χωράνε στην RAM.
Για να αποφευχθεί το deduplication σε δεδομένα που μπορεί σύντομα να αλλάξουν ή κατά πάσα πιθανότητα δεν θα συνεισφέρουν σημαντικά σε εξοικόνομηση χώρου, είναι θεμιτό να χρησιμοποιηθούν κάποιες ευριστικές τεχνικές. Για παράδειγμα κρυπτογραφημένα δεδομένα και αρχεία πολυμέσων θα πρέπει να αποφεύγονται αφού δεν αποτελούν πηγές πλεονασμού, σε αντίθεση με αρχεία που δεν έχουν προσπελαστεί για κάποιο αριθμό ημερών. Φυσικά, δεδομένα που προσπελάζονται συχνά ή δεδομένα που έχουν μεγάλες απαιτήσεις απόδοσης, δεν είναι κατάλληλοι υποψήφιοι.

Πρέπει να τονιστεί ότι η αποθήκευση ενεργών συστημάτων αρχείων σε συστήματα απαλοιφής διπλοτύπων προσανατολισμένα για backup ή archival δεδομένα, όπως το Venti [QD02] ή το Hydra Store [DGH+09] οδηγούν σε φτωχή απόδοση για τις τυχαίες λειτουργίες Ε/Ε, που αποτελούν μάλλον και τον κύριο όγκο των λειτουργιών Ε/Ε. Αυτό συμβαίνει γιατί το throughput σε αυτά τα συστήματα έχει μεγαλύτερη προτεραιότητα από το latency, οπότε για σενάρια πρωτεύουσας αποθήκευσης, ένα αντιστοίχως προσανατολισμένο σύστημα πρέπει να αναπτυχθεί.

Απαλοιφή διπλοτύπων και συμπίεση

Πέρα από την απαλοιφή διπλοτύπων, η παραδοσιακή συμπίεση (compression) μπορεί να χρησιμοποιηθεί για την εξοικονόμηση αποθηκευτικού χώρου. Συνήθως, η συμπίεση είναι λιγότερο αποδοτική σε σχέση με το deduplication, αλλά οι δύο τεχνικές μπορούν να συνδυαστούν για να αποφέρουν ακόμα καλύτερα αποτελέσματα. Αυτό πραγματοποιείται με την συμπίεση των μοναδικών τεμαχίων που έχουν αποθηκευτεί στο chunk store, ανιχνεύοντας επαναληπτικές ακολουθίες δεδομένων που υπάρχουν μέσα σε αυτά. Η πιο δημοφιλής μέθοδος συμπίεσης είναι ο αλγόριθμος DEFLATE που χρησιμοποιείται στα αρχεία zip.
Εάν οι δύο αυτές τεχνικές συνδυαστούν, πιθανώς να απαιτείται επαναπροσδιορισμός του καταλληλότερο μεγέθους τεμαχίου, ώστε να αυξηθούν στο μέγιστο τα κέρδη σε αποθηκευτικό χώρο [NKO+06]. Ένα αξιοθάυμαστο αποτέλεσμα είναι ότι υψηλά κέρδη που επιτυγχάνονται με 4KB τεμάχια και μόνο απαλοιφή διπλοτύπων, είναι δυνατόν να επιτευχθούν και με μεγαλύτερα τεμάχια έως και 64KB, εάν κατόπιν συμπιέσουμε τα τεμάχια, μιας που η απώλεια στις ευκαιρίες ανίχνευσης πλεονασμού με το μεγαλύτερο μέγεθος τεμαχίου αντισταθμίζεται από την συμπίεση των τεμαχίων [ESKK+12].

Αγνοήσαμε στην ανάλυσή μας πτυχές όπως το reference-counting και η συλλογή σκουπιδιών (garbage collection) των τεμαχίων καθώς και αλγορίθμους επανεγγραφής (rewriting algorithms) που φροντίζουν να μειώνουν τον κατακερματισμό (fragmentation) που προκύπτει με την πάροδο του χρόνου.

Καταλήγουμε δίνοντας στον πίνακα 3.1 μια περίληψη των επιλογών που ακολουθούν διάφορα συστήματα απαλοιφής διπλοτύπων, με την βοήθεια του [PP12]. Τα πρώτα αφορούν δευτερεύουσα αποθήκευση (backup storage), και αυτά μετά την οριζόντια γραμμή αφορούν πρωτεύουσα αποθήκευση (primary storage).

Όνομα	Algorithm	Timing	Fing. Index	Locality	Scope
SIS	W	O	F	-	C
Farsite	W	O	F	-	G
Venti	F - 8KB	I	F	T	C
Foundation	F - 4KB	I	F	T	C
Guo & Efstathopoulos	F - 4KB	I	P	P	C
Jumbo Store	V	I	F	-	C
Debar, ChunkFarm	V - 8KB	O	F	-	G
HYDRAStore, HydraFS	V - 64KB	I	F	-	G
Kaiser et al. [2012]	V - 8KB	I	F	-	G
Pastiche	V - 16KB	I	F	-	L
DDFS, dedupv1	V - 8KB	I	F	P	C
Dong et al. [2011]	V - 8KB	I	F	P	L
ChunkStash	V - 8KB	I	F	L	C
SiLo	V	I	P	L	L
Σ-Dedup	V - 4KB	I	F	L	L
Sparse Indexing	V - 4KB	I	S	L	C
Mad2	WV	I	F	L	G
Extreme Binning	WV - 4KB	I	S	-	L
Deep Store	V	I	F	-	L
ZFS	F - block	I	F	-	C
DBLK	F - 4KB	I	F	-	C
DeDe	F - 4KB	O	F	-	G
DDE	F - block	O	F	-	G
LiveDFS	F - block	I	F	P	C
iDedup	F - block	I	F	TP	C
Windows Server 2012	V $\geq$ 32KB	O	F	-	C

Πίνακας 3.1: *Συστήματα απαλοιφής διλοτύπων και σχεδιαστικές επιλογές*
Algorithm - Αλγόριθμος: (W)hole-file, (F)ixed-size, (V)ariable-size
Χρόνος - Timing: I(nline), (O)ffline
Fing. Index - Ευρετήριο: (F)ull Index, (P)artial index, (S)imilarity detection
Locality - Τοπικότητα: (T)emporal, (L)ogical, (P)hysical
Scope - Εύρος: (C)entralized, (G)lobal, (L)ocal
Για variable-size παραθέτουμε το μέσο μέγεθος τεμαχίου. Η ένδειξη block υποννοεί το μέγεθος μπλοκ του συστήματος αρχείων (συνήθης προεπιλογή τα 4KB) ή ένα μέγεθος μπλοκ που αφήνεται στην ευχέρεια του χρήστη

3.2 Παράδειγμα μορφής εικονικού δίσκου: qcow2

Σε αυτήν την ενότητα, θα παρουσιάσουμε τα χαρακτηριστικά, τη δομή και τις λειτουργίες της qcow2 μορφής εικονικού δίσκου. Θα προσπαθήσουμε με αυτόν τον τρόπο να διαπιστώσουμε πως κάποιες από τις παραπάνω έννοιες που εισάγαμε στις προηγούμενες ενότητες υλοποιούνται πρακτικά. Επίσης θα προετοιμάσουμε το έδαφος για την μετέπειτα αιτιολόγηση του γιατί αυτή η ευρέως διαδεδομένη μορφή εικονικού δίσκου, παρά τα πλούσια χαρακτηριστικά της, είναι ελλιπής ως προς την εκπλήρωση των αρχικών στόχων μας.

Το qcow2 είναι ένα από τα υποστηριζόμενα «image formats» του QEMU. Αποτελεί την αναπαράσταση μιας μπλοκ συσκευής δεδομένου μεγέθους εν είδη αρχείου. Κάποια από τα πλεονεκτήματα χρήσης της είναι:

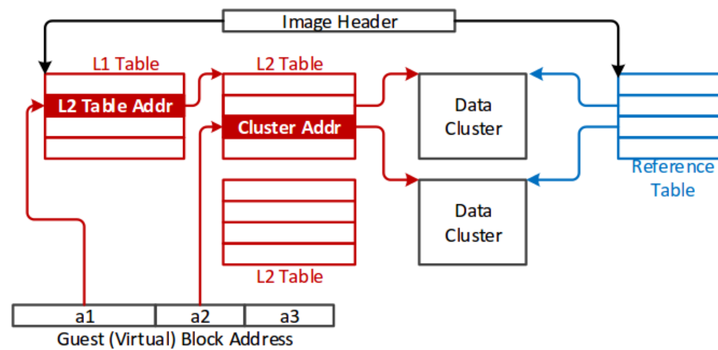
1. Μικρότερο μέγεθος αρχείου από το μέγεθος του δίσκου, ακόμα και για τα συστήματα αρχείων που δεν υποστηρίζουν sparse files

2. Υποστήριξη Copy-On-Write, όπου μια δευτερεύουσα εικόνα μπορεί να αναπαριστά τις αλλαγές που έχουν συμβεί σε έναν κυρίως εικονικό δίσκο. Νοηματικά συγγενεύει με τον κλώνο ενός εικονικού δίσκου.

3. Υποστήριξη στιγμιοτύπων, όπου μια εικόνα μπορεί να περιέχει πολλαπλά στιγμιότυπα του εικονικού δίσκου σε παρελθοντικές στιγμές

4. Προαιρετική zlib συμπίεση και AES κρυπτογράφηση

Ένας qcow2 εικονικός δίσκος είναι ένα αρχείο που περιέχει μια επικεφαλίδα, έναν δυοεπιπέδων πίνακα αναζήτησης και συμπλέγματα δεδομένων (data clusters) όπως φαίνεται στο σχήμα 3.6 [CLX+16]. Το αρχείο είναι οργανωμένο σε συμπλέγματα, που είναι η στοιχειώδης μονάδα δέσμευσης δεδομένων και μεταδεδομένων για τον εικονικό δίσκο. Κάθε cluster περιέχει ένα αριθμό από τομείς των 512 bytes. Η επικεφαλίδα μοιάζει με το superblock ενός συστήματος αρχείων, και περιέχει βασικές πληροφορίες για την εικόνα όπως οι διευθύνσεις του πίνακα αναζήτησης, του πίνακα αναφορών, του πίνακα στιγμιοτύπων και άλλα. Όπως και για τους μετρητές αναφορών, το qcow2 χρησιμοποιεί μια δομή δυο επιπέδων για την αντιστοίχηση των clusters του host με αυτών του guest. Τα δύο επίπεδα απαρτίζονται από έναν L1 και πολλούς L2 πίνακες. Ο



Σχήμα 3.6: *qcow2 format*

L1 πίνακας, έχει μεταβλητό μέγεθος (αποθηκευμένο στην επικεφαλίδα) και μπορεί να χρησιμοποιήσει πολλαπλά αλλά συνεχόμενα στο αρχείο clusters. Οι L2 πίνακες έχουν μέγεθος ακριβώς όσο ένα cluster. Ο πίνακας αναζήτησης, χρησιμοποιείται για την μετάφραση μια διεύθυνσης τομέα του guest, σε ένα offset αρχείου. Μια διεύθυνση στον guest χωρίζεται σε 3 μέρη έστω a_1, a_2, a_3. Το a_1 χρησιμοποιείται ως δείκτης στον L1 πίνακα, το a_2 ως δείκτης στον L2 πίνακα για την εύρεση του κατάλληλου data cluster και το a_3 είναι το offset εσωτερικά του cluster. Εάν ένα εκ των a_1 ή a_2 είναι 0, σημαίνει ότι το cluster που θα έπρεπε να δεικτοδοτείται δεν έχει δεσμευτεί ακόμα. Ο πίνακας αναφορών (reference table) χρησιμοποιείται για να παρακολουθείται κάθε cluster που χρησιμοποιείται από κάποιο στιγμιότυπο. Ο μετρητής αναφορών (reference counter) τίθεται σε 1 για κάθε cluster που δεσμεύεται και η τιμή του αυξάνεται όταν περισσότερα στιγμιότυπα χρησιμοποιήσουν αυτό το cluster. Μια τιμή 0, συνεπάγεται ότι το cluster είναι ελεύθερο, 1 σημαίνει ότι χρησιμοποιείται και ≥ 2 σημαίνει ότι χρησιμοποιείται από τουλάχιστον ένα στιγμιότυπο και κάθε εγγραφή στο cluster θα επιφέρει μια COW λειτουργία. Οι μετρητές αναφορών υπάρχουν και αυτοί σε μια δομή δυο επιπέδων, παρόλο που στο σχήμα 3.6 φαίνεται για απλοποίηση μόνο το ένα επίπεδο. Το πρώτο επίπεδο ονομάζεται reference table, και έχει μεταβλητό μέγεθος που εκτείνεται σε πολλαπλά συνεχόμενα clusters. Αυτό περιέχει δείκτες σε δομές δευτέρου επιπέδου, μεγέθους ενός cluster, που καλούνται refcount blocks και περιέχουν τους μετρητές αναφορών.

Τυπικά ένα αρχείο θα έχει την εξής διαρρύθμιση:

- Η επικεφαλίδα
- L1 πίνακας, ξεκινώντας από το επόμενο cluster
- Ο πίνακας αναφορών

- Ένα ή περισσότερα refcount blocks
- Επικεφαλίδες στιγμιοτύπων
- L2 πίνακες
- Data clusters

Copy-on-Write εικόνες

Μια εικόνα qcow2 μπορεί να χρησιμοποιηθεί για να αποθηκεύσει τις αλλαγές σε σχέση με μια άλλη εικόνα δίσκου, χωρίς να επηρεάσει τα περιεχόμενα της αρχικής εικόνας. Όλες οι αλλαγές καταγράφονται σε ένα ξεχωριστό αρχείο, διατηρώντας άθικτο το αρχικό αρχείο ή αλλιώς αρχείο υποστήριξης (backing file). Πολλαπλά COW αρχεία μπορεί να δείχνουν στην ίδια αρχική εικόνα, ώστε για παράδειγμα πολλές παραμετροποιήσεις να μπορούν ταυτόχρονα να ελεγχθούν, χωρίς να διακινδυνεύσουν την ακεραιότητα του συστήματος. Η νέα εικόνα, αποκαλούμενη και copy-on-write εικόνα, μοιάζει με μια ανεξάρτητη εικόνα στον χρήστη, με την διαφορά όμως, ότι τα περισσότερα δεδομένα της ανακτώνται από την αρχική εικόνα. Μόνο τα clusters που μεταβάλλονται, αποθηκεύονται στο copy-on-write αρχείο. Συνεπώς μια copy-on-write εικόνα μπορεί να παρομοιαστεί με ένα κλώνο. Το copy-on-write αρχείο περιέχει το μονοπάτι στο σύστημα αρχείων για τον αρχικό εικονικό δίσκο και η επικεφαλίδα του copy-on-write αρχείου περιέχει το offset του αρχείου που περιέχει αυτό το string μονοπατιού. Κάθε φορά που ένα cluster πρέπει να διαβαστεί από την copy-on-write εικόνα, πρώτα ελέγχεται εάν αυτό το cluster είναι δεσμευμένο μέσα στην copy-on-write εικόνα. Εάν ναι, διαβάζεται από εκεί, ενώ εάν όχι, τότε τα δεδομένα του cluster πρέπει να διαβαστούν από το αρχείο της αρχικής εικόνας.

Εσωτερικά στιγμιότυπα

Το qcow2 υποστηρίζει εσωτερικά στιγμιότυπα. Τα στιγμιότυπα συγγενεύουν με το χαρακτηριστικό των copy-on-write εικόνων, με την διαφορά ότι είναι εσωτερικά στην εικόνα και ότι τώρα είναι η αρχική εικόνα που μπορεί να δεχθεί αλλαγές και όχι τα στιγμιότυπα. Ένα στιγμιότυπο αντικατοπτρίζει τα περιεχόμενα του δίσκου σε μια παρελθοντική χρονική στιγμή. Ο αρχικός δίσκος παραμένει εγγράψιμος και, όταν τα δεδομένα του μεταβάλλονται, δημιουργείται ένα αντίγραφο των αρχικών δεδομένων για κάθε στιγμιότυπο που αναφέρεται σε αυτά. Η βασική αρχή λειτουργίας συνίσταται στην αλλαγή του πίνακα L1 για τον κανονικό δίσκο, έτσι ώστε ένα καινούργιο σύνολο από clusters να εκτίθενται στον guest. Κατά την δημιουργία ενός στιγμιοτύπου, ο πίνακας L1 αντιγράφεται και οι μετρητές αναφορών όλων των πινάκων L2 και

των data clusters που είναι προσπελάσιμα από αυτούς τους πίνακες πρέπει να αυξηθούν, έτσι ώστε κάθε νέο write να προκαλεί COW και να μην είναι ορατό από τα άλλα στιγμιότυπα. Με την φόρτωση ενός στιγμιοτύπου, ο νέος πίνακας L1 και όλοι οι πίνακες L2 που αναφέρονται σε αυτόν πρέπει να ανακατασκευαστούν από τον πίνακα αναφορών.

Οι copy-on-write εικόνες και τα εσωτερικά στιγμιότυπα λοιπόν εμπλέκουν δυο διαφορετικούς μηχανισμούς και ικανοποιούν δυο διαφορετικούς στόχους. Μια copy-on-write εικόνα αποτελεί έναν νέο εικονικό δίσκο, εξωτερικά ορατό από το σύστημα αρχείων του host, ο οποίος αποθηκεύει τις αλλαγές σε σχέση με μια αρχική εικόνα. Έτσι η αρχική εικόνα μπορεί να θεωρηθεί ως στιγμιότυπο και η COW ως κλώνος του. Αυτό το χαρακτηριστικό είναι ιδιαίτερα χρήσιμο σε ένα περιβάλλον υπολογιστικού νέφους, αφού μπορούν να κατασκευαστούν πολλά ανεξάρτητα αρχεία μια βασικής εικόνας, τα οποία μπορούν να τρέχουν παράλληλα από διαφορετικούς χρήστες. Στα εσωτερικά στιγμιότυπα, εσωτερικά μόνο τίθενται όλα τα δεσμευμένα cluster της εικόνας μόνο για ανάγνωση έτσι ώστε μια εγγραφή σε κάποιο από αυτά να προκαλέσει μια COW λειτουργία. Επομένως, η εικόνα παραμένει ο πρωτεύων δίσκος, με κάποια εσωτερικά στιγμιότυπα, ορατά και προσπελάσιμα μόνο μέσω της εικόνας του δίσκου.

Αλυσίδα αυξητικών στιγμιοτύπων

Μια εναλλακτική χρήση των COW images είναι ως αυξητικά στιγμιότυπα (incremental snapshots). Έστω ένα copy-on-write αρχείο s_1 που είναι εγγράψιμο και το οποίο βασίζεται σε ένα αρχικό αρχείο ή αρχείο υποστήριξης (backing file) s που μπορεί να θεωρηθεί και σαν ένα αρχικό στιγμιότυπο του εικονικού δίσκου. Είναι δυνατόν να δημιουργηθεί ένα δεύτερο copy-on-write αρχείο s_2 το οποίο τώρα θα έχει ως αρχείο υποστήριξής του το s_1 και άρα θα περιέχει όλες τις αλλαγές που συμβαίνουν στον εικονικό δίσκο που προκύπτει από τον συνδυασμό των s και s_1. Το s_1 μετατρέπεται από αρχείο εγγραφής των μεταβολών του ζωντανού δίσκου, σε ένα αυξητικό στιγμιότυπο του δίσκου, γιατί παύει να είναι εγγράψιμο και απλώς εμπερικλείει τις αλλαγές για τον εικονικό δίσκο, από τον χρόνο δημιουργίας του s_1, μέχρι το χρόνο δημιουργίας του s_2. Δημιουργείται λοιπόν εν γένει μια αλυσίδα στιγμιοτύπων, καθένα από τα οποία περιγράφει με αυξητικό τρόπο τις αλλαγές ανάμεσα στο προηγούμενο και στο επόμενο στιγμιότυπο. Μόνο το τελευταίο copy-on-write αρχείο αντιπροσωπεύει την ζωντανή εγγράψιμη εκδοχή του δίσκου, για την σύνθεση των δεδομένων του οποίου απαιτούνται δυνητικά όλα τα προηγούμενα αυξητικά στιγμιότυπα. Αν ένα cluster δεν

βρεθεί στο s_x αναζητείται αναδρομικά στο s_{x-1}, s_{x-2} κτλ, ίσως μέχρι και το αρχικό s. Αυτήν την προσέγγιση των αλυσιδωτών στιγμιοτύπων ακολουθεί και το μοντέλο του QEMU όταν δέχεται εντολή για λήψη online στιγμιοτύπου. Υπάρχει μέριμνα και για την συγχώνευση πολλών τέτοιων copy-on-write αρχείων σε ένα, το οποίο θα περιέχει όλες τις αλλαγές από το s_x μέχρι το s_{x+k}.

Τώρα θα περιγράψουμε τα βήματα κάποιων βασικών λειτουργικών.

Για την ανάγνωση δεδομένων από ένα qcow2 εικονικό αρχείο, δεδομένου ενός αριθμού τομέα, ακολουθούνται τα παρακάτω βήματα:

1. Ανάκτηση της διεύθυνσης του πίνακα L1, με την χρήση του l1_table_offset πεδίου που υπάρχει στην επικεφαλίδα
2. Χρήση των πρώτων a_1 bits του αριθμού τομέα, ως δείκτη στον L1 πίνακα, για την εύρεση της κατάλληλης εγγραφής που περιέχει την διεύθυνση ενός L2 πίνακα
3. Ανάκτηση της διεύθυνσης ενός L2 πίνακα
4. Χρήση των επόμενων a_2 bits του αριθμού τομέα, ως δείκτη στον L2 πίνακα, για την εύρεση της κατάλληλης εγγραφής που περιέχει τη διεύθυνση του cluster δεδομένων
5. Ανάκτηση της διεύθυνσης του cluster δεδομένων
6. Χρήση των υπόλοιπων a_3 bits του αριθμού τομέα, σαν offset εσωτερικό στο cluster δεδομένων

Αν υπάρχει ένα αρχείο υποστήριξης, τότε η παραπάνω διαδικασία πραγματοποιείται στην COW εικόνα και σε περίπτωση που κάποια διεύθυνση βρεθεί να έχει τιμή 0, τότε η διαδικασία επαναλαμβάνεται για το αρχείο υποστήριξης. Αν και πάλι κάποια διεύθυνση βρεθεί 0, τότε επιστρέφεται ένα cluster γεμάτο με μηδενικά.

Για την εγγραφή κάποιων δεδομένων σε ένα αριθμό τομέα που δεν έχει ξαναγραφτεί, ακολουθούνται τα παρακάτω βήματα:

1. Ανάκτηση της διεύθυνσης του πίνακα L1, με την χρήση του l1_table_offset πεδίου που υπάρχει στην επικεφαλίδα
2. Αναζήτηση στον L1 πίνακα, με βάση τα πρώτα a_1 bits, ώστε να ανακτηθεί η διεύθυνση του L2 πίνακα
    - Αν ο L2 πίνακας δεν είναι δεσμευμένος, δηλαδή η διεύθυνσή του έχει την τιμή 0, τότε ανακτάται μια διαθέσιμη διεύθυνση cluster, το cluster αρχικοποιείται

ως κενός L2 πίνακας και τίθεται ο αντίστοιχος μετρητής αναφορών στον πίνακα αναφορών σε 1.

- Αν ο L2 πίνακας είναι δεσμευμένος, δηλαδή η διεύθυνσή του δεν είναι μηδέν, αλλά ο μετρητής αναφορών έχει τιμή μεγαλύτερη του 1, τότε αυτός ο πίνακας είναι μοιραζόμενος και θα πρέπει να γίνει COW. Τότε ανακτάται μια διαθέσιμη διεύθυνσή cluster, το cluster αρχικοποιείται με τα περιεχόμενα του L2 πίνακα που είναι μοιραζόμενος και ο αντίστοιχος μετρητής αναφορών για το νέο cluster του L2 γίνεται 1.
- Αν ο L2 πίνακας είναι δεσμευμένος και έχει μετρητή αναφορών ίσο με 1, τότε απλώς κρατάμε την διεύθυνσή του

3. Σε αυτό το σημείο έχουμε την διεύθυνση ένα δεσμευμένου L2 πίνακα, με μία αναφορά σε αυτό. Εάν υπήρξε δέσμευση ενός νέου cluster, τότε η εγγραφή του L1 πίνακα, ενημερώνεται με την νέα διεύθυνση.

4. Αναζήτηση στον L2 πίνακα, με βάση τα επόμενα a_2 bits, ώστε να ανακτηθεί η διεύθυνση του cluster δεδομένων

    - Αν το cluster δεδομένων δεν είναι δεσμευμένο, δηλαδή η διεύθυνσή του έχει την τιμή 0, τότε ανακτάται μια διαθέσιμη διεύθυνση cluster, το cluster αρχικοποιείται με μηδενικά και τίθεται ο αντίστοιχος μετρητής αναφορών στον πίνακα αναφορών σε 1.
    - Αν το cluster δεδομένων δεν είναι δεσμευμένο, δηλαδή η διεύθυνσή του δεν είναι μηδέν, αλλά ο μετρητής αναφορών έχει τιμή μεγαλύτερη του 1, τότε αυτό το cluster είναι μοιραζόμενο και θα πρέπει να γίνει COW. Τότε ανακτάται μια διαθέσιμη διεύθυνσή cluster, το cluster αρχικοποιείται με τα περιεχόμενα του παλιού cluster δεδομένων και ο αντίστοιχος μετρητής αναφορών για το νέο cluster δεδομένων γίνεται 1.
    - Αν το cluster δεδομένων είναι δεσμευμένο και έχει μετρητή αναφορών ίσο με 1, τότε απλώς κρατάμε την διεύθυνσή του

5. Σε αυτό το σημείο έχουμε την διεύθυνση ένα δεσμευμένου cluster δεδομένων με μία αναφορά σε αυτό. Εάν υπήρξε δέσμευση ενός νέου cluster δεδομένων, τότε η εγγραφή του L2 πίνακα, ενημερώνεται με την νέα διεύθυνση.

6. Τα a_3 τελευταία bits του αριθμού διεύθυνσης χρησιμεύουν ως offset εσωτερικό του cluster δεδομένων, όπου και αυτά εγγράφονται

Προφανώς αν η αίτηση Ε/Ε εκτείνεται σε παραπάνω από ένα τομείς, τότε για κάθε τομέα εκτελούνται όσα από τα παραπάνω βήματα χρειάζονται, αφού, για παράδειγμα, για δυο συνεχόμενους τομείς δεν είναι ανάγκη να επαναληφθεί η διαδικασία από την αρχή, αλλά πιθανότατα μπορεί να επαναχρησιμοποιηθεί ο L2 πίνακας ή και το ίδιο cluster δεδομένων.

Υπάρχουν επίσης διάφορες επεκτάσεις του qcow2 προτύπου με bitmaps, για ευκολότερη δέσμευση clusters, αλλά βρίσκονται πέρα από την ανάλυσή μας. Περισσότερες λεπτομέρειες μπορούν να βρεθούν στα [McL08], [gd16a].

Εδώ χρησιμοποιήσαμε την μορφή qcow2 ως παράδειγμα μιας μορφής αποθήκευσης εικονικού δίσκου. Άλλες μορφές όπως οι vmdk, vdi και vhd ακολουθούν παρόμοια λογική και τα στιγμιότυπά τους συμβαδίζουν με το πρότυπο της αλυσίδας των αυξητικών στιγμιοτύπων. Ενδεικτικά η vmdk έχει clusters μεγέθους 64KB και υποστηρίζει μεταξύ άλλων μια διεπίπεδη δομή αντιστοίχισης, ενώ οι επόμενες δύο έχουν μονοδιάστατη δομή αντιστοίχισης και clusters μεγέθους 1MB και 2MB ή 512KB αντίστοιχα.

3.3 Παρουσίαση του προβλήματος και των δυσκολιών του

Αφού εξετάσαμε διάφορες τεχνικές εξοικονόμησης χώρου μαζί με τις σχεδιαστικές επιλογές που αυτές επιφέρουν, ήρθε η ώρα να τοποθετήσουμε το πρόβλημα σχεδιασμού μιας μορφής εικονικού δίσκου σε μια πιο στέρεα βάση, θέτοντας στόχους και εντοπίζοντας ελλείψεις και δυσκολίες.

Το πρόβλημα που προσπαθούμε να αντιμετωπίσουμε είναι η έλλειψη μιας μορφής εικονικού δίσκου, η οποία θα υποστηρίζει αποδοτικά σε χρόνο και χώρο λειτουργίες στιγμιοτύπων, κλώνων και θα μπορεί να συνεργαστεί αρμονικά με ένα σύστημα απαλοιφής διπλοτύπων. Τα χαρακτηριστικά που μας ενδιαφέρουν λοιπόν είναι:

- υποστήριξη sparseness
- στιγμιότυπα
- κλώνοι
- απαλοιφή διπλοτύπων
- λίγα μεταδεδομένα για την υποστήριξη όλων των παραπάνω

Τα παραπάνω χαρακτηριστικά συνοψίζουν τις βασικές απαιτήσεις ενός χωρικά αποδοτικού μοντέλου αποθήκευσης ενός εικονικού δίσκου, το οποίο δύναται να εξαλείφει και τον εξωτερικό και τον εσωτερικό πλεονασμό. Πέρα από τα παραπάνω θέλουμε να έχουμε ακαριαία δημιουργία στιγμιοτύπν και κλώνων αλλά και σχετικά υψηλή απόδοση, με χαμηλό latency, που σέβεται στις απαιτήσεις της πρωτογενούς αποθήκευσης. Επίσης, επιθυμητό είναι να μπορούμε να κάνουμε γρήγορο συγχρονισμό με ένα απομακρυσμένο αντίγραφο του δίσκου και να έχουμε γρήγορη δημιουργία εικονικών δίσκων. Ως περιβάλλον εφαρμογής των παραπάνω, έχουμε στο μυαλό μας ένα περιβάλλον υπολογιστικού νέφους, όπου σε κάθε φυσικό μηχάνημα τρέχουν πολλά VMs, αποθηκεύοντας τα δεδομένα τους σε εικονικούς δίσκους.

Εξετάζουμε τα παραπάνω χαρακτηριστικά πιο συγκεκριμένα, υπογραμμίζοντας τις δυσκολίες κάθε μιας από τις απαιτήσεις μας, χρησιμοποιώντας απλές σχεδιαστικές προσεγγίσεις.

Μια πρώτη ιδέα για την εξοικονόμηση χώρου είναι οι αραιοί εικονικοί δίσκοι δίσκοι (sparse virtual disks). Αυτό σημαίνει ότι όταν δημιουργείται ένας δίσκος 1GB και άρα δημιουργείται στον host ένα αρχείο 1GB, δεν θέλουμε να δεσμεύονται κατευθείαν όλα τα μπλοκ που απαιτούνται για την αποθήκευση 1GB, αλλά η διαδικασία αυτή γίνεται κατ᾽ απαίτηση (on demand) και σταδιακά, όσο ο guest γράφει δεδομένα στο δίσκο. Δηλαδή ο δίσκος είναι κατ᾽ αρχάς κενός και όσο ο χρήστης προσθέτει δεδομένα, τόσο το μέγεθος του αρχείου του εικονικού δίσκου αυξάνεται. Το πλεονέκτημα αυτής της σχεδίασης είναι ότι η δημιουργία του δίσκου είναι ακαριαία, αφού δεσμεύονται μόνο μεταδεδομένα του αρχείου στο σύστημα αρχείων του host, ενώ εξοικονομείται και χώρος όλο το χρονικό διάστημα που το μέγεθος του δίσκου είναι μικρότερο του 1GB. Το πρόβλημα με τα αραιά αρχεία είναι ότι δεν υποστηρίζονται εγγενώς από όλα τα συστήματα αρχείων, οπότε μια απλοϊκή υλοποίηση ενός εικονικού δίσκου που βασίζεται στη δυνατότητα του συστήματος αρχείων για δέσμευση μπλοκ μόνο κατ᾽ απαίτηση και συντήρηση της ψευδαίσθησης του sparseness διαφανώς με την παροχή μηδενικών για τις τρύπες, είναι άρρηκτα συνδεδεμένη με την επιλογή του κατάλληλου συστήματος αρχείων. Αυτό μπορεί να μην είναι πρόβλημα εάν ο όλος σχεδιασμός ευνοεί την χρησιμοποίηση ενός συστήματος αρχείων που υποστηρίζει αραιά αρχεία, αλλά δεν παύει να εισάγει περιορισμούς στην αρχιτεκτονική του συστήματος.

Όσο αφορά τα στιγμιότυπα και τους κλώνους, προαναφέραμε τους λόγους που απο-

τελούν αναπόσπαστο στοιχείο κάθε περιβάλλοντος εικονικοποίησης. Μια πολύ απλοϊκή υλοποίηση στιγμιοτύπων και κλώνων θα ήταν να παίρναμε ένα αρχείο εικονικού δίσκου και να το αντιγράφαμε. Το παραπάνω όμως θα ήταν και χωρικά και χρονικά ασύμφορο, αφού εμπλέκει την αντιγραφή συνήθως πολλών gigabyte, η οποία είναι μια αργή διαδικασία, ενώ ανάμεσα στα στιγμιότυπα θα υπήρχαν πολλά κοινά δεδομένα που θα αποθηκευόντουσαν πλεοναστικά, περισσότερες από μια φορές. Συνεπώς πρέπει να χρησιμοποιηθεί η τεχνική COW για την έξυπνη απαλοιφή του εσωτερικού χρονικού πλεονασμού ανάμεσα σε διαφορετικά στιγμιότυπα του δίσκου. Όπως είδαμε, ο κλασσικός τρόπος με το οποίο αυτό επιτυγχάνεται είναι με τον χωρισμό του εικονικού δίσκου σε κομμάτια και, κάθε φορά που λαμβάνεται ένα στιγμιότυπο, όλα τα κομμάτια θα πρέπει να μαρκάρονται ως μόνο για ανάγνωση. Όταν έχουμε αίτηση εγγραφής σε ένα κομμάτι που είναι μόνο για ανάγνωση, έχουμε COW και δημιουργία νέου κομματιού. Έτσι όλα τα δεδομένα μεταξύ του αυθεντικού εικονικού δίσκου και του στιγμιοτύπου του είναι αρχικά μοιραζόμενα μέχρι ο αυθεντικός εικονικός δίσκος να δεχτεί μια αίτηση για εγγραφή σε ένα κομμάτι. Τότε μόνο δημιουργείται ένα νέο κομμάτι με τα καινούργια δεδομένα, ενώ παραμένει επίσης αποθηκευμένο και το παλιό κομμάτι, ώστε να είναι προσπελάσιμα τα παλιά δεδομένα από το στιγμιότυπο. Έτσι κρατάμε ουσιαστικά μόνο τις διαφορές ανάμεσα στον δίσκο και τα στιγμιότυπά του, σε επίπεδο κομματιού (block-granularity). Αυτό όπως προείπαμε αναγκαστικά εισάγει ένα επίπεδο μεταδεδομένων τόσο για την παρακολούθηση του δικαιώματος εγγραφής, όσο και για την αντιστοίχιση μεταξύ των αριθμών τομέα του εικονικού δίσκου και αυτών των κομματιών. Αυτά τα μεταδεδομένα θα πρέπει να φροντίσουμε να είναι αρκούντως λίγα ώστε να μην κυριαρχούν της εξοικονόμησης δεδομένων από την απαλοιφή του πλεονασμού. Όσο μικρότερα είναι τα κομμάτια, από τη μία τόσα περισσότερα μεταδεδομένα απαιτούνται, και από την άλλη τόσο μεγαλύτερο πλεονασμό απαλείφουμε, οπότε πρέπει να βρούμε ένα κατάλληλο μέγεθος που θα καταλήγει στην βέλτιστη εξοικονόμηση χώρου. Επίσης, επειδή η λήψη ενός στιγμιότυπου, όπως είδαμε εμπλέκει κυρίως λειτουργίες επί των μεταδεδομένων, μια κατάλληλη δομή είναι αναγκαία ώστε οι λειτουργίες αυτές να εκτελούνται γρήγορα και η λήψη του στιγμιότυπου να είναι ακαριαία. Ένα βασικό σχεδιαστικό ζήτημα αφορά λοιπόν την δομή των μεταδεδομένων και το επίπεδο λεπτομέρειας καταμερισμού του δίσκου σε κομμάτια, ώστε να επιτραπεί αποδοτικά το COW. Τέλος τα στιγμιότυπα αυτά θα θέλαμε να είναι ανεξάρτητες οντότητες, οι οποίες θα μπορούν να είναι αυτόνομες και φο-

ρητές σε ένα περιβάλλον υπολογιστικού νέφους. Τα παραπάνω ισχύουν και για τους κλώνους.

Ας εστιάσουμε τώρα στην απαίτηση για αποδοτική αποθήκευση με απαλοιφή διπλότυπων. Στα περιβάλλοντα υπολογιστικού νέφους, είναι πολύ συχνή η ύπαρξη πολλών εικονικών μηχανών, οι οποίες όπως σημειώσαμε στην 3.1.3, έχουν εικονικούς δίσκους με δεδομένα που εμφανίζουν πλεονασμό, όπως παρεμφερή λειτουργικά συστήματα. Όσο μάλιστα αυξάνονται τα μηχανήματα, τόσο αυξάνεται και ο βαθμός του πλεονασμού. Εδώ έρχεται η τεχνική της απαλοιφής διπλοτύπων που θα βοηθήσει να εξαλειφθεί ο εξωτερικός πλεονασμός ανάμεσα στα επικαλυπτόμενα δεδομένα διαφορετικών εικονικών μηχανών. Η τεχνική αυτή, μπορεί να μην βοηθήσει στην εξάλειψη του εσωτερικού χρονικού πλεονασμού, αφού κάτι τέτοιο επιτελείται ήδη από τα στιγμιότυπα και την τεχνική COW που εφαρμόζουν. Όμως, η απαλοιφή διπλοτύπων θα συνεισφέρει στην απαλοιφή του εσωτερικού χωρικού πλεονασμού, αποθηκεύοντας μια φορά, τεμάχια δεδομένων (chunks) που εμφανίζονται επαναλαμβανόμενα σε διαφορετικές περιοχές του ίδιου εικονικού δίσκου. Ο χρήστης δηλαδή θα απολαμβάνει μια υπηρεσία απαλοιφής διπλοτύπων στον guest, χωρίς να χρειάζεται να χρησιμοποιήσει κάποιο συγκεκριμένο σύστημα αρχείων ή άλλο εργαλείο. Η απαλοιφή διπλοτύπων πρέπει να σεβαστεί τις απαιτήσεις της πρωτεύουσας αποθήκευσης, προσφέροντας χαμηλό latency, οπότε δεν μπορούμε να χρησιμοποιήσουμε κάποιο υπάρχον σύστημα προσανατολισμένο στην δευτερεύουσα αποθήκευση (backup storage). Υπάρχουν κάποια συστήματα πρωτεύουσας αποθήκευσης που θα μπορούσαν να χρησιμοποιηθούν, αλλά ίσως θα ήταν πιο κομψό να απαγκιστρωθούμε από τις υλοποιήσεις του κάθε δεδομένου συστήματος και να υλοποιήσουμε μια μορφή εικονικού δίσκου που θα διαχειρίζεται εξ ολοκλήρου τις τεχνικές εξοικονόμησης χώρου. Αυτό και θα οδηγήσει σε αρχιτεκτονική ελευθερία όπου δεν χρειάζεται το σύστημά μας να υποταχθεί στις διεπαφές και τους περιορισμούς μιας συγκεκριμένης υλοποίησης απαλοιφής διπλοτύπων, και θα ενσωματώνει σε ένα υψηλότερο επίπεδο την λογική αφαίρεση για όλες τις τεχνικές εξοικονόμησης χώρου. Δεδομένης της ανάγκης για ισορροπία ανάμεσα στην υψηλή απόδοση και στην εξοικονόμηση χώρου, πρέπει να επιλέξουμε τις κατάλληλες πολιτικές για κάθε μια από τις παραμέτρους ενός συστήματος απαλοιφής διπλοτύπων όπως αυτές παρουσιάστηκαν στην 3.1.3.

Η απαλοιφή διπλοτύπων προϋποθέτει συνταγές και ευρετήρια αποτυπωμάτων, δηλαδή και πάλι χρειαζόμαστε δομές μετάφρασης και αντιστοίχισης των αριθμών τομέα

του δίσκου. Συνεπώς, τόσο η τεχνική COW για στιγμιότυπα και κλώνους, όσο και η απαλοιφή διπλοτύπων προϋποθέτουν τον χωρισμό του εικονικού δίσκου σε τεμάχια και στη συνέχεια προσπαθούν να απαλείψουν τον πλεονασμό μεταξύ όμοιων τέτοιων τεμαχίων. Αυτός ο χωρισμός επιφέρει και την ύπαρξη δομών αντιστοίχισης αριθμών τομέα εικονικού δίσκου σε τεμάχια, οι οποίες δομές αποτελούν μεταδεδομένα. Αναδύεται λοιπόν με φυσικό τρόπο η δυνατότητα δημιουργίας μιας ενιαίας δομής που θα διαχειρίζεται τα μεταδεδομένα των τεμαχίων τόσο για την υλοποίηση του COW όσο και για την υλοποίηση της απαλοιφής διπλοτύπων. Αυτό είναι άλλωστε και ένα βασικό πλεονέκτημα της ενιαίας μορφής εικονικού δίσκου που παρουσιάζουμε. Η αυτοτέλειά του όχι μόνο δεν δεσμεύει τον εικονικό δίσκο με κάποια συγκεκριμένη τεχνολογία, αλλά δεν απαιτεί καν την συνεργασία με άλλα επίπεδα για να παρέχει τα χαρακτηριστικά του. Επίσης με το να ενοποιούμε το COW και την απαλοιφή διπλοτύπων σε ένα στρώμα, καταφέρνουμε να απαλείψουμε και πιθανό πλεονασμό στα μεταδεδομένα, αφού όπως προαναφέραμε κάθε μια από τις δυο τεχνικές απαιτεί τεμαχισμό και άρα παρόμοιας φύσης μεταδεδομένα για την διαχείριση και την αντιστοίχιση τους με τομείς του εικονικού δίσκου.

Μια άλλη απαιτήσή μας, είναι η χρήση μικρής ποσότητας μεταδεδομένων για την υποστήριξη των επιθυμητών λειτουργιών. Αυτή η απαίτηση έρχεται σε σύγκρουση με την επιθυμία για υψηλή εξοικονόμηση χώρου, αφού η τελευταία απαιτεί τόσο στην απαλοιφή διπλοτύπων όσο και στην τεχνική COW, τεμάχια μικρού μεγέθους, ώστε κάθε μεταβολή στα δεδομένα να επιφέρει όσο γίνεται μικρότερη επιβάρυνση στο χώρο αποθήκευσης. Για έναν εικονικό δίσκο σταθερού μεγέθους όμως, μικρότερο μέγεθος τεμαχίων οδηγεί σε περισσότερα τεμάχια, και συνεπώς ο αυξημένος αυτός αριθμός τους επιφέρει αύξηση των μεταδεδομένων που πρέπει να διατηρούνται για την αντιστοίχιση και παρακολούθηση των τεμαχίων. Η αύξηση των μεταδεδομένων του ευρετηρίου αποτυπωμάτων είναι ακόμα πιο επιζήμια, αφού επιφέρει και αύξηση του χρόνου αναζήτησης σε αυτό. Έτσι ο έλεγχος για το εάν το αντίστοιχο τεμάχιο είναι διπλότυπο ή όχι καθυστερεί περαιτέρω, ειδικά από τη στιγμή που το ευρετήριο είναι υπερβολικά μεγάλο για να διατηρείται στην κύρια μνήμη. Είναι λοιπόν αναγκαία η διερεύνηση του μεγέθους τεμαχίου, που σε βάθος χρόνου θα βελτιστοποιήσει την εξοικονόμηση αποθηκευτικού χώρου και δεν θα επιβραδύνει σημαντικά την διαδικασία απαλοιφής διπλοτύπων.

3.4 Ελλείψεις

Εδώ θα εξετάσουμε λύσεις που ήδη υφίστανται αλλά εκπληρώνουν μόνο τμηματικά τους παραπάνω στόχους.

Θα μπορούσαμε να χρησιμοποιήσουμε ένα σύστημα αρχείων BTRFS στον host, στο οποίο θα αποθηκεύαμε τους εικονικούς δίσκους μας. Συγκεκριμένα θα μπορούσαμε να εκμεταλλευτούμε την δυνατότητα για κλώνους αρχείων που παρέχει το BTRFS (cp --reflink=always) και έτσι να παίρνουμε στιγμιότυπα και κλώνους των αρχείων εικονικών δίσκων, αφού το BTRFS από μόνο του εσωτερικά υλοποιεί COW, και τα διαμοιραζόμενα δεδομένα θα ήταν κοινά. Στα θετικά επίσης είναι ότι το BTRFS μπορεί να υποστηρίξει απαλοιφή διπλοτύπων. Τα μειονεκτήματα αυτής της προσέγγισης είναι ότι κατ' αρχάς περιοριζόμαστε στην χρησιμοποίηση ενός δεδομένου συστήματος αρχείων στον host, του οποίου ο μηχανισμός και η πολιτική είναι εσωτερικά προκαθορισμένη. Επίσης το BTRFS εγγενώς χωλαίνει σε όρους απόδοσης για μεγάλα αρχεία με συχνές τυχαίες εγγραφές, διότι εκτελεί μονίμως COW, ακόμα και για «hot»δεδομένα. Αυτό το κάνει για να εξασφαλίσει την συνέπεια των δεδομένων (data consistency) καθώς δεν έχει journal. Αυτό σημαίνει ότι ακόμα και για δεδομένα που δεν είναι μοιραζόμενα ανάμεσα σε στιγμιότυπα, το BTRFS εκτελεί COW, δηλαδή γράφει σε ένα νέο κομμάτι τα νέα και μετά πετάει τα παλιά και όλους τους δείκτες του B-tree μέχρι εκεί, και δεν πραγματοποιεί in-place modification. Χρησιμοποιώντας το BTRFS, ένας εικονικός δίσκος θα προσομοιώνεται μέσω ενός μεγάλου αρχείου που θα υφίσταται πολλές τυχαίες εγγραφές. Το συνεχές COW θα δεσμεύει για κάθε τέτοια μικρή εγγραφή ένα καινούργιο extent σε μια νέα περιοχή του δίσκου, οδηγώντας έτσι το αρχείο εικονικού δίσκου σε κατακερματισμό (fragmentation) και καταστρέφοντας ολοσχερώς την έννοια της τοπικότητας. Αυτός ο κατακερματισμός προφανώς θα είναι τρομερά ζημιογόνος σε περιπτώσεις που ο guest εκτελεί σειριακές εγγραφές και αναγνώσεις στον εικονικό δίσκο. Επίσης, πολλές από αυτές τις τυχαίες εγγραφές ακολουθούνται από `fsync()`, κάτι το οποίο υποβαθμίζει ακόμα περισσότερο την απόδοση, αφού το BTRFS εν πολλοίς βασίζεται στο in-place modification στη RAM και στα μόνο σποραδικά flushes -checkouts στην ορολογία του BTRFS-, όπου τα μπλοκ που έχουν γίνει shadowed γράφονται στο δίσκο. Έτσι, εάν αυτή η διαδικασία του shadowing απαιτηθεί πιο συχνά από ρητά `fsync()`, πράγμα σύνηθες σε περιβάλλον εικονικοποίησης όπου πρέπει να εξασφαλισθεί η ακεραιότητα των δεδομένων στον προσομοιούμενο εικο-

νικό δίσκο, τότε θα έχουμε μια σημαντική επιβάρυνση στον χρόνο που θα σπαταλάει το BTRFS για να γράφει στον δίσκο συνεχώς shadowed μπλοκ. Αυτά τα shadowed μπλοκ δεν αφορούν μόνο extents δεδομένων αλλά και όλα τα άλλα μπλοκ μεταδεδομένων που έχει το BTRFS στα διαφορετικά δέντρα του και τα οποία μάλιστα επηρεάζουν και τους προγόνους τους στο δέντρο, οπότε η επιβάρυνση πολλαπλασιάζεται (write amplification). Συγκεκριμένα ο χρόνος που απαιτείται για συγχρονισμό στο δίσκο, αυξάνεται κατά $log n$ για καθένα από τα 4 δέντρα που οφείλουν να εμπλέκονται στην ενημέρωση μεταδεδομένων για κάθε μεταβολή του αρχείου του εικονικού δίσκου.

Μια εναλλακτική λύση που αποφεύγει τα παραπάνω θα ήταν να χρησιμοποιούμε την επιλογή απενεργοποίησης του COW σε ολόκληρο το σύστημα αρχείων που θα είχε ο host (nodatacow option) ή στα συγκεκριμένα αρχεία που θα χρησιμοποιούνταν ως εικονικοί δίσκοι (chattr +C). Έτσι τα αρχεία εικονικών δίσκων θα ενημερώνονταν in-place, αποφεύγοντας τον κατακερματισμό και την επιβάρυνση ενημέρωσης των μεταδεδομένων. Κάθε επόμενη εγγραφή σε μοιραζόμενο extent ανάμεσα στο αρχείο ενός πρωτεύοντος δίσκου και στο αρχείο ενός στιγμιοτύπου, προκαλεί COW του extent. Στα νέα shadowed extents του πρωτεύοντος αρχείου, εφαρμόζεται πάλι in-place modification, προφανώς μέχρι να ληφθεί ένα νέο στιγμιότυπο. Έτσι και τα extents του στιγμιοτύπου διατηρούνται ανέπαφα και τα extents του πρωτεύοντος εικονικού δίσκου γίνονται με in-place modification, με το COW να εκτελείται μόνο όταν απαιτείται. Ως προς την απαλοιφή διπλοτύπων, μπορούμε να την εφαρμόσουμε μόνο για τα στιγμιότυπα, με μια διεργασία που θα δέχεται τα extents των αρχείων του BTRFS και θα εκτελεί την απαλοιφή των διπλότυπων. Τέτοιο πρόγραμμα υπάρχει άλλωστε ήδη (dedupremove) [btrb].

Όλα τα παραπάνω οδηγούν σε μια κομψή και εύκολη λύση, που όμως δεν είναι κλιμακώσιμη σε ένα περιβάλλον υπολογιστικού νέφους πολλών κόμβων, ενώ είναι επίσης και περιοριστική. Η λύση αυτή δουλεύει εάν έχουμε την απλή περίπτωση ενός host με ένα τοπικό σύστημα αρχείων BTRFS, αλλά δεν επεκτείνεται σε γενικότερες περιπτώσεις όπου έχουμε ένα NAS ή ένα clustered file system ή δεν έχουμε καν σύστημα αρχείων αλλά θέλουμε να χρησιμοποιήσουμε μια άλλη block-level τεχνολογία όπως το object storage. Είμαστε δηλαδή σαφώς περιορισμένοι στο BTRFS. Επίσης, αν και υπάρχει ένα εργαλείο από τρίτους (third-party tool) που υλοποιεί απαλοιφή διπλοτύπων, μια καθολική διαχείριση των τεμαχίων για όλο το storage σύστημα του εικονι-

κοποιημένου περιβάλλοντός μας απαιτεί στην καλύτερη περίπτωση πρόσβαση στον ειδικό κώδικα και δομές αυτού του εργαλείου. Στη χειρότερη περίπτωση, αυτή η πρόσβαση είτε είναι ανέφικτη, καθώς αναφέρεται σε εσωτερικές δομές του BTRFS, είτε δεν προσφέρει δυνατότητα για ελευθερία παραμετροποίησης, καθώς οι δομές αντιστοίχισης είναι άκαμπτες. Πράγματι, επειδή η απαλοιφή πλεονασμού που πραγματοποιεί το dedupremove γίνεται σε επίπεδο exetnts του συστήματος αρχείων, δεν μπορεί να παραμετροποιηθεί εξωτερικά και τα κέρδη της είναι περιορισμένα εξαιτίας του πιθανώς μεγάλου μεγέθους των extetns. Εναλλακτικές υλοποιήσεις που δεν θα εκμεταλλευόντουσαν τις εσωτερικές δομές του BTRFS, θα κατέληγαν σε διάσπαση των λειτουργιών απαλοιφής διπλοτύπων και υποστήριξης στιγμιοτύπων και άρα στην εισαγωγή πολλαπλών δομών μεταδεδομένων. Για τους ίδιους λόγους δεν χρησιμοποιούμε και άλλα συστήματα αρχείων με παρόμοιες δυνατότητες όπως το ZFS.

Ας προχωρήσουμε τώρα στην περίπτωση του qcow2. Το qcow2 θεωρητικά αποτελεί μια ευρέως διαδεδομένη μορφή που υποστηρίζει sparseness, κλώνους και στιγμιότυπα. Επίσης, αν και δεν υπάρχει κάποια επίσημη υλοποίηση, με βάση παρουσιάσεις και συζητήσεις που υπάρχουν φαίνεται ότι γίνονται προσπάθειες για ανάπτυξη ενός συστήματος απαλοιφής διπλοτύπων που θα επιδρά στα clusters του qcow2 [Can]. Προσπερνώντας όμως το ότι δεν υπάρχει κάποια δοκιμασμένη υλοποίηση απαλοιφής διπλοτύπων, προβλήματα φαίνεται να παρουσιάζουν και άλλα χαρακτηριστικά.

Το βασικό πρόβλημα του qcow2 είναι ότι τα στιγμιότυπα και οι κλώνοι δεν είναι ανεξάρτητες οντότητες αλλά είναι κομμάτια μιας αλυσίδας εξαρτήσεων που δεν τα κάνει αυτοτελή, ευέλικτα και μεταφέρσιμα. Οι κλώνοι είναι άρρηκτα συνδεδεμένοι με το αρχείο υποστήριξης στο οποίο αναφέρονται, και τα στιγμιότυπα είναι εσωτερικά στο αρχείο του εικονικού δίσκου και μη ορατά ως αυτόνομες οντότητες σε άλλα υποσυστήματα ενός περιβάλλοντος υπολογιστικού νέφους. Τα εναλλακτικά αυξητικά στιγμιότυπα παρουσιάζουν τις ίδιες παθογενείς εξαρτήσεις, καθώς η παρουσία όλων των copy-on-write αρχείων είναι απαραίτητη για την σύνθεση του δίσκου. Έτσι, ενώ οι κλώνοι και τα στιγμιότυπα δουλεύουν πολύ καλά στο απλοϊκό σενάριο ενός host, δεν μπορούν να κλιμακωθούν κατάλληλα σε ένα περιβάλλον υπολογιστικού νέφους με εκατοντάδες κόμβους όπου τα στιγμιότυπα θα έπρεπε να μετακινούνται από τον ένα κόμβο στον άλλον, προσαρμοζόμενα στην ελαστικότητα των πόρων και στις ανάγκες διανομής. Επίσης, η διαχείριση των στιγμιοτύπων υπόκειται στις τεχνικές και στις δομές αντιστοίχισης και reference counting του qcow2. Και πάλι αυτός ο διαχω-

ρισμός δημιουργεί πολλαπλά μεταδεδομένα, πιθανώς πλεοναστικά, ενώ δεν προσφέρει μια ενιαία, αφαιρετική υλοποίηση για ενοποίηση όλων των λειτουργιών εξοικονόμησης χώρου. Ακόμα χειρότερα, στα copy-on-write αρχεία, σε περίπτωση που το ζητούμενο cluster βρίσκεται μόνο στο αρχείο υποστήριξης, απαιτείται διπλή διάσχιση των εν λόγω δομών αντιστοίχισης, αν και μετά την πρώτη προσπέλαση τα μεταδεδομένα τοποθετούνται στην μνήμη. Η απόδοση επιβαρύνεται ακόμη περισσότερο στην περίπτωση των αυξητικών στιγμιοτύπων, όπου είναι πιθανόν να διασχιστούν οι δομές αντιστοίχισης όλων των copy-on-write αρχείων, ενώ η συγχώνευσή των αρχείων της αλυσίδας είναι μεν δυνατή, αλλά χρονοβόρα. Παράλληλα, η διαγραφή ενός αυξητικού στιγμιοτύπου που είναι κομμάτι μιας μεγαλύτερης αλυσίδας δεν είναι ακαριαία, αφού πριν τη διαγραφή απαιτείται η συγχώνευση των μεταβολών που περιέχει το προς διαγραφή στιγμιότυπο, με το επόμενο στιγμιότυπό του στην αλυσίδα. Δεδομένου ότι και άλλες πολύ δημοφιλείς μορφές εικονικών δίσκων (vmdk, vdi, vhd) χρησιμοποιούν ως μοντέλο στιγμιοτύπων την αλυσίδα αυξητικών στιγμιοτύπων, τα παραπάνω επιχειρήματα έλλειψης αυτονομίας επεκτείνονται και σε αυτούς.

Ειδικότερα για το qcow2, τα εσωτερικά στιγμιότυπα χρησιμοποιούν για τα δεδομένα τους τον διαθέσιμο χώρο του εικονικού δίσκου, κάτι που είναι άκομψο και δυσχεραίνει την παρακολούθηση και διαχείριση του χώρου του δίσκου από τον χρήστη. Καταληκτικά, αναφέρουμε ότι κατά την λήψη ενός εσωτερικού στιγμιοτύπου απαιτείται ενημέρωση όλων των μετρητών αναφορών των clusters, το οποίο είναι μια χρονοβόρα διαδικασία, ενώ το COW για ένα cluster που γράφεται μόνο στη μέση, μπορεί να καταλήξει σε παραπανίσιες λειτουργίες Ε/Ε (2 αναγνώσεις και 3 εγγραφές αντί για 1 ανάγνωση και 1 εγγραφή που είναι το ελάχιστο απαραίτητο).

Τέλος, θεωρούμε αποφευκτέα την χρήση κάποιας ήδη υπάρχουσας υλοποίησης συστήματος απαλοιφής διπλοτύπων σε κάποιο άλλο στρώμα, καθώς αυτή και πάλι θα μας δεσμεύσει με μια συγκεκριμένη διεπαφή και θα περιορίσει τις αρχιτεκτονικές μας επιλογές. Γενικώς δεν είναι επιθυμητό τα εμπλεκόμενα αντικείμενα σε μια μορφή εικονικού δίσκου να είναι άρρηκτα συνδεδεμένα με ένα σύστημα αρχείων ή μια συγκεκριμένη τεχνολογία αποθήκευσης, καθώς έτσι καταστρέφεται η ευελιξία του σχεδιασμού.

Υπογραμμίζουμε ξανά, ότι η ενιαία διαχείριση των λειτουργιών εξοικονόμησης χώρου, πέρα από την απαλοιφή διπλών μεταδεδομένων, εξασφαλίζει ταχύτητα, κομψότητα και ανεξαρτησία από άλλα επίπεδα στην στοίβα ενός περιβάλλοντος νέφους.

3.5 Μια πρώτη προσέγγιση: Επίπεδη Συνταγή

Τόσο για τα τεμάχια που χρειάζεται η απαλοιφή διπλοτύπων όσο και για την απαλοιφή πλεονασμού ανάμεσα στα στιγμιότυπα με την τεχνική COW, απαιτείται τεμαχισμός του εικονικού δίσκου σε κομμάτια. Μια πρώτη ιδέα λοιπόν θα ήταν να τον χωρίσουμε σε τεμάχια (chunks) ίσου μεγέθους και να διατηρούμε μια δομή αντιστοίχισης με την μορφή πίνακα για μετάφραση από αριθμό τομέα εικονικού δίσκου σε τεμάχια. Αυτή η δομή αποκαλείται και συνταγή (recipe). Τα τεμάχια θα είναι πολλαπλάσια του μεγέθους τομέα, οπότε και η μετάφραση ενός αριθμού τομέα σε τεμάχιο γίνεται σε 2 βήματα. Αρχικά πρέπει να βρούμε τον σειριακό αριθμό του τεμαχίου που αντιστοιχεί στον τομέα, κάτι που προκύπτει από ένα απλό γραμμικό μετασχηματισμό, διαιρώντας το sector number με το sectors_per_chunk ($=\frac{chunk_size}{sector_size}$). Στη συνέχεια, ο σειριακός αριθμός του τεμαχίου λειτουργεί ως δείκτης στον πίνακα αντιστοίχισης. Η θέση του πίνακα που προκύπτει θα έχει μια εγγραφή (entry) που θα περιέχει το αναγνωριστικό του τεμαχίου, το οποίο ανάλογα με τον τύπο του underlying storage μπορεί να είναι ένα μονοπάτι στο σύστημα αρχείων ή ένα object id. Πέρα από το αναγνωριστικό τεμαχίου, η εγγραφή αυτή θα περιέχει και το δικαίωμα εγγραφής στο τεμάχιο, έτσι ώστε να διαχωρίζει τα τεμάχια που είναι «hot» και τα τεμάχια που ανήκουν σε κάποιο στιγμιότυπο και πρέπει να γίνουν COW σε κάθε νέα αίτηση εγγραφής. Τέλος, η προσέγγιση αυτή μπορεί να υποστηρίξει και sparseness, αφού τόσο τα τεμάχια όσο και οι εγγραφές του πίνακα θα δεσμεύονται κατ᾽ απαίτηση, αλλά και απαλοιφή διπλοτύπων τεμαχίων.

Για να ληφθεί ένα στιγμιότυπο, αρκεί να αντιγράψουμε μόνο την συνταγή του πρωτεύοντος δίσκου και έτσι όλα τα τεμάχια θα γίνουν κατευθείαν διαμοιραζόμενα[2]. Παράλληλα πρέπει να μαρκάρουμε στην συνταγή του πρωτεύοντος δίσκου ότι όλα τα τεμάχια δεν είναι πλέον hot και άρα εγγράψιμα, αλλά απαιτούν COW. Έτσι οι επόμενες αιτήσεις εγγραφής στον κύριο δίσκο πυροδοτούν COW, μεταβολή δικαιωμάτων εγγραφής και ενημέρωση των δεικτών της συνταγής στα νέα τεμάχια. Τα παλιά τεμάχια μένουν απαράλλαχτα και είναι ακόμη προσβάσιμα από την συνταγή που ανήκει στο στιγμιότυπο.

Παρόλο που φαίνεται να πετυχαίνουμε τους στόχους μας, μπορούμε να διακρίνουμε ότι υπάρχουν περιθώρια για βελτίωση. Αν ο δίσκος είναι πολύ μεγάλος (τάξης των

[2] εφεξής ταυτίζουμε τον όρο «πρωτεύων δίσκος» με μια ζωντανή εγγράψιμη εκδοχή ενός εικονικού δίσκου, σε αντιπαραβολή με την μόνο για ανάγνωση σταθερή εκδοχή του στιγμιοτύπου

100GB) και τα μεγέθη τεμαχίων είναι πολύ μικρά, τότε τα μεταδεδομένα αυξάνονται σε βαθμό που ο χρόνος αντιγραφής τους κατά τη λήψη του στιγμιοτύπου είναι πλέον μη αμελητέος. Συγκεκριμένα για εικονικό δίσκο 100GB με τεμάχια 4KB, η συνταγή θα είναι 550MB και άρα η δημιουργία ενός στιγμιοτύπου με ένα συμβατικό δίσκο θα απαιτήσει πάνω από 10s. Παράλληλα παρατηρούμε ότι οι συνταγές των στιγμιοτύπων αυτών θα αρχίσουν δεσμεύουν έναν μη αμελητέο χώρο στο δίσκο, ειδικά στην περίπτωση που έχουμε συχνά στιγμιότυπα. Τα παραπάνω εξαναγκάζουν την επίπεδη συνταγή σε χρήση μεγάλου μεγέθους τεμαχίου, ώστε να έχει λίγα entries και άρα μικρό μέγεθος. Έτσι η λήψη στιγμιοτύπων θα είναι γρήγορη και δεν θα οδηγεί σε έκρηξη μεταδεδομένων, αλλά τα μεγάλα τεμάχια θα δυσχεράνουν την ανίχευση πλεονασμού.

Τα παραπάνω προβλήματα μπορούμε να τα αποφύγουμε με την εισαγωγή μιας δεντρικής δομής αντί της επίπεδης (flat) δομής που έχει μέχρι στιγμής η συνταγή. Έτσι, κατά την λήψη ενός στιγμιοτύπου, θα αντιγράφεται μόνο ο κόμβος-ρίζα, ο οποίος θα είναι πολύ μικρότερος (110KB για 2 επίπεδα και 6KB για 3 επίπεδα, για δίσκο 100GB και τεμάχια 4KB). Συνεπώς με αυτήν την στρατηγική έχουμε ελαχιστοποιήσει το χρόνο λήψης στιγμιοτύπου. Πρακτικά τον έχουμε εκμηδενίσει, αφού για δίσκο 100TB και τεμάχια 4KB έχουμε αντιγραφή 3.5MB για 2 επίπεδα και 64KB για 3 επίπεδα, κάτι που γίνεται σχεδόν ακαριαία. Παράλληλα θα δούμε ότι η δεντρική δομή, πέρα από τον διαμοιρασμό των δεδομένων, θα επιτρέψει και τον διαμοιρασμό μεταδεδομένων ανάμεσα στα στιγμιότυπα, ώστε συχνά στιγμιότυπα να μην επιβαρύνουν το σύστημα αποθήκευσης με πλεοναστικά μεταδεδομένα.

3.6 Ο σχεδιασμός μας: Δενδρική Συνταγή

Οι παραπάνω παρατηρήσεις μας οδήγησαν στην δημιουργία του ακόλουθου σχεδιασμού που συνδυάζει τεχνολογίες εξοικονόμησης χώρου, με την δυνατότητα αυτόνομων στιγμιοτύπων και κλώνων, κάτω από ένα ενιαίο διαχειριστικό σχήμα μεταδεδομένων.

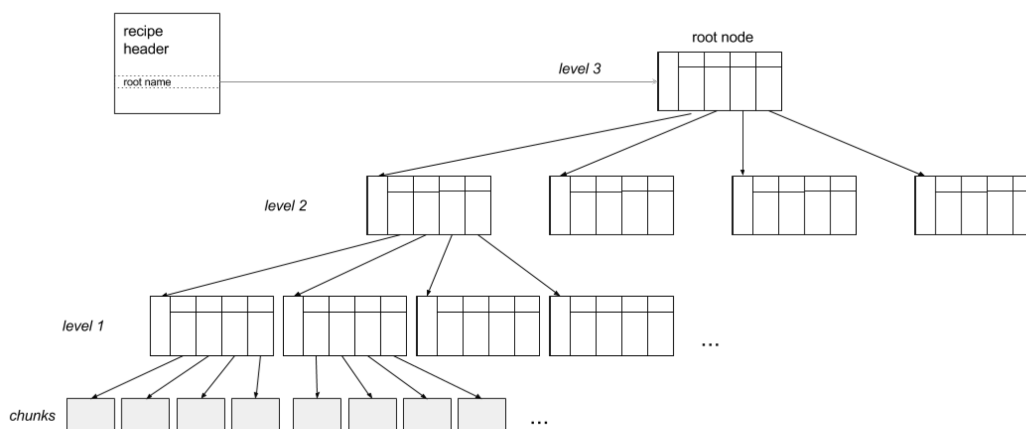
Ο σχεδιασμός μας συνοπτικά έχει τα εξής χαρακτηριστικά:

- τα δεδομένα του εικονικού δίσκου χωρίζονται σε τεμάχια (chunks) σταθερού μεγέθους
- υπάρχει μια συνταγή (recipe) που συνδέει τους αριθμούς τομέων εικονικού δίσκου

με τα τεμάχια

- η συνταγή αυτή αποτελείται από ένα δέντρο με κόμβους που, αν είναι εσωτερικοί, δείχνουν σε άλλους κόμβους, και, αν είναι φύλλα, δείχνουν σε τεμάχια. Η επικεφαλίδα της συνταγής δείχνει στον κόμβο-ρίζα και για να γίνει η μετάφραση αριθμού τομέα εικονικού δίσκου σε τεμάχιο πρέπει να γίνει διάσχιση από την ρίζα σε κάποιο φύλλο.
- μπορούν να υπάρχουν περισσότερες από μία συνταγές για κάθε εικονικό δίσκο. Πέρα από την βασική συνταγή για τον πρωτεύοντα εικονικό δίσκο υπάρχουν συνταγές που αντιπροσωπεύουν στιγμιότυπα και κλώνους του.
- υπάρχει διαμοιρασμός των κοινών τεμαχίων και των κοινών κόμβων ανάμεσα σε πρωτεύοντες δίσκους, στιγμιότυπα και κλώνους.
- όταν ένα κόμβος/τεμάχιο που είναι διαμοιραζόμενος ανάμεσα στον πρωτεύοντα εικονικό δίσκο και κάποιο στιγμιότυπο απαιτήσει αλλαγή, τότε γίνεται COW, και ένας νέο κόμβος/τεμάχιο δημιουργείται για να αποθηκεύσει τις αλλαγές.
- υποστηρίζεται δημιουργία και διαγραφή στιγμιοτύπων καθώς και επαναφορά του πρωτεύοντος εικονικού δίσκου σε προγενέστερη κατάσταση (snapshot restore)
- οι κόμβοι του δέντρου έχουν σταθερό και ίδιο μεταξύ τους μέγεθος, ενώ ο χρήστης μπορεί να ορίσει τον αριθμό των επιπέδων και το μέγεθος των τεμαχίων κατά την δημιουργία του εικονικού δίσκου, και έκτοτε αυτά μένουν σταθερά

Έχουμε έναν εικονικό δίσκο ο οποίος στα πρότυπα ενός πραγματικού, οφείλει να αποθηκεύει τα δεδομένα του σε τομείς των 512bytes. Οι τομείς αυτοί ομαδοποιούνται σε τεμάχια που έχουν μέγεθος πολλαπλάσιο του μεγέθους ενός τομέα. Κάθε τεμάχιο είναι διακριτό από τα υπόλοιπα χάρη σε ένα μοναδικό όνομα τεμαχίου (chunk name). Η αντιστοίχιση από αριθμό τομέα εικονικού δίσκου σε τεμάχιο γίνεται με διάσχιση ενός δέντρου-συνταγής, η οποία ξεκινάει από την ρίζα, και σε κάθε κόμβο υπολογίζεται ο επόμενος κόμβος ή οι επόμενοι κόμβοι που πρέπει να μεταβούμε για να μεταφραστούν όλοι οι τομείς μιας αίτησης Ε/Ε σε ονόματα τεμαχίων. Η διαδικασία αυτή σταματάει στους κόμβους-φύλλα που περιέχουν τα ονόματα των τεμαχίων. Στην πραγματικότητα, η δενδρική συνταγή μας είναι ένα B-tree παρόμοιο με αυτά που χρησιμοποιεί το BTRFS, μόνο που στην περίπτωσή μας δεν έχουμε COW για κάθε εγγραφή σε κόμβο/τεμάχιο, αλλά μόνο για τις εγγραφές σε κόμβους/τεμάχια που είναι μοιραζόμενα ανάμεσα σε στιγμιότυπα και τον πρωτεύοντα δίσκο.



Σχήμα 3.7: *Η δεντρική συνταγή για ένα εικονικό δίσκο με 3 επίπεδα και 3 εγγραφές ανά κόμβο*

Πιο αναλυτικά, κάθε κόμβος του ιεραρχικού δέντρου συνταγής περιέχει εγγραφές (entries)[3], που ανάλογα με το επίπεδο του δέντρου που βρίσκεται ο κόμβος, αναφέρονται είτε σε τεμάχια είτε σε άλλους κόμβους. Κάθε κόμβος περιέχει στην αρχή μια επικεφαλίδα κόμβου (Node Header) που περιγράφει το επίπεδο (level) του δέντρου στο οποίο βρίσκεται ο κόμβος, και το μέγεθος του κόμβου και στη συνέχεια ανάλογα με το μέγεθός του ένα συγκεκριμένο αριθμό από entries. Τα επίπεδα αρχίζουν την αρίθμησή τους από το 1 για το κατώτερο, το οποίο περιέχει κόμβους-φύλλα με entries που δείχνουν σε τεμάχια, και το υψηλότερο επίπεδο το έχει ο κόμβος-ρίζα. Το μέγεθος του κόμβου είναι σταθερό για όλους τους κόμβους σε ένα εικονικό δίσκο και είναι συνάρτηση του μεγέθους τεμαχίου, του μεγέθους του δίσκου και του ύψους του δέντρου. Συγκεκριμένα, κατά την δημιουργία του εικονικού δίσκου, αρχικά υπολογίζεται ο αριθμός των τεμαχίων που απαιτούνται για την συγκρότηση ενός εικονικού δίσκου με το δοθέν μέγεθος (total chunks), απλώς διαιρώντας το μέγεθος του δίσκου με το μέγεθος του τεμαχίου και στρογγυλοποιώντας προς τα πάνω. Στη συνέχεια, υπολογίζεται ο αριθμός των εγγραφών που χρειάζεται να έχει κάθε κόμβος (entries_per_node ή epn) ώστε στο πρώτο επίπεδο να δεικτοδοτηθούν όλα τα τεμάχια. Στο υψηλότερο επίπεδο, έστω h μιας που ταυτίζεται και με το ύψος του δέντρου, θα έχουμε μόνο τον κόμβο-ρίζα και άρα epn entries. Στο αμέσως χαμηλότερο επίπεδο $(h-1)$ θα έχουμε τους epn κόμβους στους οποίους δείχνει ο κόμβος-ρίζα και άρα συνολικά epn^2 entries, στο αμέσως χαμηλότερο $(h-2)$ θα έχουμε τους epn^2 κόμβους

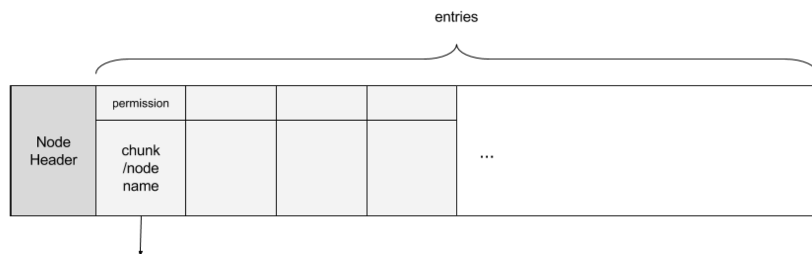
[3]εφεξής θα χρησιμοποιούμε τον όρο «entry» αντί για «εγγραφή», ώστε να αποφύγουμε πιθανή σύγχυση με την ταυτόσημη απόδοση στα ελληνικά του όρου «write»

στους οποίους δείχνουν οι epn κόμβοι του επιπέδου $h-1$ και άρα συνολικά epn^3 entries και ούτω καθεξής, μέχρι να φτάσουμε στο επίπεδο 1 όπου θα έχουμε epn^{h-1} κόμβους οι οποίοι θα δείχνουν σε epn^h τεμάχια. Άρα έχουμε την ισότητα:

$$epn^h = total_chunks \Rightarrow epn = \lceil total_chunks^{\frac{1}{h}} \rceil = \left\lceil \sqrt[h]{total_chunks} \right\rceil$$

Για να εξάγουμε το συνολικό μέγεθος του κόμβου, προσθέτουμε στο μέγεθος που καταλαμβάνουν οι epn εγγραφές το μέγεθος της επικεφαλίδας του κόμβου.

Έτσι λοιπόν, κάθε κόμβος του δέντρου περιέχει ένα σταθερό αριθμό από entries. Κάθε entry περιέχει ένα όνομα κόμβου ή τεμαχίου και με αυτόν τον τρόπο «δείχνει» σε αυτόν τον κόμβο ή σε αυτό το τεμάχιο. Ανάλογα με το επίπεδό του κόμβου και την θέση του κόμβου στο επίπεδο, υπολογίζεται ποιο εύρος τομέων καλύπτει κάθε entry. Για παράδειγμα το πρώτο entry του πρώτου κόμβου του πρώτου επιπέδου καλύπτει τους τομείς 0 έως $\frac{chunksize}{512} - 1$. Το τέταρτο entry του τρίτου κόμβου του δευτέρου επιπέδου καλύπτει τους τομείς $(2epn^2 + 3epn) * \frac{chunksize}{512}$ έως $(2epn^2 + 4epn) * \frac{chunksize}{512} - 1$ και εν γένει το x-οστό entry του y-οστού κόμβου του z επιπέδου καλύπτει τους τομείς $((y-1)epn^z + (x-1)epn^{z-1}) * \frac{chunksize}{512}$ έως $((y-1)epn^z + x * epn^{z-1}) * \frac{chunksize}{512} - 1$. Έτσι, ανάλογα με το σε ποιους τομείς αναφέρεται η αίτηση Ε/Ε, προσπελάζουμε και τα κατάλληλα entries του κόμβου. Αυτή η διαδικασία αναδρομικά καταλήγει στους κόμβους-φύλλα, όπου κάθε entry δείχνει σε ένα τεμάχιο και έτσι η αίτηση μπορεί να ικανοποιηθεί προσπελάζοντας τα δεδομένα -όσα χρειάζονται- αυτού του τεμαχίου.



Σχήμα 3.8: *Ένας τυπικός κόμβος (node)*

Κάθε entry περιέχει πέρα από το όνομα του κόμβου/τεμαχίου και τα δικαιώματα εγγραφής (permissions) ανάλογα με το αν ο κόμβος/τεμάχιο είναι «hot», οπότε και μπορούμε να γράψουμε, ή αν είναι διαμοιραζόμενος σε στιγμιότυπα, οπότε και δεν μπορούμε να γράψουμε. Αν είναι hot, μια αίτηση εγγραφής γράφει κατευθείαν στον εν λόγω κόμβο/τεμάχιο. Αν είναι read-only, τότε κατά τα πρότυπα του COW, δημιουρ-

γούμε έναν νέο κόμβο/τεμάχιο και γράφουμε σε αυτόν, αφήνοντας τον αρχικό απαράλλαχτο. Αυτό το δικαίωμα εγγραφής δεν αφορά μόνο τον κόμβο/τεμάχιο στον οποίο «δείχνει» το entry αλλά και όλο το υποδέντρο που απλώνεται από κάτω. Αυτό γίνεται έτσι ώστε με την λήψη ενός στιγμιοτύπου να μην χρειάζεται διάσχιση όλων των κόμβων του δέντρου για να μετατρέψουμε τα δικαιώματά τους σε μόνο για ανάγνωση, αλλά αυτή η μετατροπή να γίνει σταδιακά, καθώς οι κόμβοι γίνονται COW. Αυτή είναι μια «lazy» ιδέα που προέρχεται από τον μηχανισμό reference counting των B-trees που παρουσιάσαμε προηγουμένως στην ενότητα 3.1.2. Έτσι κάθε φορά που σε μια αίτηση εγγραφής προσπελάζεται ένα entry που έχει read-only δικαίωμα και δείχνει σε ένα κόμβο K, τότε θα δημιουργείται ο καινούργιος shadowed κόμβος K', θα αντιγράφονται τα entries του κόμβου K στον K', αλλά τα δικαιώματα όλων αυτών των entries θα γίνονται read-only. Αυτό γιατί, αφού ο κόμβος K ήταν read-only, με τον lazy-αλγόριθμό μας ισχύει επαγωγικά ότι και όλοι κόμβοι/τεμάχια του υποδέντρου του θα είναι read-only, παρόλο που δεν φροντίσαμε για λόγους απόδοσης να ενημερώσουμε τα δικαιώματά τους κατά την λήψη του στιγμιοτύπου. Την κατά απαίτηση (on demand) λογική που το COW εφαρμόζει στα δεδομένα, την επεκτείνουμε λοιπόν και στα μεταδεδομένα.

Τέλος, να αναφέρουμε ότι το entry μπορεί να περιέχει αντί για το όνομα του κόμβου ή του τεμαχίου και την τιμή 0. Αυτός είναι ο βασικός μηχανισμός που ο δίσκος μας εφαρμόζει sparseness και είναι μια ιδέα δανειζόμενη από την υλοποίηση των αραιών αρχείων στα συστήματα αρχείων ext. Και εδώ λοιπόν, μια αίτηση εγγραφής που θα χρειαστεί να προσπελάσει ένα μηδενικό entry θα δημιουργήσει εκείνη την στιγμή τον κόμβο ή το τεμάχιο, ενώ μια αίτηση ανάγνωσης θα επιστρέψει μηδενικά. Άρα λοιπόν δεσμεύουμε μόνο τα τεμάχια στα οποία έχει γίνει εγγραφή στον εικονικό δίσκο και μόνο όσους κόμβους χρειάζονται για την προσπέλασή των τελευταίων. Ο εικονικός δίσκος μας αυξάνει το χώρο που καταλαμβάνει στο πραγματικό φυσικό δίσκο σταδιακά.

Ο λόγος που τοποθετούμε επικεφαλίδα στην αρχή ενός κόμβου είναι για να μπορεί ένας κόμβος να είναι μια αυτοτελής οντότητα που θα μπορεί να χρησιμοποιηθεί ανεξάρτητα από το υπόλοιπο δέντρο. Έτσι είναι υποχρεωτικό να τοποθετήσουμε το επίπεδο και το μέγεθος του κόμβου, ώστε σε περίπτωση προσπέλασης έξω από το πλαίσιο του εικονικού δίσκου στον οποίο ανήκει, να είναι γνωστό το εύρος των τομέων που καλύπτει κάθε εγγραφή του κόμβου.

Μια συνταγή (recipe) λοιπόν απαρτίζεται από μια συλλογή από κόμβους και από μια επικεφαλίδα συνταγής (RecipeHeader). Μια συνταγή μπορεί να αντικατοπτρίζει έναν πρωτεύοντα εικονικό δίσκο, ένα στιγμιότυπο ή έναν κλώνο. Στην ουσία ο κλώνος αντιμετωπίζεται ως ένας πρωτεύων εικονικός δίσκος που αρχικά μοιράζεται κάποια δεδομένα. Η επικεφαλίδα της συνταγής μπορεί να αναφέρεται λοιπόν σε ένα πρωτεύοντα εικονικό δίσκο ή σε ένα στιγμιότυπο, και περιλαμβάνει χρήσιμες πληροφορίες για το δέντρο των κόμβων που περιγράφουν τον εικονικό δίσκο, όπως το μέγεθος του εικονικού δίσκου, το μέγεθος των τεμαχίων, τον αριθμό των επιπέδων του δέντρου, το μέγεθος κάθε κόμβου, το εάν αυτή η συνταγή αναφέρεται σε στιγμιότυπο, την διεύθυνση του κόμβου ρίζας της δενδρικής δομής και άλλες πληροφορίες χρήσιμες στην υλοποίηση, όπως θα δούμε παρακάτω.

Ανακεφαλαιώνοντας έχουμε τις εξής οντότητες:

- εικονικός δίσκος: τα δεδομένα του τεμαχίζονται σε τεμάχια και η αντιστοίχιση μεταξύ αριθμού τομέα εικονικού δίσκου και τεμαχίου γίνεται μέσω μιας συνταγής με δεντρική μορφή
- τεμάχιο: ομάδα δεδομένων με μέγεθος πολλαπλάσιο του τομέα
- συνταγή: δομή αντιστοίχισης για την μετάφραση τομέων σε ονόματα τεμαχίων. Μια συνταγή απαρτίζεται λογικά από μια επικεφαλίδα συνταγής και ένα δέντρο από κόμβους.
- επικεφαλίδα συνταγής: περιέχει πληροφορίες για τον εικονικό δίσκο και ένα δείκτη στον κόμβο-ρίζα της δενδρικής συνταγής
- κόμβος: δομική μονάδα της δενδρικής συνταγής που απαρτίζεται από ένα αριθμό εγγραφών (entries). Κάθε entry δείχνει σε έναν άλλο κόμβο, αν ο κόμβος είναι εσωτερικός, ή σε ένα τεμάχιο, αν ο κόμβος είναι φύλλο, και περιέχει επίσης και το δικαίωμα εγγραφής, δηλαδή την πληροφορία για το αν ο κόμβος/τεμάχιο που δεικτοδοτείται ανήκει σε κάποιο στιγμιότυπο ή όχι και άρα πληροφορεί και για το εάν απαιτείται COW σε μια αίτηση εγγραφής.

Σε μια αίτηση Ε/Ε που έρχεται από τον guest λοιπόν, αρχικά προσπελάσεται ο κόμβος-ρίζα. Σε αυτόν, όπως και σε κάθε άλλο κόμβο, διαβάζονται τα κατάλληλα entries, των οποίων τα υποδέντρα περιέχουν τους τομείς της αίτησης. Η ανάγνωση γίνεται με μία λειτουργία Ε/Ε. Αν έχουμε αίτηση εγγραφής και ο κόμβος/τεμάχιο που δείχνει ένα entry είτε δεν έχει δεσμευτεί ακόμα είτε ανήκει σε κάποιο στιγμιότυπο και άρα πρέ-

πει να γίνει COW, τότε δεσμεύεται ο κατάλληλος κόμβος/τεμάχιο, και στην COW περίπτωση γίνεται επίσης αντιγραφή από τον παλιό κόμβο/τεμάχιο. Αφού ολοκληρωθούν όλες οι δεσμεύσεις νέων κόμβων/τεμαχίων, γίνεται ενημέρωση των ονομάτων και των δικαιωμάτων των entries που αναφέρονται στους νεοσύστατους κόμβους/τεμάχια. Η ενημέρωση γίνεται σε μία λειτουργία Ε/Ε. Στη συνέχεια, για κάθε επιλεγμένο entry προσπελάσεται ο κόμβος/τεμάχιο που δείχνεται, με αναδρομική εφαρμογή της εν λόγω διαδικασίας, στα πρότυπα μιας DFS διάσχισης του δέντρου. Αυτή σταματά στα τεμάχια, όπου και υπάρχει ανάγνωση/εγγραφή των κατάλληλων τομέων.

Ονοματοδοσία

Για να δεικτοδοτούμε τους διάφορους κόμβους και τα διάφορα τεμάχια, εισάγουμε το παρακάτω σχήμα ονοματοδοσίας, υπό το πρίσμα ενός μεγάλου περιβάλλοντος υπολογιστικού νέφους με πολλούς εικονικούς δίσκους. Κάθε εικονικός δίσκος πρέπει να είναι μοναδικά διαχωρίσιμος σε όλο το σύστημά μας, οπότε του αναθέτουμε ένα μοναδικό αναγνωριστικό (virtual disk id). Κάθε τεμάχιο ενός τέτοιου δίσκου οφείλει να είναι διακριτό από τα υπόλοιπα, οπότε αναθέτουμε σειριακά αύξοντες αριθμούς για όσα τεμάχια απαιτούνται, ανάλογα με το μέγεθος του εικονικού δίσκου. Επειδή όμως κατά την λήψη στιγμιοτύπων, τα τεμάχια αυτά θα πρέπει να μένουν αμετάβλητα (immutable) και τα νέα δεδομένα να γράφονται σε καινούργια τεμάχια που και αυτά με την σειρά τους μπορεί να χρειαστεί να γίνουν immutable, πρέπει κάθε τεμάχιο να έχει και ένα snapshot id για να είναι μοναδικό. Συνεπώς κάθε στιγμιότυπο ενός δίσκου έχει ένα snapshot id. Αρχικά το snapshot id είναι 1. Κατά την λήψη ενός στιγμιοτύπου αυξάνεται το snapshot id και τα νέα τεμάχια που γίνονται COW αποκτούν το νέο αυτό snapshot id. Τα snapshot ids δεν ανακυκλώνονται αλλά αυξάνονται μονοτονικά. Έτσι τα τεμάχια με το ίδιο snapshot id ουσιαστικά περιέχουν τα τεμάχια που άλλαξαν σε σχέση με το προηγούμενο στιγμιότυπο του εικονικού δίσκου. Έτσι ένα αναγνωριστικό τεμαχίου (chunk id) προσδιορίζεται μονοσήμαντα σε ολόκληρο το σύστημα από το virtual disk id, το snapshot id και τον αύξοντα αριθμό του.

Παρομοίως, και οι κόμβοι πρέπει να είναι διακριτοί μεταξύ τους. Επειδή εδώ έχουμε δεντρική δομή, ένα node id οφείλει να περιέχει τον αριθμό του επιπέδου του και τον αύξοντα αριθμό του στο συγκεκριμένο επίπεδο. Οι αύξοντες αριθμοί σε ένα επίπεδο ανατίθενται σειριακά από τους αριστερούς προς τους δεξιούς κόμβους. Όπως και τα τεμάχια, έτσι και οι κόμβοι μπορεί να γίνουν immutable κατά την λήψη ενός στιγμιο-

τύπου, οπότε και αυτά συνδέονται με ένα snapshot id. Έτσι ένα node id προσδιορίζεται μονοσήμαντα σε ολόκληρο το σύστημα από το virtual disk id, το snapshot id, το επίπεδό του στο δέντρο και τον αύξοντα αριθμό του στο επίπεδο.
Ένας κλώνος ενός εικονικού δίσκου δεν είναι απαραίτητο να υπάρχει κάτω από τον ονοματόχωρο του αρχικού δίσκου, οπότε αντιμετωπίζεται ως ένας ξεχωριστός εικονικός δίσκος που μπορεί απλώς να αναφέρεται σε ίδιους κόμβους και τεμάχια. Τέλος, επειδή σκοπεύουμε να εφαρμόσουμε απαλοιφή διπλοτύπων στους immutable κόμβους και τεμάχια των στιγμιοτύπων, αναδύεται η ανάγκη για την ύπαρξη ενός ξεχωριστού ονοματόχωρου όπου τόσο οι κόμβοι όσο και τα τεμάχια θα περιγράφονται και θα προσδιορίζονται με μοναδικό τρόπο από το περιεχόμενό τους, κάτι που είναι εφικτό μέσω μιας τιμής κατακερματισμού (αποτύπωμα).

Ένα entry λοιπόν μπορεί να περιέχει ένα chunk id, ένα node id ή ένα αποτύπωμα, αλλά και την τιμή 0 εάν ο κόμβος/τεμάχιο δεν έχει δημιουργηθεί ακόμα.

Στιγμιότυπα και Κλώνοι

Ο χρήστης πρέπει να έχει την δυνατότητα να πάρει στιγμιότυπα του δίσκου, δηλαδή να αποθηκεύσει την κατάσταση των δεδομένων του κάποια χρονική στιγμή. Το στιγμιότυπο αυτό μπορεί αργότερα να το χρησιμοποιήσει για διάβασμα (να διαβάζει τα δεδομένα του δίσκου όπως ήταν την χρονική στιγμή που τραβήχτηκε το στιγμιότυπο), καθώς και να αποκαταστήσει τον κύριο δίσκο με το στιγμιότυπο, επαναφέροντας τα δεδομένα που αυτός είχε την στιγμή που λήφθηκε το στιγμιότυπο. Για να πάρουμε ένα στιγμιότυπο ενός εικονικού δίσκου, αρκεί να αντιγράψουμε τα περιεχόμενα του ριζικού κόμβου σε ένα νέο κόμβο και να εφαρμόσουμε δικαιώματα μόνο για ανάγνωση σε όλες τις εγγραφές τόσο του παλιού όσο και του νέου ριζικού κόμβου. Έτσι, χρησιμοποιώντας τον lazy τρόπο εφαρμογής δικαιωμάτων που περιγράψαμε, το δέντρο γίνεται σημασιολογικά immutable ακαριαία, αλλά τα δικαιώματα των κόμβων/τεμαχίων αλλάζουν σταδιακά. Ο παλιός κόμβος-ρίζα ανήκει στον πρωτεύοντα εικονικό δίσκο και είναι προσπελάσιμος μέσω της επικεφαλίδας συνταγής του πρωτεύοντος δίσκου και ο νέος κόμβος-ρίζα ανήκει στο στιγμιότυπο και είναι προσπελάσιμος μέσω μιας νεοσύστατης επικεφαλίδας συνταγής του στιγμιοτύπου.
Έστω ότι μετά από την λήψη ενός στιγμιοτύπου έχουμε μια αίτηση εγγραφής στον πρωτεύοντα εικονικό δίσκο και άρα η διάσχιση του δέντρου θα γίνει μέσω του κόμβου-ρίζας που ανήκει στον πρωτεύοντα δίσκο και όχι του κόμβου-ρίζας που ανήκει στο

στιγμιότυπο. Αρχικά πρέπει να βρεθεί το σύνολο των εγγραφών που πρέπει να προσπελάσουμε, που για απλότητα ας υποθέσουμε ότι είναι μία, έστω e. Επειδή η εγγραφή αυτή έχει δικαίωμα μόνο για ανάγνωση, πρέπει να γίνει COW του κόμβου που δείχνει η e, έστω K. Δημιουργείται ένας νέος κόμβος K' και αντιγράφονται τα περιεχόμενα (εγγραφές) του K στον K', με την διαφορά ότι οι εγγραφές στον K' έχουν όλες δικαιώματα μόνο για ανάγνωση. Αυτό γιατί δείχνουν σε κόμβους/τεμάχια που έγιναν immutable, αφού λήφθηκε ένα στιγμιότυπο, και άρα κάθε προσπάθεια εγγραφής σε αυτούς θα πρέπει να πυροδοτεί ένα COW. Τέλος, ενημερώνεται η εγγραφή e του κόμβου ρίζας, ώστε να δείχνει στον K' και να έχει πλέον δικαίωμα εγγραφής, αφού δημιουργήθηκε ένα νέο τεμάχιο για τις hot εγγραφές στον πρωτεύοντα δίσκο. Η ενημέρωση αυτή γίνεται προφανώς στον κόμβο-ρίζα του πρωτεύοντος δίσκου και όχι στον κόμβο-ρίζα του στιγμιοτύπου που παραμένει αμετάβλητος και του οποίου η αντίστοιχη εγγραφή e συνεχίζει να δείχνει στον K. Η παραπάνω διαδικασία συνεχίζεται αναδρομικά για τον K', δηλαδή στη θέση του κόμβου-ρίζας του πρωτεύοντος δίσκου μπαίνει ο K'. Μόλις φτάσουμε σε κάποιον κόμβο-φύλλο L', βρίσκουμε τις κατάλληλες εγγραφές του και άρα τα κατάλληλα τεμάχια. Για κάθε τέτοιο τεμάχιο που απαιτείται εγγραφή, δημιουργούμε ένα καινούργιο, αν χρειάζεται αντιγράφουμε τα δεδομένα του παλιού στο καινούργιο, γράφουμε τα νέα δεδομένα και τέλος ενημερώνουμε τις εγγραφές του L' ώστε να δείχνουν στα νέα τεμάχια και να έχουν δικαίωμα εγγραφής.

Η προσπέλαση για τις αιτήσεις ανάγνωσης γίνεται κανονικά με βάση τους κόμβους/τεμάχια που δείχνουν τα entries, χωρίς κάποια επιπλέον μέριμνα. Σε μια αίτηση ανάγνωσης μπορεί να προσπελάσουμε τόσο immutable όσο και νεοσύστατους mutable κόμβους/τεμάχια. Μια αίτηση εγγραφής σε ένα στιγμιότυπο απαγορεύεται εκ προοιμίου.

Για να γίνει αποκατάσταση ή αλλιώς επαναφορά ενός στιγμιοτύπου (snapshot restore), το μόνο που απαιτείται είναι η αντιγραφή των εγγραφών του κόμβου-ρίζας του στιγμιοτύπου στον κόμβο-ρίζα του πρωτεύοντος δίσκου. Τα δικαιώματα του αποκατεστημένου κόμβου-ρίζας του πρωτεύοντος εικονικού δίσκου θα είναι μόνο για ανάγνωση, μιας που το δέντρο του στιγμιοτύπου πρέπει να παραμείνει immutable.

Η παραπάνω διαδικασία, έχει το πλεονέκτημα της ακαριαίας δημιουργίας στιγμιοτύπου, ενώ καθίσταται εμφανές πως αν μετά από λίγο ληφθεί ένα καινούργιο στιγμιότυπο, τα επιπλέον μεταδεδομένα που θα αποθηκευτούν θα είναι μόνο οι κόμβοι που

θα έχουν προσπελαστεί για τεμάχια που έχουν μεταβληθεί. Θα ποσοτικοποιήσουμε αυτό το κέρδος σε μεταδεδομένα αργότερα, αλλά ήδη μπορεί να γίνει εμφανής η αισθητή εξοικονόμηση.

Για να υποστηρίξουμε κλώνους, αρκεί να δημιουργήσουμε ένα νέο δίσκο και να αντιγράψουμε τον κόμβο-ρίζα του στιγμιοτύπου από το οποίο θέλουμε να δημιουργήσουμε τον κλώνο. Προφανώς για τους ίδιους λόγους με παραπάνω, πρέπει να θέσουμε τα δικαιώματα του κόμβου-ρίζας του κλώνου σε μόνο για ανάγνωση. Και εδώ έχουμε και ακαριαία δημιουργία του κλώνου αλλά και μοιραζόμενα δεδομένα και μεταδεδομένα, εξοικονομώντας χώρο. Μια αίτηση εγγραφής σε ένα κλώνο ακολουθεί την διαδικασία σταδιακών COW που περιγράψαμε και για τα μεταδεδομένα ενός πρωτεύοντος εικονικού δίσκου μετά από την λήψη στιγμιοτύπου.

Απαλοιφή διπλοτύπων

Εδώ θα περιγράψουμε την εφαρμογή ενός συστήματος απαλοιφής διπλοτύπων πάνω στην μορφή του εικονικού δίσκου που προαναφέραμε.

Αρχικά, η inline προσέγγιση επιφέρει μια μη αμελητέα επιβάρυνση στο κρίσιμο μονοπάτι, αφού πρέπει το σύστημα να παρεμβληθεί σε μια αίτηση εγγραφής, να υπολογίσει την τιμή κατακερματισμού των δεδομένων και να κοιτάξει στο ευρετήριο αποτυπωμάτων για το εάν υπάρχει ήδη το εν λόγω τεμάχιο. Παρόλο που ίσως αυτή η τεχνική να αποτρέψει την εγγραφή κάποιων τεμαχίων, επειδή αυτά θα υπάρχουν ήδη στο σύστημά μας, τα κέρδη σε διεκπεραιωτικότητα (throughput) δεν θα υπερκαλύπουν την αυξημένη καθυστεηση (latency) που θα δημιουργεί η χρονοβόρα αναζήτηση σε ένα ευρετήριο αποτυπωμάτων. Επίσης, τα δεδομένα μας πιθανόν να εμφανίζουν μεγάλη μεταβλητότητα, δηλαδή ένα τεμάχιο στον πρωτεύοντα δίσκο μπορεί να χρειαστεί να γραφτεί πολλές φορές σε μικρό χρονικό διάστημα. Έτσι οι επιδράσεις του αυξημένου latency θα πολλαπλασιαστούν και τα οφέλη της απαλοιφής διπλοτύπων θα είναι μάλλον πενιχρά για δεδομένα που υφίστανται συνεχώς αλλαγές. Στην βιβλιογραφία υπάρχουν πολλές αντιμαχόμενες απόψεις γύρω από την αποδοτικότητα του inline deduplication για πρωτεύουσα αποθήκευση, με προτάσεις για την επιτάχυνση των αναζητήσεων [ZLP08] και συστήματα που την ακολουθούν (πίνακας 3.1). Εμείς υιοθετούμε την offline προσέγγιση, θεωρώντας τα δεδομένα μας ευμετάβλητα και θέλοντας να διατηρήσουμε χαμηλό το latency, ειδικά από την στιγμή που η δενδρική προσέγγισή μας, ήδη επιφέρει μια κάποια αύξηση. Παράλληλα, το inline deduplication απαιτεί

την αδιάλειπτη χρήση πόρων (μνήμη, CPU), οι οποίοι σε ένα περιβάλλον υπολογιστικού νέφους πιθανώς να είναι πολύτιμοι για την λειτουργία του υπόλοιπου συστήματος ή να πρέπει να είναι διαθέσιμοι τουλάχιστον κάποιες στιγμές υψηλού φορτίου. Το offline deduplication από την άλλη έχει την δυνατότητα να απαλείψει τα διπλότυπα οποιαδήποτε χρονική στιγμή και μπορεί να προγραμματιστεί για περιόδους χαμηλού φορτίου όπου οι παραπάνω πόροι θα είναι διαθέσιμοι. Με βάση τα παραπάνω λοιπόν, προτείνουμε offline απαλοιφή διπλοτύπων στα imutable δεδομένα, δηλαδή στα τεμάχια που ανήκουν σε στιγμιότυπα. Μάλιστα μπορούμε να μην περιοριστούμε μόνο στα δεδομένα αλλά να επεκταθούμε και στα μεταδεδομένα, δηλαδή στους κόμβους που ανήκουν σε στιγμιότυπα, μετατρέποντας την δενδρική συνταγή μας σε ένα Merkle tree [Mer87]. Τέλος, εάν έχουμε κάποια πληροφορία για έναν εικονικό δίσκο που αλλάζει μόνο τμηματικά ή εάν υπάρχει υποψία για υψηλή αυτοαναφορά, μπορεί να υποστηριχθεί και offline deduplication σε κάποια υποπεριοχή[4] του πρωτεύοντα δίσκου, προφανώς με κλείδωμα κατά την διάρκεια της απαλοιφής. Τα δικαιώματα εγγραφής των τεμαχίων αυτής της υποπεριοχής θα τεθούν σε μόνο για ανάγνωση και κάθε μετέπειτα προσπάθεια εγγραφής θα πυροδοτήσει COW.

Η διαδικασία της απαλοιφής διπλοτύπων είναι η ακόλουθη:
Κάθε φορά που θα λαμβάνεται ένα στιγμιότυπο, το όνομα του κόμβου-ρίζας του στιγμιοτύπου θα αποθηκεύεται έτσι ώστε αργότερα, σε κάποια στιγμή χαμηλού φορτίου, το σύστημα απαλοιφής διπλοτύπων να γνωρίζει ποια δέντρα είναι immutable και προσφέρονται για απαλοιφή διπλοτύπων. Ο επιπλέον προσωρινός χώρος αποθήκευσης των ονομάτων των ριζικών κόμβων των στιγμιοτύπων θεωρούμε ότι είναι αμελητέος. Εναλλακτικά, μπορούμε να μαρκάρουμε τις επικεφαλίδες συνταγής των στιγμιοτύπων που δεν έχουν υποστεί deduplication. Την βολική στιγμή λοιπόν που θα προγραμματιστεί η απαλοιφή διπλοτύπων σε αυτό το στιγμιότυπο, το σύστημα θα αρχίσει να διασχίζει την δενδρική συνταγή με ανοδικό τρόπο (bottom up). Για κάθε κόμβο/τεμάχιο θα υπολογίζει την τιμή κατακερματισμού των δεδομένων του και θα ελέγχει στο ευρετήριο αποτυπωμάτων εάν το αποτύπωμα υπάρχει και άρα ο κόμβος/τεμάχιο είναι ήδη αποθηκευμένος. Εάν όχι, θα εισάγει το αποτύπωμα μαζί με τη διεύθυνση αποθήκευσης στο ευρετήριο αποτυπωμάτων. Η διεύθυνση αποθήκευσης πιθανότητα είτε θα νοηματοδοτείται από το όνομα του τεμαχίου είτε θα μπορεί να εξαχθεί με βάση αυτό. Επίσης, ανάλογα με τις κατώτερες επιλογές του συ-

[4] ευθυγραμμισμένη προφανώς με το μέγεθος τεμαχίου

στήματος αποθήκευσης μπορεί να είναι βολικό ο κόμβος/τεμάχιο να αποθηκεύεται σε κάποιο άλλο σημείο του συστήματος (chunk store), οπότε τότε θα αποθηκεύεται στο ευρετήριο και η νέα διεύθυνση αποθήκευσης. Είτε όμως το τεμάχιο υπάρχει, είτε όχι, η συνταγή μας θα πρέπει να ενημερώνεται ώστε αντί για το προηγούμενο όνομα του κόμβου/τεμαχίου (το chunk id ή node id που προαναφέραμε) να περιέχει το αποτύπωμα (τιμή κατακερματισμού των δεδομένων του). Έτσι λοιπόν έχουμε το node/chunk id να είναι ένα καθολικό αναγνωριστικό για όσο χρόνο ένας κόμβος/τεμάχιο δεν έχει εισαχθεί στο σύστημα απαλοιφής διπλοτύπων. Για να ενημερωθεί η δενδρική συνταγή, ένας κόμβος θα ξεκινάει την διαδικασία απαλοιφής διπλοτύπων αφού πρώτα όλοι οι κόμβοι/τεμάχια παιδιά του την ολοκληρώσουν. Αυτό είναι σημαντικό ώστε στις εγγραφές του κόμβου-πατέρα να προλάβουν να αντικατασταθούν τα ονόματα των παιδιών του με τα αποτυπώματα τους. Ειδάλλως, ο κόμβος-πατέρας θα γίνει deduplicated έχοντας ως δεδομένα του εγγραφές με node/chunk ids και όχι με αποτυπώματα. Τέτοιου είδους εγγραφές είναι προφανές ότι θα είναι άκυρες, αφού οι κόμβοι/τεμάχια θα είναι προσπελάσιμοι πλέον μέσω των αποτυπωμάτων τους και όχι μέσω των παλιών τους ονομάτων. Γι᾽ αυτό είναι σημαντικό η διάσχιση του δέντρου να γίνει bottom-up. Η αναδρομική αυτή διαδικασία σταματάει στον κόμβο-ρίζα του στιγμιοτύπου, ο οποίος, αφού ενημερωθεί με τα αποτυπώματα των παιδιών της, θα υποστεί deduplication και εν συνεχεία θα ενημερώσει την επικεφαλίδα συνταγής ώστε η τελευταία να δείχνει στον κόμβο-ρίζα όχι με το παλιό όνομά του, αλλά με το νέο αποτύπωμά του. Ως μια τελευταία μικρή λεπτομέρεια, αναφέρουμε ότι πριν από το deduplication ενός κόμβου, καλό είναι και όλα δικαιώματα των εγγραφών του να τεθούν μόνο για ανάγνωση, ώστε δυο κόμβοι με πανομοιότυπα αποτυπώματα παιδιών να μην αποκτήσουν διαφορετικό αποτύπωμα μόνο και μόνο λόγω διαφορετικών δικαιωμάτων. Έτσι κι αλλιώς τα δικαιώματα και των δυο από σημασιολογικής άποψης είναι μόνο για ανάγνωση, απλώς ενδέχεται λόγω του lazy τρόπου εφαρμογής του immutabilty, πρακτικά να καταλήξουν να έχουν διαφορετικά δικαιώματα.

Επιλέχθηκε επίσης ο τεμαχισμός σταθερού μεγέθους σαν αλγόριθμος τεμαχισμού επειδή το σύστημα αρχείων του guest ήδη επενεργεί σε σταθερού μεγέθους μπλοκ. Αυτό προφανώς δεν αναιρεί κάποια πλεονεκτήματα του μεταβλητού τεμαχισμού, όπως η «ανοσία» του σε εισαγωγές δεδομένων σε ενδιάμεσα σημεία ενός τόμου, αλλά θεωρούμε ότι τέτοιες περιπτώσεις είναι περιορισμένες στην πρωτεύουσα αποθήκευση, όπου έχουμε συστήματα αρχείων που διαχειρίζονται δεδομένα σε μπλοκ και αποθη-

κεύουν αρχεία μικρού συνήθως μεγέθους. Τεμάχια χαμηλότερα των 4KB μάλλον δεν θα είναι ιδιαίτερα βοηθητικά, από την στιγμή που έτσι κι αλλιώς το σύστημα αρχείων του guest θα διαχειρίζεται μπλοκ $\geq$4KB. Τα περισσότερα συστήματα απαλοιφής διπλοτύπων για πρωτεύουσα αποθήκευση χρησιμοποιούν τεμάχια σταθερού μεγέθους, ενώ μελέτες [JM09],[JPZ$^+$11] καταδεικνύουν ότι ο τεμαχισμός σε σταθερά μεγέθη μπορεί να οδηγήσει σε εξίσου αποτελεσματική απαλοιφή πλεονασμού. Επιπροσθέτως, μια προσέγγιση τεμαχισμού μεταβλητού μεγέθους απαιτεί την υποστήριξη ενός δυναμικού δέντρου, με δυναμικό αριθμό εγγραφών ανά κόμβο, κάτι που καταλύει την δυνατότητα για εύκολο υπολογισμό του εύρους τομέων που καλύπτει κάθε entry, και εισάγει επιπρόσθετη σχεδιαστική πολυπλοκότητα για την πλοήγηση και διαχείριση του δυναμικού αυτού δέντρου.

Παρά την πρότασή μας, ο σχεδιασμός και η μορφή του εικονικού δίσκου δεν αποκλείει μια inline προσέγγιση που θα χρησιμοποιεί την δενδρική δομή ως συνταγή. Παρόλο που επίσης δεν το προτείνουμε για λόγους απόδοσης, μπορεί να υπάρξει χωρίς τροποποιήσεις byte-to-byte σύγκριση των περιεχομένων ώστε να αποφευχθούν συγκρούσεις των τιμών κατακερματισμού (hash-collisions). Τόσο το εύρος του της απαλοιφής διπλοτύπων, όσο και η επιλογή επιπέδου ασφάλειας για εξασφάλιση ιδιωτικότητας, αφήνονται στην ευχέρεια του εκάστοτε συστήματος. Για κάποια περιβάλλοντα μπορεί να έχει νόημα να εκτελείται τοπική απαλοιφή διπλοτύπων ανά κόμβο, σε περίπτωση που τοποθετούνται με έξυπνο τρόπο παρεμφερείς εικονικοί δίσκοι στον ίδιο host, ενώ κάποια άλλα μπορούν να υποστηρίξουν μια καθολική προσέγγιση, επιλύοντας ζητήματα απαιτήσεων σε μνήμη και ταυτοχρονισμού κατά την προσπέλαση του ευρετηρίου αποτυπωμάτων. Ακόμη, αν δεν υπάρχουν αυστηρές απαιτήσεις ασφαλείας, μπορούμε να έχουμε convergent encryption, όπου κάθε κόμβος/τεμάχιο θα κρυπτογραφείται με βάση το αποτύπωμά του και η πρόσβαση στο στιγμιότυπο ενός χρήστη θα αποτρέπεται με κρυπτογράφηση της επικεφαλίδας συνταγής που περιέχει το αποτύπωμα του κόμβου-ρίζας. Η κρυπτογράφηση αυτή θα είναι η μόνη που θα γίνεται με ένα ιδιωτικό κλειδί μοναδικό για κάθε χρήστη. Δεν θα επεκταθούμε σε ζητήματα της κρυφής μνήμης αποτυπωμάτων (fingerprint cache) μιας και χρήζουν μεγάλης ανάλυσης, αλλά σημειώνουμε ότι το σύστημά μας δεν εισάγει κάποιους περιορισμούς.

Δυνατότητες παραμετροποίησης και επεκτάσεις

Στον παραπάνω σχεδιασμό, ελεύθερες παράμετροι που επιλέγονται από τον χρήστη

του εικονικού δίσκου θεωρούνται το μέγεθος κάθε τεμαχίου και ο αριθμός των επιπέδων της δενδρικής δομής. Σε συνδυασμό με το μέγεθος του δίσκου, καθορίζεται το μέγεθος κάθε κόμβου. Οι εν λόγω επιλογές μπορούν να τεθούν κατάλληλα ώστε να ρυθμίσουμε την επιθυμητή ισορροπία ανάμεσα στην απόδοση και στην εξοικονόμηση χώρου. Περισσότερα για την επίδραση τους στην απόδοση θα δούμε στο κεφάλαιο 5.

Μια ιδιαίτερα χρήσιμη λειτουργικότητα που μπορεί να υλοποιηθεί αποδοτικά με τον παραπάνω σχεδιασμό είναι ο συγχρονισμός με ένα αντίγραφο ασφαλείας του εικονικού δίσκου, σε κάποιον απομακρυσμένο εξυπηρετητή. Θεωρούμε λοιπόν ότι υπάρχουν δυο διαφορετικές εκδοχές, ή αλλιώς δυο στιγμιότυπα, του εικονικού δίσκου σε δυο απομακρυσμένους εξυπηρετητές, οι οποίες διαφέρουν κατά ένα μικρό ποσοστό. Αντί λοιπόν να αποσταλούν όλα τα δεδομένα του εικονικού δίσκου, που μπορεί να είναι δεκάδες gigabyte, αρκεί να αποσταλούν τα αποτυπώματα των τεμαχίων που έχουν μεταβληθεί. Έτσι σε μια επίπεδη συνταγή, θα αποστέλλονταν η συνταγή με τα αποτυπώματα στον απομακρυσμένο εξυπηρετητή, αυτός θα ήλεγχε ποια αποτυπώματα υπάρχουν ήδη και θα αιτούνταν για την αποστολή μόνο των τεμαχίων που έχουν αλλάξει. Η διαδικασία αυτή εξοικονομεί σημαντικό εύρος ζώνης του δικτύου, αλλά και πάλι για ένα μεγάλο δίσκο με μικρό μέγεθος τεμαχίου μπορεί να εμπλέξει την αποστολή πολλών megabyte. Η δενδρική δομή βελτιώνει και αυτήν την διαδικασία, αφού τώρα είναι δυνατόν να αποστέλλεται το αποτύπωμα ενός κόμβου, και, εάν αυτός ο κόμβος υπάρχει, θα γλιτώνουμε την αποστολή ολόκληρου του υποδέντρου που βρίσκεται από κάτω του. Συγκεκριμένα, για κάθε υποδέντρο που αντιπροσωπεύεται από έναν κόμβο του οποίου το αποτύπωμα δεν είναι γνωστό, θα πρέπει να αποσταλούν epn αιτήσεις, μία για κάθε entry. Συνεπώς όσο πιο πολλά είναι τα ιεραρχικά επίπεδα και όσο πιο μικρό το μέγεθος του κόμβου, τόσο λιγότερες αιτήσεις αναμένεται να χρειαστεί να στείλουμε, δεδομένου ότι δεν θα έχουμε τελείως ομοιόμορφες αποκλίσεις αλλά συνήθως οι αλλαγές θα είναι εντοπισμένες και θα μπορούν να συνοψίζονται μέχρι κάποιο επίπεδο, αφήνοντας τους άλλους κόμβους του ίδιου επιπέδου ανεπηρέαστους. Έτσι, αναδύεται η δυνατότητα για σημαντική εξοικονόμηση στην ποσότητας μεταδεδομένων που πρέπει να αποστείλουμε. Περισσότερα για την αποτελεσματικότητα της δενδρικής συνταγής στην εξοικονόμηση εύρους ζώνης δικτύου σε μια τέτοια διεργασία συγχρονισμού μελετάμε στην 5.3.2.

Ο υπάρχων σχεδιασμός δίνει την δυνατότητα και για προσθαφαίρεση επιπέδων σε μια δενδρική συνταγή. Είναι δυνατόν να προσθέσουμε ενα επίπεδο στην αρχή, ως ρίζα

του δέντρου, ώστε να αυξήσουμε το μέγεθος του δίσκου ή να αλλάξουμε την συσχέτιση μεταξύ απόδοσης και εξοικονόμησης χώρου. Αντιστοίχως, αλλά με μεγαλύτερη πολυπλοκότητα και επιβάρυνση, είναι δυνατή η αναδιανομή των δεδομένων και μεταδεδομένων ενός δέντρου σε ένα δέντρο με λιγότερα επίπεδα είτε για συρρίκνωση του εικονικού δίσκου, είτε πάλι για μεταβολή της σχέσης απόδοσης προς εξοικονόμησης χώρου. Υπάρχει επίσης και η δυνατότητα για αντιμετώπιση μιας δενδρικής συνταγής ως μια διαμέριση (partition), με την χρήση της επικεφαλίδας συνταγής ως πίνακα δεικτοδότησης πολλών διαφορετικών διαμερίσεων ενός εικονικού δίσκου. Αυτό μπορεί να επιτρέψει την εύκολη αναδιάταξη των διαμερίσεων στον εικονικό δίσκο, αφού το μόνο που θα αλλάζαμε είναι η σειρά των ονομάτων των κόμβων-ριζών. Μια τέτοια λειτουργικότητα όμως, αποσυνδέει την έννοια του δέντρου συνταγής από την έννοια του εικονικού δίσκου, ενώ η παρεχόμενη λειτουργικότητα αναδιάταξης διαμερίσεων δεν φαίνεται να παρουσιάζει ιδιαίτερη δημοτικότητα.

Στην παραπάνω ανάλυση δεν θίξαμε ζητήματα συλλογής σκουπιδιών και μηχανισμών μέτρησης αναφορών (reference counting). Όταν ένα στιγμιότυπο διαγράφεται, τότε υπάρχει περίπτωση κάποιοι από τους κόμβους/τεμάχια του να μην είναι πλέον προσπελάσιμοι, αφού το στιγμιότυπο έχει διαγραφεί και οι αντίστοιχοι κόμβοι/τεμάχια έχουν γίνει COW για τον πρωτεύοντα δίσκο. Έτσι θεωρούνται «σκουπίδια» και κανονικά θα έπρεπε να αποδεσμευτεί ο χώρος που καταλαμβάνουν. Ο πιο προφανής τρόπος να κάνουμε συλλογή σκουπιδιών θα ήταν να κρατάμε, όπως το qcow2, μια δομή που θα αποθηκεύει τους δείκτες σε κάθε κόμβο/τεμάχιο. Δηλαδή για κάθε κόμβο ή τεμάχιο θα κρατάμε τον συνολικό αριθμό των από κοινού ιδιοκτητών του, ανάλογα με το πόσοι πρωτεύοντες δίσκοι, στιγμιότυπα και κλώνοι μπορούν να τον προσπελάσουν. Κάθε φορά που λαμβάνεται ένα στιγμιότυπο ή ένας κλώνος, οι αριθμοί αναφοράς όλων των τεμαχίων/κόμβων του αρχικού πρωτεύοντος δίσκου/στιγμιοτύπου πρέπει να αυξάνονται κατά ένα, και κάθε φορά που διαγράφεται ένα στιγμιότυπο, όλοι οι αριθμοί αναφοράς των κόμβων/τεμαχίων που ήταν προσπελάσιμοι μέσω του στιγμιοτύπου θα πρέπει να μειώνονται κατά ένα. Αν κατά τη μείωση ο αριθμός φτάσει στο 0, τότε ο κόμβος/τεμάχιο θα διαγράφεται. Επίσης, κάθε φορά που έχουμε COW, θα πρέπει να μειώνεται κατά ένα ο αριθμός αναφοράς του αρχικού κόμβου/τεμαχίου που γίνεται COW. Ακριβώς αυτήν την πολιτική ακολουθεί και το BTRFS. Οι αριθμοί αναφοράς δεν είναι συνετό να τοποθετηθούν πάνω στην δενδρική συνταγή, γιατί αυτό θα λειτουργήσει αποτρεπτικά για την απαλοιφή διπλοτύπων, αφού θα επηρεά-

ζει το αποτύπωμα. Μια νέα δενδρική δομή όμως για όλους τους εικονικούς δίσκους, ίσως να απαιτήσει πολλή μνήμη, και άρα, εάν μεταφερθεί στον δίσκο, θα εισάγει την επιπλέον επιβάρυνση μιας λειτουργίας Ε/Ε για κάθε προσπέλαση κόμβου/τεμαχίου.

Μια εναλλακτική υλοποίηση συλλογής σκουπιδιών θα ήταν το mark and sweep, σύμφωνα με το οποίο ανά τακτά χρονικά διαστήματα εξειδικευμένος κώδικας εκτελεί την εξής διαδικασία. Αρχικά, προσπελάζονται όλοι οι αποθηκευμένοι κόμβοι και τεμάχια που υπάρχουν στο σύστημα και είτε μαρκάρονται με κάποιο αναγνωριστικό είτε αποθηκεύονται τα ονόματά τους σε κάποια λίστα. Αυτό είναι εφικτό μέσω της σειριακής προσπέλασής τους στο φυσικό δίσκο ή στα containers όπου είναι αποθηκευμένοι. Σε δεύτερη φάση, επιχειρείται μια αναδρομική προσπέλαση όλων των κόμβων και των τεμαχίων που είναι προσβάσιμα μέσω όλων των εικονικών δίσκων ή στιγμιοτύπων, δηλαδή πρακτικά μέσω όλων των επικεφαλίδων συνταγής. Αυτοί οι προσπελάσιμοι κόμβοι και τεμάχια μαρκάρονται με διαφορετικό τρόπο ή τα ονόματά τους αποθηκεύονται σε μια διαφορετική λίστα. Όσοι κόμβοι/τεμάχια προσπελάστηκαν στην πρώτη φάση, αλλά όχι στην δεύτερη, σημαίνει ότι είναι μη προσβάσιμοι και άρα άχρηστοι, οπότε και μπορεί να αποδεσμευτεί ο χώρος που αυτοί καταλαμβάνουν. Με περιοδικές εφαρμογές της παραπάνω διαδικασίας, ανακυκλώνεται ο αποθηκευτικός χώρος στο φυσικό δίσκο και αποδεσμεύονται παρωχημένοι κόμβοι και τεμάχια.

3.7 Θεωρητική αξιολόγηση

Εκπλήρωση των στόχων

Ο σχεδιασμός μας υλοποιεί λοιπόν τους στόχους που θέσαμε για τους εξής λόγους:

- Υποστηρίζει sparseness, αφού δεν δεσμεύει εκ των προτέρων χώρο στο φυσικό δίσκο για τους κόμβους και τα τεμάχια, αλλά αυτό γίνεται σταδιακά κατά τις αιτήσεις εγγραφής για τα τεμάχια. Φροντίζει να επιστρέφει διαφανώς μηδενικούς τομείς στις αιτήσεις ανάγνωσης. Η υποστήριξη sparseness για τα δεδομένα και τα μεταδεδομένα του εικονικού δίσκου, δεν εξαρτάται μάλιστα από ένα σύστημα αρχείων που θα μπορεί να υποστηρίζει αραιά αρχεία.
- Υποστηρίζει στιγμιότυπα τα οποία με βάση την τεχνική COW μοιράζονται κοινούς κόμβους και τεμάχια μέχρι αυτοί να χρειαστεί να αλλάξουν. Επιτυγχάνεται

έτσι η απαλοιφή του εσωτερικού χρονικού πλεονασμού σε επίπεδο λεπτομέρειας (granularity) που είναι παραμετροποιήσιμο. Προσφέρει επίσης ακαριαία λήψη στιγμιοτύπων, αφού αντιγράφεται μόνο ο κόμβος-ρίζα του δέντρου-συνταγής. Τα δικαιώματα μόνο για ανάγνωση, εφαρμόζονται αρχικά στον κόμβο-ρίζα και σταδιακά εξαπλώνονται στους κόμβους και στα τεμάχια, ώστε να μην επιβαρύνεται χρονικά η δημιουργία του στιγμιοτύπου. Τα στιγμιότυπα αυτά μπορούν μάλιστα να γίνουν και on-line, όπως θα δούμε στην ενότητα 4.4, αρκεί το σύστημα αρχείων του guest να είναι σε συνεπή κατάσταση (frozen-state) κατά την στιγμή της λήψης τους.

- Υποστηρίζει κλώνους, δημιουργώντας έναν καινούργιο εικονικό δίσκο, που όμως θα μοιράζεται αρχικά τους κόμβους και τα τεμάχια του αρχικού πρωτεύοντα εικονικού δίσκου. Οι αλλαγές θα γίνονται σταδιακά με COW, μόνο σε όσους κόμβους και τεμάχια πρέπει να αλλάξουν. Και εδώ έχουμε απαλοιφή του εσωτερικού χρονικού πλεονασμού σε επίπεδο λεπτομέρειας (granularity) που είναι παραμετροποιήσιμο.
- Υποστηρίζει offline απαλοιφή διπλοτύπων για τους κόμβους/τεμάχια που ανήκουν σε κάποιο στιγμιότυπο και άρα είναι immutable. Έτσι απαλείφουμε τον εσωτερικό χωρικό πλεονασμό και ανάλογα με το πού θα τοποθετηθεί το ευρετήριο αποτυπωμάτων, μπορούμε να έχουμε host-level ή cluster-level απαλοιφή εξωτερικού πλεονασμού. Η απαλοιφή διπλοτύπων είναι διαφανής ως προς την προσπέλαση των κόμβων/τεμαχίων, ενώ υποστηρίζεται και για τους κόμβους (μεταδεδομένα) αλλά και για υποπεριοχές του πρωτεύοντος δίσκου.
- Χρησιμοποιούμε την ίδια δομή αντιστοίχισης (συνταγή) για να υποστηρίξουμε όλες τις λειτουργίες εξοικονόμησης χώρου και άρα εξαλείφουμε και τυχόν πλεονασμό στα μεταδεδομένα.
- Η δενδρική δομή των συνταγών, επιτρέπει ακαριαία δημιουργία στιγμιοτύπων, βοηθάει στην αποδοτικότερη επίτευξη μιας λειτουργίας συγχρονισμού με απομακρυσμένα δεδομένα, οδηγεί σε αραιά μεταδεδομένα που δεσμεύονται κατ' απαίτηση και συμβάλει σημαντικά στην απαλοιφή κοινών μεταδεδομένων μεταξύ των στιγμιοτύπων. Χάρη στην υποστήριξη sparseness, ακαριαία είναι και η δημιουργία πρωτευόντων εικονικών δίσκων.
- Διατηρώντας χαμηλό το ύψος του δέντρου θεωρούμε ότι μπορούμε να βρούμε ισορροπία ανάμεσα στην απόδοση και την εξοικονόμηση χώρου.

Σύγκριση σε σχέση με επίπεδη προσέγγιση

Προαναφέραμε ότι το βασικό πλεονέκτημα σε σχέση με την επίπεδη προσέγγιση είναι πως έχουμε ταχύτερη λήψη στιγμιοτύπων, μιας και αντιγράφεται μόνο ο κόμβος-ρίζα. Αυτός έχει πολύ μικρότερο μέγεθος από τον πίνακα μεταδεδομένων που απαιτείται για την δεικτοδότηση όλων των τεμαχίων και ο οποίος πρέπει να αντιγραφεί στην επίπεδη προσέγγιση.

Ένα ακόμη πλεονέκτημα είναι ότι στην επίπεδη προσέγγιση, x στιγμιότυπα θα καταλάβουν $x * recipe_size$ χώρο, παρόλο που πιθανότατα θα έχουν πολλά τεμάχια που είναι κοινά και άρα πολλά κομμάτια της συνταγής θα είναι κοινά. Με άλλα λόγια, θα υπάρχουν πανομοιότυπα entries στα μεταδεδομένα πολλαπλών συνταγών. Αυτό ισχύει ειδικά στην περίπτωση που λαμβάνονται συχνά στιγμιότυπα. Η δεντρική δομή όμως μας επιτρέπει να εφαρμόσουμε μια μορφή απαλοιφής πλεονασμού, όπου, αν δύο κόμβοι είναι κοινοί, τότε όλο το υποδέντρο από κάτω είναι μοιραζόμενο και δεν υπάρχει αποθηκευμένο πολλές φορές.

Επιπροσθέτως, η δενδρική δομή δίνει την δυνατότητα να εξοικονομηθούν μεταδεδομένα και στην περίπτωση που ο εικονικός δίσκος είναι μερικώς μόνο γεμάτος. Ένας εικονικός δίσκος με επίπεδη δομή, ακόμα και αν έχει δεσμεύσει δεδομένα μόνο στο 20% του συνολικού μεγέθους του, πρέπει να αποθηκεύσει μεταδεδομένα που αναφέρονται σε όλο το μέγεθός του. Αντιθέτως, με μια δενδρική συνταγή, εάν τα δεδομένα αυτά είναι εντοπισμένα σε μια περιοχή του δίσκου, αποθηκεύεται ένα ποσοστό μεταδεδομένων πολύ κοντά στο 20%, αφού δεν θα αποθηκευτούν οι κόμβοι που συνοψίζουν μη δεσμευμένες περιοχές του δίσκου. Έτσι η ιδιότητα του sparseness επεκτείνεται και στα μεταδεδομένα. Ο ισχυρισμός ότι η επίπεδη συνταγή μπορεί να γίνει αραιή βασιζόμενη σε αραιές υλοποιήσεις κάποιου κατωτέρου στρώματος εισάγει αρχιτεκτονικούς περιορισμούς και περιορίζει την μεταφερσιμότητά της.

Επίσης, ένας άλλος στόχος που είχαμε θέσει αρχικά ήταν η πραγματοποίηση ενός γρήγορου συγχρονισμού με ένα απομακρυσμένο αντίγραφο (remote replication). Αυτό στην πραγματικότητα είναι η δυνατότητα να συγχρονίσουμε ένα στιγμιότυπο σε δυο απομακρυσμένους φυσικούς δίσκους, στέλνοντας μόνο όσα δεδομένα χρειάζονται. Στο επίπεδο μοντέλο, αυτό μεταφράζεται στην διάσχιση όλης της συνταγής, στην αποστολή όλων των αποτυπωμάτων, τα οποία είναι τόσα όσα και τα τεμάχια, και, αφού ελεγχθεί αν το αποτύπωμα υπάρχει ή όχι στον απομακρυσμένο host, αποστέλ-

λονται τα τεμάχια που λείπουν.
Στο δεντρικό μοντέλο όμως, μπορούμε να εξοικονομήσουμε εύρος ζώνης δικτύου, αφού πιθανότατα θα χρειαστεί να στείλουμε λιγότερα αποτυπώματα. Αυτό γιατί αν το αποτύπωμα ενός ενδιάμεσου κόμβου υπάρχει, τότε δεν θα χρειαστεί να στείλουμε κανένα από τα αποτυπώματα του υποδέντρου που βρίσκεται από κάτω του. Βέβαια αυτό προϋποθέτει μιας μορφής «συνομιλία» ανάμεσα στις διεργασίες των δυο host που εκτελούν τον απομακρυσμένο συγχρονισμό, και η απόκριση για το εάν ένα αποτύπωμα υπάρχει στον απομακρυσμένο host, οπότε και δεν χρειάζεται να σταλούν τα αποτυπώματα των υποδέντρων, είναι πιθανόν να εισάγει μια επιπλέον καθυστέρηση στην όλη διαδικασία.

Ένα άλλο πλεονέκτημα της δεντρικής αυτής δομής είναι ότι μπορεί να επεκτείνει την εφαρμογή της απαλοιφής διπλοτύπων στην συνταγή. Στο επίπεδο μοντέλο, η συνταγή δεν υπόκειται στο σύστημα απαλοιφής διπλοτύπων. Αυτό μπορεί να γίνει σε μια δενδρική δομή, εάν στα στιγμιότυπα πέρα από τα τεμάχια υπολογίζουμε τα αποτυπώματα και των κόμβων και τους υποβάλλουμε στο ίδιο σύστημα απαλοιφής διπλοτύπων (merkle tree). Αυτό βέβαια δεν θα αυξήσει πολύ το deduplication ratio, αφού η πιθανότητα δυο κόμβοι να έχουν το ίδιο αποτύπωμα (και άρα να δείχνουν στο ίδιο περιεχόμενο είτε είναι τεμάχια είτε άλλοι κόμβοι) χωρίς να έχουν κάποια χρονική σχέση μεταξύ τους (να ανήκουν δηλαδή σε στιγμιότυπα του ίδιου δίσκου) είναι μικρή. Η πιθανότητα να έχουν το ίδιο αποτύπωμα όντας κομμάτια διαφορετικών στιγμιοτύπων του ίδιου δίσκου είναι αξιόλογη αλλά έχουμε ήδη απαλείψει αυτήν την μορφή πλεονασμού με την τεχνική COW που εφαρμόζουν τα στιγμιότυπα. Ίσως όμως να καταφέρουμε να επιτύχουμε κάποιο μικρό inter-deduplication όταν σε δυο διαφορετικούς δίσκους υπάρχει μια αλληλουχία κάμποσων συνεχόμενων τεμαχίων που είναι κοινά (π.χ. λόγω κώδικα του λειτουργικού συστήματος ή λόγω κοινού μεγάλου αρχείου). Ο ισχυρισμός ότι και η επίπεδη συνταγή μπορεί να υποστεί απαλοιφή διπλοτύπων με τον χωρισμό της σε μπλοκ, συνεπάγεται την δημιουργία μιας δεύτερης συνταγής που θα κρατάει την αλληλουχία των αποτυπωμάτων αυτών των μπλοκ. Αυτή η δεύτερη συνταγή ουσιαστικά ισοδυναμεί με τον κόμβο-ρίζα μιας διεπίπεδης δενδρικής δομής.

Ένα πρώτο μειονέκτημα της δεντρικής δομής είναι το ότι αυξάνεται η επιβάρυνση σε μεταδεδομένα, αφού για να δεικτοδοτηθούν όλα τα τεμάχια, οι κόμβοι του τελευταίου επιπέδου θα καταλάβουν χώρο τουλάχιστον ίσο με αυτό της επίπεδης συνταγής και θα έχουμε και την επιπρόσθετη επιβάρυνση όλων των επιπλέον ενδιάμεσων κόμβων.

Η συνολική επιβάρυνση είναι:

$$node_size * \frac{1 - total_chunks}{1 - total_chunks^{\frac{1}{levels}}}$$

και η επιπρόσθετη σε σχέση με την επίπεδη προσέγγιση δίνεται με τον ίδιο τύπο αλλά με $levels - 1$ αντί για $levels$.
Βλέπουμε λοιπόν ότι ανάλογα με το μέγεθος του δίσκου πρέπει να περιοριστούμε στον αριθμό των επιπέδων για να μην έχουμε απότομη αύξηση των μεταδεδομένων (metadata explosion).

Το βασικό μειονέκτημα της δεντρικής προσέγγισης όμως, είναι ότι τα επιπλέον επίπεδα που προσθέτουμε, προσθέτουν επιπλέον λειτουργίες Ε/Ε, για διάβασμα ή και γράψιμο των ενδιάμεσων κόμβων, οπότε αυξάνουν το latency μιας αίτησης.

Άρα συνοψίζοντας τα πλεονεκτήματα και τα μειονεκτήματα της δεντρικής δομής έχουμε:

+ ταχύτερη λήψη στιγμιοτύπου
+ εξοικονόμηση χώρου για μια σειρά από στιγμιότυπα που εμφανίζουν πλεονασμό στη συνταγή
+ εξοικονόμηση μεταδεδομένων για μερικώς γεμάτους δίσκους
+ πιο οικονομικό συγχρονισμό των στιγμιοτύπων σε απομακρυσμένους host
+ απαλοιφή διπλοτύπων για τους κόμβους
- αύξηση του latency των λειτουργιών Ε/Ε
- αύξηση των μεταδεδομένων για γεμάτο δίσκο

Συνεπώς λόγω των πλεονεκτημάτων της, η παραπάνω δομή μας επιτρέπει να διατηρήσουμε χαμηλό το μέγεθος τεμαχίου, κάτι που απαιτείται για υψηλό deduplication ratio και χαμηλό χρόνο αναζήτησης στο ευρετήριο αποτυπωμάτων, χωρίς να θυσιάσουμε τον χρόνο λήψης στιγμιοτύπου ή να φοβόμαστε για απότομη αύξηση των μεταδεδομένων μετά από συχνά στιγμιότυπα.

Πλεονεκτήματα σε σχέση με υπάρχουσες υλοποιήσεις

Ο σχεδιασμός μας προσφέρει τα παρακάτω σε σχέση με προϋπάρχουσες λύσεις:

- ανεξαρτησία από τον σχεδιασμό των κατώτερων επιπέδων στο storage stack του περιβάλλοντός μας. Μπορεί να απαγκιστρωθεί τόσο από συγκεκριμένα συστήματα

αρχείων όσο και από την ίδια την ύπαρξη ενός συστήματος αρχείων, προσαρμόζοντας τις οντότητες που έχουμε εισάγει και υλοποιώντας κατάλληλους οδηγούς μπλοκ για το εκάστοτε μοντέλο αποθήκευσης. Απαγκιστρωνόμαστε λοιπόν και από συγκεκριμένες τεχνολογίες και από απαιτήσεις για συγκεκριμένο αρχιτεκτονικό σχεδιασμό.

- αυτόνομες ευκίνητες οντότητες. Ο διαχωρισμός και η αυτοτέλεια των οντοτήτων σε συνδυασμό με το μικρό μέγεθός τους, συνεπάγεται ευελιξία και ευνοεί την μεταφερσιμότητά τους. Αυτό είναι επιθυμητό σε ένα περιβάλλον υπολογιστικού νέφους στο οποίο οι πόροι δυναμικά μεταβάλλονται και ο σχεδιασμός είναι κατανεμημένος.
- κλιμακωσιμότητα σε ένα περιβάλλον υπολογιστικού νέφους πολλών εκατοντάδων κόμβων, αφού δεν περιορίζεται στο απλοϊκό σενάριο ενός host που αποθηκεύει τους εικονικούς δίσκους ως αρχεία σε ένα σύστημα αρχείων, σχεδιασμός που αναπόφευκτα εμφανίζει εξαρτήσεις απόδοσης και σημεία συμφόρησης. Χάρη στον τεμαχισμό δεδομένων και μεταδεδομένων, οι οντότητες όλων των εικονικών δίσκων μπορούν να είναι ανεξαρτήτως διαχειρίσιμες και άρα να κλιμακώνουν μαζί με τις δυνατότητες του συστήματος φυσικής αποθήκευσης. Λόγω της αυτονομίας και μεταφερσιμότητάς τους, οι οντότητες μπορούν να είναι κατανεμημένες και να εκμεταλλεύονται έξυπνους αλγορίθμους του περιβάλλοντος υπολογιστικού νέφους, οι οποίοι για βελτιστοποίηση θα τους τοποθετούν στα κατάλληλα σημεία φυσικής αποθήκευσης. Έτσι είναι δυνατόν το σύστημα να καταλύει σημεία συμφόρησης που θέτουν φράγματα στην απόδοση.
- ένα ενιαίο διαχειριστικό πλαίσιο των λειτουργιών εξοικονόμησης χώρου, πάνω στο οποίο μπορούμε να έχουμε καλύτερη εποπτεία για τις επιπτώσεις της εκάστοτε παραμετροποίησης των μεταβλητών, και το οποίο διευκολύνει την ρύθμισή τους.
- απαλοιφή πλεονασμού στα μεταδεδομένα. Πολλές διαχειριστικές δομές που απαιτούνται για τις διαφορετικές τεχνικές εξοικονόμησης χώρου είναι παρεμφερείς, οπότε η ενοποίηση όλων των λειτουργιών σε ένα επίπεδο με μια ενιαία δομή αντιστοίχισης, αποφεύγει την πιθανότητα επανάληψής τους σε διαφορετικά επίπεδα και δημιουργεί πρόσφορο έδαφος για τεχνικές εξοικονόμησης χώρου στα μεταδεδομένα.
- ταχύτητα, αφού αντίθετα με την διαδεδομένη προσέγγιση της αλυσίδας αυξητικών στιγμιοτύπων, αποφεύγει την διάσχιση πολλαπλών δομών για την σύνθεση του πρωτεύοντος δίσκου και απαλλάσσει την λειτουργία διαγραφής ενός στιγμιοτύπου από την ανάγκη για συγχώνευση των αλλαγών.

3.8 Μοντέλο Απόδοσης

Εδώ θα προσπαθήσουμε να καταστρώσουμε ένα μοντέλο που θα σκιαγραφεί την απόδοση την δενδρικής μορφής του εικονικού δίσκου, ως προς τις λειτουργίες Ε/Ε που απαιτούνται. Όπως προαναφέραμε, οι ελεύθερες παράμετροι της δενδρικής δομής μας είναι το μέγεθος των τεμαχίων και το ύψος του δέντρου.

Αρχικά λαμβάνουμε την περίπτωση των αιτήσεων ανάγνωσης. Έχουμε τριών ειδών λειτουργίες Ε/Ε: τις λειτουργίες εξαιτίας των τεμαχίων, αυτές εξαιτίας του ύψους του δέντρου και τις λειτουργίες προσπέλασης επιπλέον κόμβων, ανάλογα με το μέγεθος της αίτησης και την δομή του εκάστοτε δέντρου.

Αρχικά έχουμε λειτουργίες Ε/Ε που απαιτούνται για την προσπέλαση των καθαρών δεδομένων του εικονικού δίσκου, δηλαδή την προσπέλαση των τεμαχίων. Ο αριθμός των λειτουργιών Ε/Ε που απαιτούνται σε μια αίτηση του guest, είναι ίσος με των αριθμό των τεμαχίων που πρέπει να προσπελαστούν και δίνεται από τον: $\lceil \frac{request_size}{chunksize} \rceil$. Αν θέλουμε να πάρουμε την μέση περίπτωση, πρέπει να προσθέσουμε +0.5, αφού κατά μέσο όρο, ο πρώτος τομέας της αίτησης θα βρίσκεται στο μέσο του τεμαχίου. Ο παραπάνω τύπος εξαρτάται αντιστρόφως ανάλογα μόνο από το μέγεθος του τεμαχίου, οπότε ο διπλασιασμός του μεγέθους του τεμαχίου θα επιφέρει υποδιπλασιασμό των απαιτούμενων λειτουργιών Ε/Ε. Αυτό προφανώς μέχρι το σημείο που το μέγεθος της αίτησης είναι μεγαλύτερο από το μέγεθος του τεμαχίου.

Επιπροσθέτως, υπάρχουν οι λειτουργίες Ε/Ε που εισάγονται σε μια αίτηση του guest εξαιτίας του ύψους του δέντρου, καθώς για την τελική προσπέλαση των τεμαχίων, πρέπει να διασχιστεί ένα μονοπάτι κόμβων από την ρίζα μέχρι τον τελικό κόμβο-φύλλο. Προφανώς η σχέση είναι αμιγώς γραμμική ως προς το ύψος του δέντρου και επηρεάζει τον χρόνο απόκρισης (latency) μιας αίτησης.

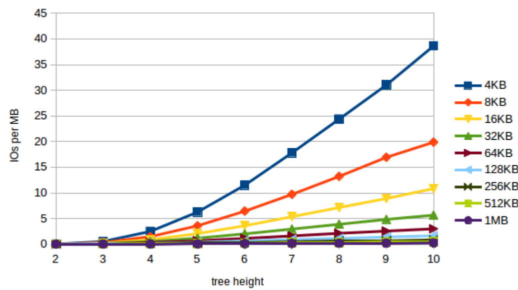
Η τρίτη κατηγορία προσπελάσεων εμπερικλείει και πάλι προσπέλαση κόμβων (μεταδεδομένων), αλλά αυτή τη φορά αναφέρεται όχι μόνο στις απαραίτητες προσπελάσεις σε μια στοιχειώδη αίτηση, αλλά στις προσπελάσεις επιπλέον κόμβων που μπορεί να απαιτηθούν εάν το μέγεθος της αίτησης είναι μεγάλο. Στην περίπτωση αυτή στο τελευταίο επίπεδο απαιτούνται περισσότερα τεμάχια απ' όσα μπορεί να δεικτοδοτήσει ένας κόμβος-φύλλο, οπότε πρέπει να φορτωθούν και διπλανοί κόμβοι-φύλλα. Αυτή η λογική επεκτείνεται αναδρομικά και για εσωτερικούς κόμβους μέχρι την ρίζα. Εδώ

έχουμε μια ιδιαίτερα πολύπλοκη μαθηματική σχέση για την εύρεση των λειτουργιών Ε/Ε για τους επιπλέον κόμβους που απαιτούνται, και η οποία εμπλέκει το μέγεθος του τεμαχίου, το ύψος του δέντρου αλλά και το μέγεθος του δίσκου, πέρα από το μέγεθος της εκάστοτε αίτησης. Η σχέση αυτή παρουσιάζεται στην καλύτερή της περίπτωση -όταν και η αίτηση αναφέρεται στο πρώτο entry κάθε κόμβου κάθε επιπέδου- και στηρίζεται στην λογική της προσπέλασης των απαιτούμενων entries για την δεικτοδότηση των επιθυμητών τεμαχίων.

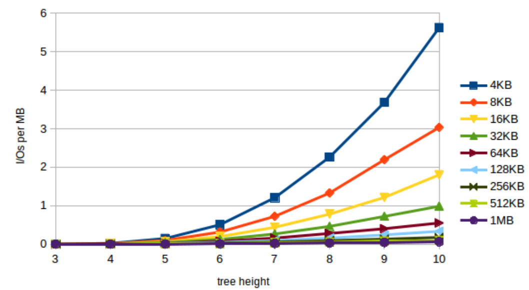
$$\sum_{k=1}^{h-1} \left\lceil \frac{request_size}{epn^k * chunksize} \right\rceil \qquad \text{όπου h είναι το ύψος του δένρου}$$

Σε αδρές γραμμές, όσο μειώνεται το μέγεθος του τεμαχίου τόσο αυξάνονται τα entries που απαιτούνται για την δεικτοδότηση τεμαχίων στο τελευταίο επίπεδο και άρα τόσο αυξάνονται οι κόμβοι-φύλλα και άρα επαγωγικά οι κόμβοι του δευτέρου επιπέδου για την δεικτοδότηση των κόμβων-φύλλων, και ούτω καθεξής μέχρι την ρίζα. Παράλληλα, όπως φαίνεται από την σχέση της ενότητας 3.6, η αύξηση του ύψους του δέντρου, οδηγεί σε μείωση των epn, μείωση που βέβαια δεν είναι γραμμική. Με σταθερό το μέγεθος του δίσκου και το μέγεθος των τεμαχίων, ο αριθμός των τεμαχίων και άρα ο αριθμός των απαιτούμενων entries για την δεικτοδότηση των τεμαχίων δεν αλλάζει, οπότε αναπόφευκτα αφού οι κόμβοι έχουν μικρότερο μέγεθος, αυξάνεται ο αριθμός τους. Αυτό ισχύει επαγωγικά και για τα ανώτερα επίπεδα, αφού αφ᾽ ενός η μείωση του epn για το ψηλότερο δέντρο αυξάνει τον αριθμό των απαιτούμενων κόμβων ανωτέρου επιπέδου και αφ᾽ ετέρου η αύξηση του αριθμού των κόμβων του κατώτερου επιπέδου επιφέρει αυξημένες ανάγκες δεικτοδότησης στο ανώτερο επίπεδο. Συνεπώς, για ένα ψηλότερο δέντρο απαιτείται η προσπέλαση περισσότερων κόμβων για το ίδιο εύρος τομέων. Παρατηρούμε λοιπόν ότι η μείωση του μεγέθους του τεμαχίου ή η αύξηση του ύψους του δέντρου επιφέρουν αύξηση των επιπλέον κόμβων που ενδέχεται να προσπελαστούν σε μια αίτηση.

Επειδή η οπτικοποίηση της παραπάνω σχέσης καθίσταται δύσκολη, θα παρουσιάσουμε τις επιπλέον λειτουργίες Ε/Ε που απαιτούνται ανά MB μιας αίτησης, για προσπέλαση επιπλέον κόμβων στα επίπεδα 1 και 2. Εξάγουμε τον μέσο όρο των λειτουργιών ανά MB, για όλα τα μεγέθη δίσκων από 1GB έως και 1TB, και αφήνουμε ως ελεύθερες παραμέτρους το μέγεθος του τεμαχίου και το ύψος του δέντρου.



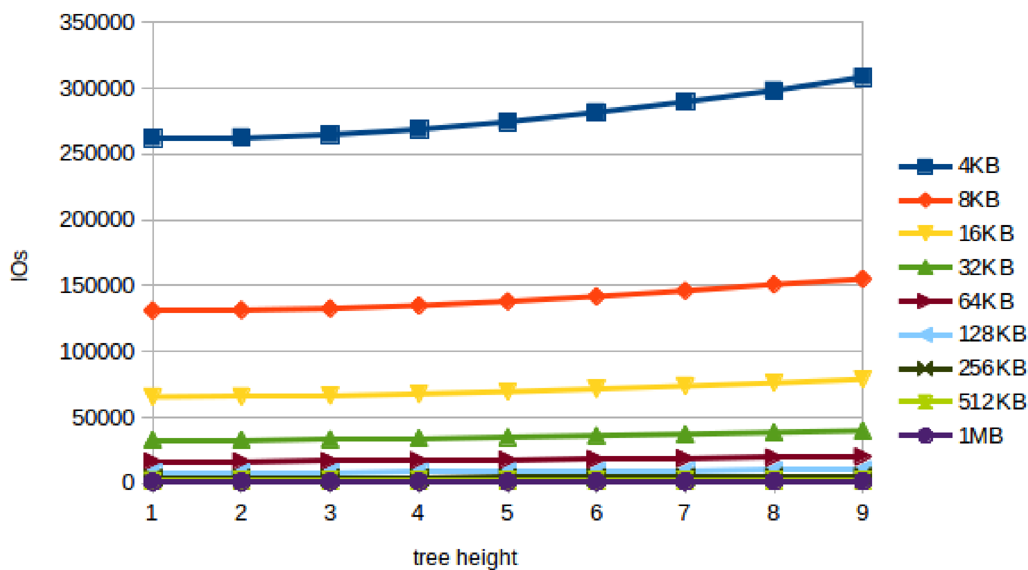
Σχήμα 3.9: *Επιπλέον κόμβοι/λειτουργίες Ε/Ε ανά MB στο 1ο επίπεδο*



Σχήμα 3.10: *Επιπλέον κόμβοι/λειτουργίες Ε/Ε ανά MB στο 2ο επίπεδο*

Από τα διαγράμματα 3.9 και 3.10 για τα επίπεδα 1 και 2, μπορούμε να παρατηρήσουμε τον ρυθμό αύξησης των λειτουργιών Ε/Ε ανά MB, σε σχέση με την μείωση του μεγέθους των τεμαχίων και την αύξηση του ύψους του δέντρου. Αντίστοιχα διαγράμματα προκύπτουν και για την προσπέλαση των κόμβων για υψηλότερα επίπεδα.

Το παραπάνω μοντέλο που παρουσιάσαμε αναφέρεται στις λειτουργίες που απαιτούνται για μια μεμονωμένη αίτηση. Μια πιο ρεαλιστική ανάλυση μπορεί να μελετήσει το σύνολο των λειτουργιών Ε/Ε που απαιτούνται για μια αλληλουχία αιτήσεων του guest, που αναφέρονται σε ένα πεπερασμένο χρονικό παράθυρο και ένα αντανακλούν ένα φορτίο εργασίας. Αυτό είναι μάλλον ενδεικτικότερο καθώς μετά την πρώτη προσπέλασή τους, οι κόμβοι και τα τεμάχια διατηρούνται στην μνήμη της host page cache, οπότε κάθε χρονικά κοντινή προσπέλασή τους δεν θα πυροδοτήσει εκ νέου πρόσβαση στο δίσκο. Συνεπώς, ρόλο παίζει και το μοτίβο προσπέλασης. Για διασκορπισμένες, τυχαίες αιτήσεις μικρού μεγέθους, ο αριθμός λειτουργιών Ε/Ε κυριαρχείται από την δεύτερη κατηγορία λειτουργιών, αυτήν που εξαρτάται αμιγώς από το ύψος του δέντρου. Αντιθέτως, αν έχουμε εντοπισμένες, σειριακές αιτήσεις μεγάλου μεγέθους, τότε ο κύριος όγκος των λειτουργιών Ε/Ε οφείλεται στην προσπέλαση των τεμαχίων και άρα εξαρτάται από το μέγεθος του τεμαχίου. Για αιτήσεις που καλύπτουν μεγάλη περιοχή του δίσκου επιδρούν δευτερευόντως της προσπέλασης τεμαχίων και οι λειτουργίες προσπέλασης επιπλέον κόμβων. Όλα τα παραπάνω θα μελετηθούν εκτενέστερα στο κεφάλαιο της πειραματικής αξιολόγησης, αλλά για να έχουμε μια αίσθηση των συσχετισμών, παρουσιάζουμε τον αριθμό λειτουργιών Ε/Ε για την σειριακή προσπέλαση 1GB, κάτι που δεν εξαρτάται από τον αριθμό των αιτήσεων του guest στις οποίες αυτή η προσπέλαση πραγματοποιείται, αφού οι κόμβοι και τα τεμάχια φορτώνονται στην host page cache.

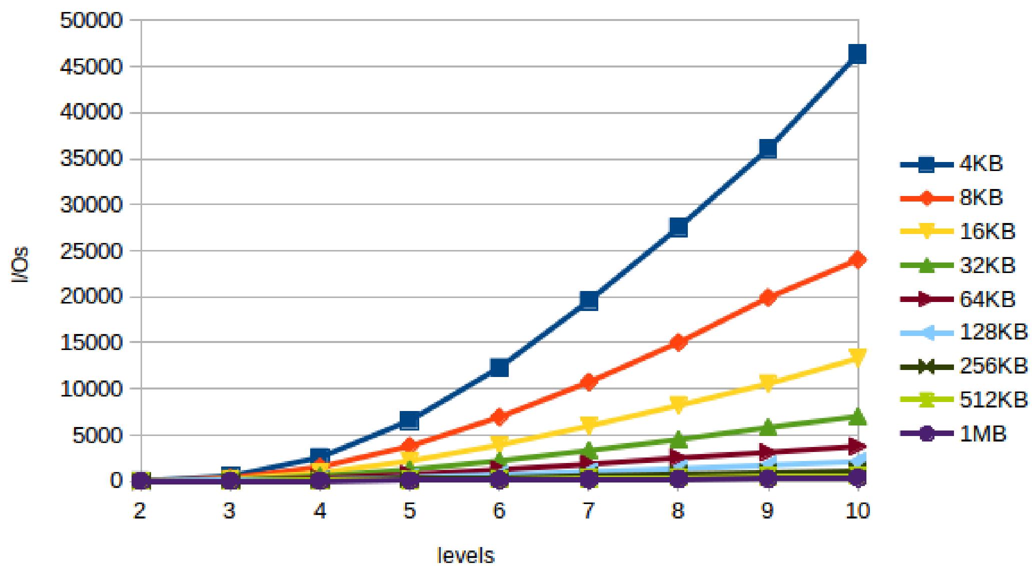


Σχήμα 3.11: *Συνολικές λειτουργίες Ε/Ε για την προσπέλαση 1GB, συνυπολογίζοντας τις προσπελάσεις των τεμαχίων*

Από τα σχήματα 3.11 και 3.12 συμπεραίνουμε ότι ο αριθμός των λειτουργιών Ε/Ε κυριαρχείται σε συντριπτικό βαθμό από τις προσπελάσεις των τεμαχίων, ειδικά για χαμηλά δέντρα ή μικρά τεμάχια, όπως φαίνεται και από τις πολύ ελαφριές κλίσεις των καμπυλών του διαγράμματος 3.11. Μέχρι και δέντρου ύψος 7 το ποσοστό των λειτουργιών που δεν οφείλονται σε τεμάχια αλλά σε επιπλέον κόμβους είναι μικρότερο από 15%, και για δέντρα με ύψος μικρότερο του 5 και τεμάχια μικρότερα ή ίσα των 128KB το ποσοστό είναι μικρότερο από 2%.

Προφανώς, σε περίπτωση που έχουμε μη δεσμευμένα entries (sparse λειτουργία) και όταν επιστρέφονται διαφανώς μηδενικά, απαλείφονται οι λειτουργίες που αφορούν το υποδέντρο κάτω από το μηδενικό entry.

Οι αιτήσεις εγγραφής ακολουθούν την ίδια μοντελοποίηση με αυτές της ανάγνωσης, αφού απαιτείται ακριβώς η ίδια διάσχιση του δέντρου. Η βασική διαφορά είναι ότι τώρα ενδέχεται να χρειαστεί COW ενός κόμβου ή ενός τεμαχίου. Η διενέργεια του COW για ένα κόμβο απαιτεί 2 λειτουργίες Ε/Ε για την αντιγραφή του κόμβου -μία ανάγνωσης και μια εγγραφής- αλλά και άλλη μία λειτουργία Ε/Ε για την ενημέρωση των δικαιωμάτων του πατρικού κόμβου. Θεωρούμε ότι η λειτουργία ενημέρωσης του πατρικού κόμβου μπορεί να συνοψίσει τις ενημερώσεις πολλών κόμβων-παιδιών. Το COW στα τεμάχια μπορεί να πραγματοποιηθεί με την μικρή βελτιστοποίηση αποφυ-



Σχήμα 3.12: *Συνολικές λειτουργίες Ε/Ε για την προσπέλαση 1GB, χωρίς τις προσπελάσεις των τεμαχίων*

γής της λειτουργίας ανάγνωσης, στην περίπτωση που πρόκειται να γραφεί ολόκληρο το τεμάχιο. Συνεπώς, στην χειρότερη περίπτωση που όλοι οι κόμβοι και τα τεμάχια πρέπει να γίνουν COW, μια αίτηση εγγραφής μπορεί να απαιτήσει τις τριπλάσιες λειτουργίες από μια αίτηση ανάγνωσης για τους κόμβους, και τις διπλάσιες για τα τεμάχια (γιατί τα τεμάχια δεν έχουν την επιπλέον λειτουργία ενημέρωσης δικαιωμάτων που έχουν οι κόμβοι, αλλά έχουν μόνο τις λειτουργίες της αντιγραφής). Ανάλογα με την υλοποίηση είναι πιθανόν να εισάγονται και επιπλέον λειτουργίες που αφορούν την δέσμευση του νέου κόμβου/τεμαχίου, όπως για παράδειγμα λειτουργίες επί των δομών ενός συστήματος αρχείων ή ενός ευρετηρίου στο δίσκο.

Εφόσον το COW περιέχει τόσο την δέσμευση νέων τεμαχίων και κόμβων όσο και την ανάγνωση από τους παλιούς και την ενημέρωση του γονεϊκού κόμβου, συμπεραίνουμε ότι η περίπτωση πρώτης εγγραφής σε ένα κενό δίσκο (καθαρός δίσκος) είναι πιο γρήγορη από την COW περίπτωση.

Για να ολοκληρώσουμε την ανάλυσή μας αναφέρουμε ότι η λήψη ενός στιγμιοτύπου απαιτεί 2 λειτουργίες Ε/Ε για την αντιγραφή του κόμβου-ρίζας -μία ανάγνωσης και μία εγγραφής-, μια λειτουργία εγγραφής για την ενημέρωση των δικαιωμάτων του αρχικού κόμβου-ρίζας και άλλη μία λειτουργία εγγραφής για την δημιουργία της επικεφαλίδας συνταγής του στιγμιοτύπου. Εφόσον προείπαμε ότι οι κλώνοι δημιουρ-

γούνται μόνο από στιγμιότυπα, για την δημιουργία ενός κλώνου απαιτούνται μόνο 2 λειτουργίες Ε/Ε για την αντιγραφή του κόμβου-ρίζας και μία για την δημιουργία της επικεφαλίδας-συνταγής του κλώνου. Αυτό γιατί σε αντίθεση με τα στιγμιότυπα δεν απαιτείται η ενημέρωση των δικαιωμάτων του κόμβου-ρίζας του στιγμιοτύπου, αφού αυτά είναι ήδη όλα μηδενικά. Για την αποκατάσταση ενός εικονικού δίσκου με ένα στιγμιότυπο, απαιτούνται 2 λειτουργίες Ε/Ε, μία για την ανάγνωση των δεδομένων του κόμβου-ρίζας και μία για την εγγραφή τους στον κόμβο-ρίζα του βασικού εικονικού δίσκου.

Για μια πιο ακριβή μοντελοποίηση της απόδοσης, στην παραπάνω ανάλυση οφείλουμε να προσθέσουμε και άλλες λειτουργίες που εισάγονται από τις ανάγκες του σχήματος υλοποίησής μας. Παρακάτω αναλύουμε κάποιες τέτοιες λειτουργίες στην περίπτωση της υλοποίησής μας. Τέτοιες είναι οι λειτουργίες Ε/Ε για την ανάγνωση των καταλόγων, οι οποίες αυξάνονται με την αύξηση του αριθμού των τεμαχίων και κόμβων, και άρα με την αύξηση του μεγέθους τεμαχίου, του ύψους του δέντρου και του μεγέθους του δίσκου. Επίσης, στις περιπτώσεις εγγραφής έχουμε λειτουργίες για την ενημέρωση του inode table (χρόνος τελευταίας μεταβολής), ενώ κατά την πρώτη δέσμευση ενός κόμβου ή τεμαχίου έχουμε και επιπλέον λειτουργίες Ε/Ε για ενημέρωση bitmaps και superblock.

Στην παρούσα υλοποίηση, όπως θα φανεί στη συνέχεια, σε μια αίτηση εγγραφής, πρέπει για κάθε κόμβο που έχει παιδί που απαιτεί COW, να διαβάσουμε μια φορά την επικεφαλίδα συνταγής του πρωτεύοντος δίσκου ώστε να μπορούμε να προσφέρουμε υποστήριξη για online στιγμιότυπα. Για τον ίδιο λόγο, η λήψη ενός στιγμιοτύπου ή ενός κλώνου συμπεριλαμβάνει και μια λειτουργία εγγραφής για την ενημέρωση της επικεφαλίδας συνταγής του πρωτεύοντος δίσκου.

Ακόμη, δεν θα πρέπει να παραβλέπουμε το γεγονός ότι ένα σύστημα αρχείων μπορεί να διατηρεί ή να διαταράσσει την τοπικότητα, τοποθετώντας συνεχόμενα τεμάχια σε συνεχόμενες ή μη θέσεις μνήμης. Έτσι μια σειριακή λειτουργία Ε/Ε για μια περιοχή του εικονικού δίσκου x τεμαχίων, μπορεί στον host να μεταφράζεται σε 1 έως x λειτουργίες Ε/Ε, ανάλογα με τον τρόπο χωροθέτησης των τεμαχίων από το σύστημα αρχείων.

Σε όλη την παραπάνω ανάλυση, επικεντρωνόμαστε στον αριθμό των λειτουργιών Ε/Ε που απαιτούνται, αλλά δεν εστιάζουμε τόσο στο μέγεθος κάθε τέτοιας λειτουργίας. Αυτό είναι εσκεμμένο, μιας που για εικονικό δίσκο έως και 1TB, το μέγεθος του μεγαλύτερου δυνατού κόμβου και άρα το μέγιστο μέγεθος μιας λειτουργίας Ε/Ε είναι

360KB. Συνήθως μάλιστα τόσο μεγάλες προσπελάσεις απαιτούνται σε σειριακά και όχι τυχαία μοτίβα Ε/Ε και κυρίως για τους κόμβους-φύλλα. Παρομοίως το μέγεθος μιας λειτουργίας Ε/Ε τεμαχίου φράσσεται από το μέγεθος του τεμαχίου, που στην εργασία μας είναι συνήθως μικρό (συνήθως όχι μεγαλύτερο του 128KB). Συνεπώς, θεωρούμε ότι το μέγεθος των λειτουργιών είναι σχετικά μικρό και στον χρόνο περάτωσης κυριαρχεί ο χρόνος έκδοσης, επεξεργασίας και εξυπηρέτησης ενός αιτήματος και όχι ο χρόνος μεταφοράς των δεδομένων. Αυτό τόσο σε επίπεδο λειτουργικού συστήματος, όσο και σε επίπεδο δίσκου, με τον χρόνο τοποθέτησης της κεφαλής να κυριαρχεί συντριπτικά του χρόνου μεταφοράς των δεδομένων από την σειριακή ανάγνωση/εγγραφή τους. Επομένως, προκύπτει η ότι στο άνωθεν μοντέλο απόδοσης μας ενδιαφέρει κυρίως ο αριθμός των κόμβων και τεμαχίων που προσπελάζονται, αλλά για σχετικά μεγάλα τεμάχια (>32KB), αναμένουμε το μέγεθος της λειτουργίας να έχει κάποια επίδραση στην απόδοση. Έτσι, ιδίως για μεγάλα τεμάχια, ο υποδιπλασιασμός του μεγέθους των τεμαχίων οδηγεί σε μια απόδοση χειρότερη από την διπλάσια, μιας που αυξάνεται και το μέγεθος των λειτουργιών Ε/Ε σε μη αμελητέο βαθμό.

Από την ανάλυσή μας παραβλέπουμε πτυχές όπως ο χρόνος που δαπανάται στις κλήσεις συστήματος ή ο χρόνος μεταφοράς από την host page cache στους buffers του οδηγού.

4 Υλοποίηση οδηγού εικονικού δίσκου

Εδώ θα περιγράψουμε μια υλοποίηση της παραπάνω μορφής εικονικού δίσκου που αναλύσαμε στο κεφάλαιο του σχεδιασμού. Η υλοποίηση αυτή πραγματώνεται με την χρήση ενός οδηγού εικονικού δίσκου (virtual disk block driver) που μεταχειρίζεται κάθε θεμελιώδη οντότητα του συστήματός μας ως αρχείο στο σύστημα αρχείων του host.

4.1 Πυρήνας της υλοποίησης: οντότητες ως αρχεία του συστήματος αρχείων

Για να μπορέσουμε να ποσοτικοποιήσουμε τις επιπτώσεις στην απόδοση, επιλέγουμε να υλοποιήσουμε ένα απλό σύστημα που θα υιοθετεί τον παραπάνω σχεδιασμό. Σε αυτό το σύστημα επιλέγουμε κάθε στοιχειώδης οντότητα που απαρτίζει τον εικονικό δίσκο να αποτελεί ένα αρχείο στο σύστημα αρχείων του host. Οι στοιχειώδεις οντότητες είναι τα τεμάχια, οι κόμβοι και οι επικεφαλίδες συνταγής.

Έτσι, τα ονόματα των τεμαχίων και των κόμβων, που όπως προείπαμε συνίστανται σε chunk ids, node ids ή αποτυπώματα (τιμές κατακερματισμού), θα είναι ονόματα αρχείων σε κάποιο μονοπάτι του συστήματος αρχείων του host. Τα δεδομένα του εικονικού δίσκου θα αντικατοπτρίζονται από τα δεδομένα των τεμαχίων, και τα μεταδεδομένα του εικονικού δίσκου από τα δεδομένα των κόμβων και των επικεφαλίδων συνταγής. Η επιλογή αυτή προσφέρει κομψότητα και απλότητα, καθώς εκμεταλλευόμαστε τους μηχανισμούς του συστήματος αρχείων για αυτόματη διαχείριση των οντο-

τήτων, εύκολη προσπέλασή τους και δέσμευση κατάλληλου φυσικού χώρου για αυτά.

Σε αντιδιαστολή με άλλες μορφές εικονικών δίσκων, όπως το qcow2, τα δεδομένα και μεταδεδομένα του εικονικού δίσκου δεν αποθηκεύονται σε ένα μεγάλο αρχείο αλλά σε πολλά μικρότερα. Τα παραπάνω αρχεία, ομαδοποιούνται ανάλογα με τον ονοματόχωρό τους ή ισοδύναμα ανάλογα με την οντότητά τους και αποθηκεύονται σε διαφορετικούς καταλόγους στο σύστημα αρχείων του host, σε μια συγκεκριμένη ιεραρχική δομή. Θεωρούμε ότι το σύστημά μας χρησιμοποιεί έναν αρχικό κατάλογο (έστω dtree) για αποθήκευση όλων των εικονικών δίσκων στον εν λόγω host, και έπειτα έχουμε όλα τα τεμάχια-αρχεία να αποθηκεύονται σε ένα υποκατάλογο chunk-store, όλοι οι κόμβοι-αρχεία σε ένα υποκατάλογο node-store και όλα τα αρχεία με τις επικεφαλίδες συνταγής σε ένα υποκατάλογο recipe-store. Το recipe-store επιπλέον υποδιαιρείται σε υποκαταλόγους, έναν για κάθε ξεχωριστό εικονικό δίσκο [1]. Μέσα σε κάθε τέτοιο υποκατάλογο εικονικού δίσκου που παίρνει το όνομά του από το όνομα που θα δώσει στον εικονικό δίσκο ο χρήστης, υπάρχει ένα αρχείο επικεφαλίδας συνταγής για τον πρωτεύοντα δίσκο και τα κάποια αρχεία επικεφαλίδας συνταγής για τα στιγμιότυπα. Τέλος, η ιεραρχική δομή καταλόγων συμπληρώνεται με την ύπαρξη ενός αρχείου info στον αρχικό κατάλογο dtree, το οποίο αποθηκεύει το τελευταίο δεσμευμένο uid για τον εικονικό δίσκο, ώστε κατά την δημιουργία ενός νέου εικονικού δίσκου να μπορεί να παρασχεθεί το επόμενο διαθέσιμο uid.

Οι οντότητες μας λοιπόν αποκτούν υπόσταση υπό την μορφή αρχείων, τα οποία οφείλουν να είναι διακριτά, μονοσήμαντα προσδιοριζόμενα και προσπελάσιμα με την βοήθεια κάποιας διεύθυνσης. Όλα τα παραπάνω προκύπτουν αβίαστα με την βοήθεια του συστήματος αρχείων, το οποίο αναθέτει σε κάθε αρχείο ένα μοναδικό μονοπάτι. Έτσι αρκεί να δημιουργήσουμε έναν μηχανισμό για να ονοματοδοτήσουμε μοναδικά κάθε οντότητα. Ο μηχανισμός αυτός, δημιουργεί τρεις ονοματόχωρους, έναν για κάθε θεμελιώδη οντότητα, και στη συνέχεια για να διαχωρίζουμε μοναδικά τις οντότητες σε έναν ονοματόχωρο χρησιμοποιούμε ένα δεδομένο μοτίβο ονοματοδοσίας. Το μοτίβο αυτό είναι ξεχωριστό για κάθε οντότητα, έχει νόημα μόνο μέσα στον ονοματόχωρό του και απαρτίζεται από αριθμούς όπως το uid του δίσκου, τον μετρητή στιγμιοτύπων, τον αύξοντα αριθμό ενός τεμαχίου, το επίπεδο ενός κόμβου και τον αύξοντα αριθμό ενός κόμβου στο επίπεδο. Για την υποστήριξη ενός συστήματος απαλοιφής διπλοτύπων, απαιτείται στην απλή περίπτωση και ένας νέος ονοματόχωρος αποτυ-

[1] και οι κλώνοι όπως προείπαμε θεωρούνται ξεχωριστοί εικονικοί δίσκοι

πωμάτων, όπου τα αρχεία θα ονοματοδοτούνται με βάση την τιμή κατακερματισμού τους. Για λόγους οικονομίας χρόνου και μειωμένης ανταποδοτικότητας στην πρωτοτυπία των πειραματικών αποτελεσμάτων, δεν υλοποιήσαμε ένα σύστημα απαλοιφής διπλοτύπων για τη δενδρική δομή μας. Φροντίσαμε όμως για την ενσωμάτωση της απαιτούμενης σημασιολογίας στις δομές και την λογική του κώδικα, ώστε μια μετέπειτα εισαγωγή του να μην απαιτήσει καμία τροποποίηση στην υπάρχουσα υλοποίηση. Για παράδειγμα, λήφθηκε μέριμνα ώστε τα ονόματα των κόμβων/τεμαχίων να υποστηρίζουν αποτυπώματα, ενώ προσοχή δόθηκε και σε ζητήματα ταυτοχρονισμού που θα εισήγαγε ένα σύστημα απαλοιφής διπλοτύπων.

Τα τεμάχια-αρχεία περιέχουν μόνο δεδομένα του εικονικού δίσκου (όχι μεταδεδομένα), σε αναλογία με τα δεδομένα που θα γράφονταν στους τομείς ενός φυσικού δίσκου. Οι κόμβοι-αρχεία και τα αρχεία επικεφαλίδας συνταγής, περιέχουν μεταδεδομένα, και συνεπώς για να είναι δυνατή η αποκωδικοποίησή τους οφείλουν να ακολουθούν συγκεκριμένα δομικά πρότυπα. Απαρτίζονται λοιπόν από κάποιες δομές που τοποθετούνται σε έναν προκαθορισμένο σκελετό, όπως θα δούμε στη συνέχεια.

Για να συνδεθούν και να λειτουργήσουν οι παραπάνω ιδέες, απαιτείται η ύπαρξη ενός κομματιού λογισμικού που θα εφαρμόσει τους κατάλληλους αλγορίθμους και θα υποστηρίξει έναν λειτουργικό εικονικό δίσκο με τα προαναφερθέντα χαρακτηριστικά. Για αυτό το σκοπό γράψαμε έναν οδηγό εικονικού δίσκου QEMU, ο οποίος ενσωματώνεται στο μοντέλο οδηγών του QEMU για εικονικές συσκευές.

Ο οδηγός εικονικού δίσκου γράφτηκε σε C, ενώ επιλέχθηκε το σύστημα αρχείων ext4 για τον host, ως ένα ευρέως διαδεδομένο, δοκιμασμένο και γρήγορο σύστημα αρχείων. Στο τέλος του κεφαλαίου, αφού παρουσιαστούν όλες οι λεπτομέρειες της υλοποίησης, παραθέτουμε τα βήματα που πραγματοποιούνται σε μια αίτηση Ε/Ε του guest.

Ονοματοδοσία και ονοματόχωροι αρχείων

Προαναφέραμε για τον διαχωρισμό των οντοτήτων σε 3 ονοματόχωρους. Ο διαχωρισμός αυτός ουσιαστικά πραγματοποιείται με τον διαχωρισμό των αρχείων στους 3 διαφορετικούς καταλόγους (chunk-store, node-store, recipe-store).

Οι επικεφαλίδες συνταγής αναφέρονται σε εικονικούς δίσκους και στιγμιότυπα, οπότε και αποφασίσαμε να ονοματοδοτούνται με βάση το όνομα του εικονικού δίσκου ή του στιγμιοτύπου τους. Κάθε εικονικός δίσκος οφείλει να έχει μοναδικό όνομα και αυτό

το όνομα δημιουργεί ένα δικό του υπο-ονοματόχωρο όπου υπάρχουν οι επικεφαλίδες του πρωτεύοντος εικονικού δίσκου και των στιγμιοτύπων του. Στα πλαίσια αυτού του υπο-ονοματόχωρου, ο πρωτεύων δίσκος έχει το όνομα του που δόθηκε στον εικονικό δίσκο ενώ κάθε στιγμιότυπο έχει το μοναδικό όνομα που δόθηκε κατά την δημιουργία του. Και αυτός ο υπο-ονοματόχωρος πραγματώνεται ως κατάλογος με τα αρχεία να λαμβάνουν τα ονόματα που προείπαμε.

Για να ονοματοδοτήσουμε τις υπόλοιπες 2 οντότητες χρησιμοποιούμε τα Name Identifiers (nid). Ένα nid μονοσήμαντα προσδιορίζει έναν κόμβο ή ένα τεμάχιο στα πλαίσια του ονοματόχωρού του και μπορεί να περιέχει το όνομα του αρχείου ενός κόμβου (node file name) ή το όνομα του αρχείου ενός τεμαχίου (chunk file name) ή την τιμή κατακερματισμού ενός κόμβου/τεμαχίου (fingerprint). Ένα entry αποτελείται από ένα nid και δικαιώματα (read-only 0 / read-write 1) για τον κόμβο/τεμάχιο στον οποίο δείχνει. Το NID ορίζεται από το nid.value που είναι ένας αριθμός που περιέχει το όνομα, δηλαδή το μοναδικό αναγνωριστικό του κόμβου/τεμαχίου, και από το nid.type που ορίζει αν το όνομα είναι όνομα αρχείου κόμβου, όνομα αρχείου τεμαχίου ή αποτύπωμα. Ανάλογα με το nid.type υπάρχει και αυτόματη σύνδεση του ονόματος αρχείου με τον κατάλληλο ονοματόχωρο, δηλαδή το εκάστοτε αρχείο αναζητείται στον κατάλληλο -με βάση τον τύπο του- κατάλογο.

Τα ονόματα των αρχείων συντίθενται από αριθμούς που αναφέραμε στο κεφάλαιο του σχεδιασμού, έτσι ώστε κάθε οντότητα να είναι διακριτή και μονοσήμαντα προσδιορίσιμη. Με βάση αυτούς τους αριθμούς προκύπτει η δομή των ονομάτων. Με βάση το ανώτατο όριο για κάθε τέτοιο αριθμό προκύπτει το μέγεθος του και άρα και επαγωγικά προσδιορίζεται και το μέγεθος του κάθε τύπου ονόματος.
Το nid.value για τα τεμάχια είναι: uid|cow_cntr|chunk_num (16 bytes)
Το nid.value για τους κόμβους είναι: uid|cow_cntr|level|num_in_level (17 bytes)
Το nid.value για τα hashes είναι: hash (ανάλογα με τον αλγόριθμο κατακερματισμού που χρησιμοποιείται στην απαλοιφή διπλοτύπων)
Τέλος υπάρχει και το NID_TYPE_NULL το οποίο υπονοεί ότι το entry δεν δείχνει πουθενά, γιατί δεν έχουν γραφτεί δεδομένα στο υποδέντρο που έχει αναλάβει να δεικτοδοτεί. Τα δεδομένα αυτά δεν έχουν γραφτεί, γιατί η περιοχή του δίσκου που ορίζει το υποδέντρο είναι άδεια.

Το NID.value αποτελείται λοιπόν από bytes που πρέπει να γίνουν ascii χαρακτήρες,

για να μπορούμε να αναφερθούμε σε όνομα του αρχείου, στα πλαίσια των ονομάτων αρχείων του συστήματος αρχείων. Γι' αυτό χρησιμοποιούμε μια δεκαεξαδική μετατροπή του όπου κάθε 4bits αναπαρίστανται και από έναν δεκαεξαδικό ψηφίο. Σε όλα τα παραπάνω πεδία εκτός του uid που είναι πρώτο, προσθέτουμε στην αρχή όσα μηδενικά χρειάζεται για την πλήρη αναπαράσταση όλων των byte (π.χ. cow_cntr είναι 32bit οπότε το 1010 δεν μετατρέπεται σε a αλλά σε 0000000a).

Μορφή αρχείων μεταδεδομένων: δομές και σκελετός

Εδώ θα παρουσιάσουμε την μορφή των βασικών δομών που χρησιμοποιούνται στα μεταδεδομένα μας και θα δομήσουμε τον σκελετό που τα αρχεία των μεταδεδομένων οφείλουν να έχουν.

Οι επικεφαλίδες συνταγών περιγράφουν έναν πρωτεύοντα εικονικό δίσκο ή έναν κλώνο ή ένα στιγμιότυπο. Περιέχουν πληροφορίες που περιγράφουν τον εικονικό δίσκο και είναι χρήσιμες για τον οδηγό μας και άλλα επίπεδα της στοίβας διαχείρισης εικονικών συσκευών μπλοκ του QEMU. Ο σκελετός τους έχει την εξής δομή:

0-63 bytes:	RecipeHeader	
64-127 bytes:	SnapshotHeader	μόνο σε στιγμιότυπα
128-255 bytes:	virtual disk name	όσα bytes χρειαστούν, το μέγιστο 128
256-511 bytes:	root nid	όσα bytes έχει το hash, το μέγιστο 256

Για να αποφύουμε την σύγχυση, το RecipeHeader είναι μια δομή που υπάρχει μέσα σε ένα αρχείο επικεφαλίδας συνταγής (recipe header file). Η δομή αυτή παρέχει βασικές πληροφορίες που είναι απαραίτητο να γραφτούν σε ένα μη-πτητικό μέσο αποθήκευσης.

```
struct RecipeHeader {
 uint32_t magic;       αριθμός χρήσιμος κατά το probe της μορφής
                       του εικονικού δίσκου
 uint32_t uid;         μοναδικό αναγνωριστικό εικονικού δίσκου
                       (πρωτεύοντος ή κλώνου)
 uint64_t size;        μέγεθος δίσκου σε bytes
 uint32_t chunksize;   μέγεθος τεμαχίου σε bytes
 uint8_t levels;       αριθμός επιπέδων δενδρικής συνταγής
 uint64_t node_size;   μέγεθος κόμβου
```

`uint16_t is_snapshot;`	περιγράφει αν η δεδομένη επικεφαλίδα συνταγής αναφέρεται σε στιγμιότυπο
`uint32_t next_snapshot_num;`	ο αύξων αριθμός του επόμενου στιγμιοτύπου που θα ληφθεί
`uint16_t disk_name_size;`	το μέγεθος του ονόματος του εικονικού δίσκου

```
}
```

Η SnapshotHeader επικεφαλίδα υπάρχει μόνο στις επικεφαλίδες συνταγής που αναφέρονται σε στιγμιότυπα και είναι καθαρά για σύμπλευση με τις απαιτήσεις του στρώματος μπλοκ του QEMU. Οι παρακάτω πληροφορίες στιγμιοτύπων, οφείλουν να αποθηκεύονται για κάθε μορφή εικονικού δίσκου που υποστηρίζει στιγμιότυπα.

```
struct SnapshotHeader {
 uint32_t date_sec;
 uint32_t date_nsec;
 uint64_t vm_clock_nsec;
 uint64_t vm_state_size;
}
```

Το virtual disk name απλώς αποθηκεύει το όνομα του εικονικού δίσκου σε ascii κωδικοποίηση. Γράφεται στο αρχείο επικεφαλίδας και για πρακτικούς λόγους (αποδοτική υλοποίηση απαρίθμησης στιγμιοτύπων). Το root nid είναι το nid του κόμβου-ρίζας ενός εικονικού δίσκου (ή ενός στιγμιοτύπου ή ενός κλώνου) και είναι απαραίτητο να γραφτεί στο αρχείο επικεφαλίδας ώστε να υπάρχει σύνδεση του εικονικού δίσκου (ή ενός στιγμιοτύπου ή ενός κλώνου) με την δενδρική συνταγή του.

Οι κόμβοι έχουν την εξής μορφή:

0-15 bytes:	NodeHeader
16-... bytes:	πίνακας με πολλά στιγμιότυπα της δομής RecipeEntry. Κάθε entry «δείχνει» σε ένα κόμβο ή τεμάχιο

```
struct NodeHeader {
 uint8_t level;
 uint64_t node_size;
}
```

Όπως προαναφέραμε στο κεφάλαιο του σχεδιασμού, το NodeHeader είναι απαραί-

τητο ώστε οι κόμβοι να έχουν ανεξάρτητη υπόσταση και να μπορούν να χρησιμοποιηθούν έξω από τα πλαίσια ενός συγκεκριμένου εικονικού δίσκου.

`struct RecipeEntry {`

`NID nid;`	όνομα δεικτοδοτούμενου κόμβου/τεμαχίου με βάση την σύμβαση ονοματοδοσίας
`uint8_t permission;`	δικαιώματα εγγραφής δεικτοδοτούμενου κόμβου/τεμαχίου (0: read-only, 1: read-write)

`}`

Η ονοματοδοσία και τα μεγέθη που επιλέχθηκαν για τα διάφορα πεδία των δομών δίνουν τα εξής άνω όρια για το σύστημά μας:
μέγιστο μέγεθος εικονικού δίσκου: 16 Exabyte
μέγιστο ύψος δέντρου: 200 επίπεδα (πρακτικά περιοριζόμαστε στα 10)
μέγιστο μέγεθος τεμαχίου: 128MB (πρακτικά περιοριζόμαστε σε <1MB)
μέγιστος αριθμός στιγμιοτύπων: $2^{32} - 1 \approx 4.294$ δισεκατομμύρια
Επιπλέον πρακτικούς περιορισμούς θέτουν το μέγεθος του συστήματος αρχείων του host και ο αριθμός των inodes και άρα των αρχείων που υποστηρίζει.

Βελτιστοποιήσεις και λεπτομέρειες υλοποίησης

Όλες οι δομές της C πιάνουν μόνο όσα bytes απαιτούνται και δεν προστίθεται padding για βελτιστοποίηση. Αυτό γίνεται μέσω του QEMU_PACKED που ουσιαστικά εφαρμόζει την επιλογή __attribute__((__packed__)) σε ένα struct. Ενώ σε κάποιες περιπτώσεις αυτό δεν είναι σημαντικό, στην περίπτωση του RecipeEntry είναι εξαιρετικά σοβαρό, οπότε ακόμα και ένα byte, πολλαπλασιαζόμενο με όλα τα entries των κόμβων θα επηρεάσει σημαντικά την ποσότητα των μεταδεδομένων. Γι' αυτό άλλωστε και το nid.type καταλαμβάνει 1 byte και δεν το ορίσαμε με την βοήθεια του enum, όπου και θα καταλάμβανε 4.

Για βελτιστοποίηση της απόδοσης και για να γλιτώσουμε άσκοπες λειτουργίες Ε/Ε, για ένα τεμάχιο που στην παρούσα αίτηση πρέπει να γίνει COW αλλά μετά πρόκειται να γραφτεί ολόκληρο, δεν απαιτείται αντιγραφή του παλαιοτέρου και μετά εγγραφή των νέων δεδομένων, αλλά μόνο εγγραφή στο νεοσύστατο τεμάχιο. Εάν δεν γράφεται ολόκληρο, τότε επιλέγουμε να το αντιγράψουμε όλο και όχι να αντιγράψουμε τα δυο κομμάτια που περισσεύουν εκατέρωθεν του αιτούμενου, αφού για τόσο μικρά μεγέθη

όσο αυτά των τεμαχίων, η επιβάρυνση 2 λειτουργιών Ε/Ε είναι μάλλον μεγαλύτερη από αυτήν της 1 σειριακής μεγαλύτερου μεγέθους. Θα μπορούσαμε να επιτύχουμε περαιτέρω βελτιστοποίηση συγχωνεύοντας τις δυο λειτουργίες εγγραφής σε μια.

Στην περίπτωση που δημιουργήθηκαν κόμβοι/τεμάχια, οι ενημερώσεις των entries του πατέρα με καινούργια nid γίνονται όλες μαζί σε μία αίτηση εγγραφής, ώστε να αυξηθεί η απόδοση σε σχέση με πολλαπλές λειτουργίες Ε/Ε για κάθε ξεχωριστό entry που ενημερώνεται.

Επίσης, όπως προείπαμε, μόλις βρεθεί ένα μηδενικό entry (NID_TYPE_NULL), επιστρέφονται κατευθείαν μηδενικά για το αντίστοιχο εύρος τομέων, χωρίς περαιτέρω διάσχιση του υποδέντρου.

Για ταχύτητα στις αναγνώσεις/εγγραφές, παρακάμπτουμε το επίπεδο μπλοκ του QEMU και εφαρμόζουμε native system calls, δηλαδή `pread()/pwrite()` στους file descriptors των αρχείων, αντί για `bdrv_read(), brdv_write()` σε αρχεία που θα αντιμετωπίζονταν ως raw και θα εξυπηρετούνταν από τον raw εικονικό οδηγό του QEMU. Εφαρμόζουμε βελτιστοποίηση ώστε το αρχείο του κόμβου-ρίζας να είναι πάντα ανοικτό στον πίνακα των file descriptors, ώστε να μην δαπανάμε χρόνο για κλήσεις συστήματος `open()`. Το ίδιο συμβαίνει και για το αρχείο της επικεφαλίδας συνταγής. Επίσης η δημιουργία και εγγραφή ενός τεμαχίου απαιτεί μόνο μια κλήση συστήματος `open()`.

Επιλέξαμε να μην εκμεταλλευτούμε την δυνατότητα των αραιών αρχείων για τα αρχεία κόμβων και τεμαχίων, αφού μπλοκ δεδομένων του host που δεσμεύονται σε διαφορετικές χρονικές στιγμές, κατά κανόνα δεν είναι συνεχόμενα στο δίσκο. Έτσι όμως εισάγεται κατακερματισμός στα δεδομένα και τα μεταδεδομένα, και διαταράσσεται η τοπικότητα, η οποία αποτελεί βασικό χαρακτηριστικό για την αποτελεσματική εκτέλεση λειτουργιών Ε/Ε. Αποφασίσαμε λοιπόν κατά την δημιουργία ενός αρχείου κόμβου/τεμαχίου, να χρησιμοποιούμε την `posix_fallocate()` έτσι ώστε να δεσμεύσουμε όσο χώρο αυτό το αρχείο θα χρειαστεί στο σύστημα αρχείων του host. Στην πραγματικότητα απλώς δεσμεύονται τα data blocks/extents και ενημερώνονται οι δείκτες του inode ώστε να δείχνουν σε αυτά, χωρίς όμως να γράφονται μηδενικά σε αυτά τα block/extents και γι᾽ αυτό το λόγο είναι και πιο γρήγορη ως διαδικασία. Χρησιμοποιούμε πρωθύστερη δέσμευση χώρου (preallocation) ώστε να διασφαλίσουμε ότι ο χώρος αυτός θα είναι φυσικά συνεχόμενος και το αρχείο μας δεν θα υποστεί κατακερματισμό. Αυτό είναι χρήσιμο, αφού θέλουμε για λόγους απόδοσης, η ανάγνωση και η

εγγραφή των θεμελιωδών οντοτήτων μας να μπορούν να πραγματοποιούνται σε μια σειριακή λειτουργία Ε/Ε.

Όταν πρέπει να αντιγράψουμε μεγάλους κόμβους, καλούμε την `posix_fadvise()` με παράμετρο `POSIX_FADV_SEQUENTIAL`, ώστε να ενημερώσουμε τον πυρήνα για την σειριακή προσπέλαση του αρχείου, και εν συνεχεία αυτός να διενεργήσει readahead και άλλες βελτιστοποιήσεις για να αυξήσει την απόδοση.

Χρησιμοποιούμε τόσο την επιλογή `O_NOATIME` στις κλήσεις συστήματος `open()` των κόμβων/τεμαχίων όσο και την επιλογή `noatime` και `nodirtime` στο ext4. Αυτές οι επιλογές απενεργοποιούν την εγγραφή της χρονοσφραγίδας τελευταίας προσπέλασης στα μεταδεδομένα ενός αρχείου κάθε φορά που αυτό ανοίγει. Η πολιτική αυτή ακολουθείται γιατί δεν μας ενδιαφέρει ο τελευταίος χρόνος προσπέλασης και το μόνο που κάνει είναι να εισάγει την επιβάρυνση της εγγραφής μεταδεδομένων.

Με βάση τα ανωτέρω, καλό είναι ο αρχικός κατάλογος που θα ενσωματώσει τα δεδομένα και τα μεταδεδομένα όλων των εικονικών δίσκων του συστήματος, να τοποθετηθεί σε ένα ξεχωριστό δίσκο ή διαμέριση, ώστε να μην επηρεάζεται και να μην επηρεάζει άλλες εφαρμογές και δεδομένα που μπορεί να τρέχουν στον host. Διατηρώντας ξεχωριστή διαμέριση μειώνεται και η πιθανότητα κατακερματισμού των αρχείων σε απομακρυσμένους τομείς του φυσικού δίσκου.

4.2 Υλοποίηση οδηγού εικονικού δίσκου και ενσωμάτωση στον QEMU

Η υλοποίηση των παραπάνω έγινε για τον QEMU emulator, ώστε να υποστηρίξει ένα εικονικό δίσκο που μπορεί να χρησιμοποιηθεί κανονικά από ένα guest VM. Για να υλοποιηθεί η λειτουργικότητα που περιγράψαμε, γράψαμε έναν οδηγό εικονικού δίσκου για τον QEMU (QEMU virtual block driver) που ενσωματώνει την λογική και τους αλγορίθμους του σχεδιασμού μας. Πέρα από την προσαρμογή του οδηγού αυτού στον κώδικα του QEMU δεν χρειάζεται καμία άλλη τροποποίηση. Ο οδηγός αυτός ενσωματώνεται στο μοντέλο των οδηγών εικονικών συσκευών μπλοκ του QEMU και επιτελεί τα εξής:

- εισάγει τις κατάλληλες δομές και υλοποιεί τις απαιτούμενες συναρτήσεις με βάση

το μοντέλο των οδηγών εικονικών συσκευών μπλοκ του QEMU

- δημιουργεί τα κατάλληλα αρχεία, με τα κατάλληλα ονόματα, στα κατάλληλα σημεία του δέντρου καταλόγων

- δημιουργεί τα αρχεία μεταδεδομένων, με την χρήση συγκεκριμένων προτύπων και δομών για κάθε τύπο αρχείου

- επιτελεί την διαδικασία μετάφρασης από έναν αριθμό τομέα του εικονικού δίσκου σε ένα αρχείο τεμαχίου, κάνοντας COW όπου απαιτείται

- υλοποιεί δημιουργία εικονικών δίσκων, δημιουργία, επαναφορά και διαγραφή στιγμιοτύπων και δημιουργία κλώνων

Μοντέλο εικονικών συσκευών και οδηγών εικονικών συσκευών QEMU

Με βάση τον πηγαίο κώδικα του QEMU[gd16b], οι δυο βασικές έννοιες στο Qdev (Qemu Device Model) είναι οι συσκευές (devices) και οι δίαυλοι (buses). Μια συγκεκριμένη εικονική συσκευή αναπαριστάται από ένα DeveiceState και ένας εικονικός δίαυλος από ένα BusState. Μια συσκευή έχει έναν γονεϊκό δίαυλο και μπορεί να προσφέρει έναν ή περισσότερους διαύλους ως παιδιά της. Αντίστοιχα ένας δίαυλος έχει μια γονεϊκή συσκευή και μπορεί να προσφέρει μια ή περισσότερες συσκευές ως παιδιά του. Οι συσκευές αποκτούν συγκεκριμένες διευθύνσεις στον δίαυλο. Έτσι δημιουργείται ένα αυστηρό δέντρο με εναλλασσόμενα επίπεδα συσκευών και διαύλων. Η ρίζα του δέντρου είναι το κύριο system bus, ή αλλιώς SysBus.

Μια εικονική συσκευή πέρα από την generic δομή `DeviceState`, αποκτά και μια πιο συγκεκριμένη δομή δεδομένων ανάλογα με την διεπαφή της (interface). Για παράδειγμα για SCSI την `SCSIDevice`, για ISA την `ISADevice`, για virito την `VirtIODevice` κ.ο.κ. Οι δομές αυτές αναπαριστούν τα στιγμιότυπα των συσκευών έτσι όπως έχουν εισαχθεί στο δέντρο συσκευών της εικονικής μηχανής και περιέχουν πληροφορίες για την κατάσταση της συσκευής. Για την δημιουργία τόσο των `DeviceState` όσο και των συγκεκριμένων επεκτάσεών τους, υπάρχουν αντίστοιχες κλάσεις `DeviceClass` και `SCSIDeviceClass`, `ISADeviceClass`, `VirtIODeviceClass`, κ.ο.κ. Οι κλάσεις αυτές, πέρα από τις μεθόδους δημιουργίας των στιγμιοτύπων των συγκεκριμένων συσκευών, ενσωματώνουν πιθανώς σε μεθόδους και επιπλέον λογική που απαιτείται ώστε να λαμβάνουν τα αιτήματα του guest.

Το παραπάνω μοντέλο των εικονικών συσκευών, καταρτίζει την αρχιτεκτονική διαρρύθμιση των συσκευών της εικονικής μηχανής, αντίστοιχη με την πραγματική αρχιτεκτονική διαρρύθμιση συσκευών και διαύλων σε ένα φυσικό μηχάνημα. Αυτή η διαρρύθμιση προβάλλεται στο guest λειτουργικό σύστημα, το οποίο την ερμηνεύει ως πραγματική και ταυτίζει κάθε `DeviceState` με μια συσκευή στην οποία μπορεί να έχει πρόσβαση το εικονικό μηχάνημα. Εμείς επικεντρωνόμαστε στις εικονικές συσκευές μπλοκ, δηλαδή ουσιαστικά στους εικονικούς δίσκους.

Για να είναι χρηστικές οι παραπάνω εικονικές συσκευές μπλοκ και να μπορούν να εξυπηρετήσουν αιτήσεις του guest πρέπει να συνδεθούν με τον κώδικα που εξομοιώνει την λειτουργία τους. Όπως παρουσιάσαμε στην ενότητα 2.2.3, η εξομοίωση της λειτουργίας των εικονικών δίσκων, βασίζεται στην ιδέα της αντιστοίχισης των αριθμών εικονικών τομέων σε κατάλληλες θέσεις μιας άλλης οντότητας αποθήκευσης, η οποία μπορεί να είναι οτιδήποτε (αρχείο του host, συσκευή μπλοκ, αντικείμενο κ.ο.κ.). Από την στιγμή όμως που υπάρχουν διάφορες τέτοιες αντιστοιχίσεις και διάφοροι τρόποι αυτές να υλοποιηθούν, αναδύονται πολλές διαφορετικές μορφές εικονικών δίσκων (formats), και συνάμα η ανάγκη για υλοποιήσεις όλων αυτών των διαφορετικών σχεδιασμών. Ο κώδικας που πραγματώνει την λογική κάθε ξεχωριστής μορφής εικονικού δίσκου, συνιστά έναν οδηγό εικονικών συσκευών μπλοκ του QEMU ή αλλιώς έναν οδηγό εικονικού δίσκου QEMU ή ισοδύναμα έναν specific qemu block driver.

Από τα παραπάνω προκύπτει ότι υπάρχουν δυο διαφορετικές όψεις μιας εικονικής συσκευής. Η guest-point-of-view, που περιγράφει πώς η εικονική συσκευή συνδέεται στο guest εικονικό μηχάνημα, προσδιορίζοντας την διεπαφή και την εικονική θέση της στην εικονική ιεραρχία διαύλων και συσκευών, και η host-point-of-view, που ασχολείται με την μορφή (format) του εικονικού δίσκου, τις εμπλεκόμενες οντότητες του host και με την σύνδεση του κατάλληλου οδηγού ώστε να εξυπηρετούνται τα αιτήματα σε τομείς και να καθίστανται διαθέσιμα τα δεδομένα στον guest.

Η σύνδεση ανάμεσα σε αυτές τις δύο όψεις, που ουσιαστικά ανάγεται στην σύνδεση ενός `DeviceState` με έναν specific qemu block driver, πραγματοποιείται με την μεσολάβηση κάποιων generic στρωμάτων. Αυτά ενοποιούν διαχειριστικές λειτουργίες κοινές σε όλες τις εικονικές συσκευές, πριν καταλήξουμε στην διακλάδωση σε συγκεκριμένες υλοποιήσεις ανάλογα με τον specific οδηγό που αντιστοιχεί σε κάθε εικονικό δίσκο.

Υπάρχουν τα εξής στρώματα λογισμικού που παρεμβάλλονται στην εξυπηρέτηση μιας λειτουργίας ενός εικονικού δίσκου.

- block backend layer
- generic block drivers layer
- specific block driver layer

Το block backend layer, έχει στόχο ακριβώς αυτήν την σύνδεση ανάμεσα στη guest-point-of-view και στην host-point-of-view. Κάθε στιγμιότυπο ενός `DeviceState` συνδέεται με ένα στιγμιότυπο μιας generic δομής, που ονομάζεται `BlockBackend`. Στην συνέχεια αυτό το `BlockBackend` καλείται να συνδεθεί με τον κατάλληλο οδηγό εικονικής συσκευής, ανάλογα με την μορφή του εικονικού δίσκου. Το generic block drivers layer, έχει στόχο να ομαδοποιήσει και να καταστρώσει μια λογική αφαίρεση των κοινών λειτουργιών των εικονικών δίσκων, έτσι ώστε να υπάρχει ένα πρότυπο συναρτήσεων που οι specific qemu block drivers θα υλοποιούν κατά βούληση.

Για κάθε λειτουργία του εικονικού δίσκου λοιπόν λοιπόν, υπάρχει μια αλληλουχία κλήσεων από το ανώτερο στο κατώτερο στρώμα. Κάθε στρώμα επικοινωνεί με τα διπλανά του με βάση ένα σαφώς ορισμένο API. Το generic block drivers layer δέχεται αιτήσεις από το block backend layer, τις προεπεξεργάζεται κατάλληλα και στη συνέχεια καλεί την συγκεκριμένη υλοποίηση του οδηγού για την εν λόγω λειτουργία. Εάν μια τέτοια υλοποίηση δεν υποστηρίζεται από τον εκάστοτε οδηγό ή εάν δεν χρειάζεται να υποστηρίζεται, προσπαθεί με generic τρόπο να την εξυπηρετήσει αυτό.

Κατά την έναρξη ενός VM, αναλύονται οι επιλογές που έχει δώσει ο χρήστης, μέσα στις οποίες ορίζονται και οι εικονικοί δίσκοι που έχουν δοθεί για προσάρτηση. Συνήθως, κάθε εικονικός δίσκος απαιτεί τουλάχιστον μια επικεφαλίδα (virtual disk header file), η οποία περιέχει χρήσιμες πληροφορίες για τον εικονικό οδηγό και θεωρείται αντιπροσωπεύει τον εικονικό δίσκο στον host, πέρα από τις οντότητες που μπορεί να εμπλέκονται για την αποθήκευση των δεδομένων ή των μεταδεδομένων του εικονικού δίσκου. Για να γίνει ο δίσκος ορατός στον guest, δημιουργείται το αντικείμενο `DeviceState` και οι επεκτάσεις του, τοποθετούνται στην ιεραρχία και στη συνέχεια ο QEMU είναι δυνατόν να προσφέρει στον guest την ψευδαίσθηση μιας διεπαφής υλικού. Ανάλογα με τις πληροφορίες της επικεφαλίδας ή τις επιλογές που έδωσε ο χρήστης, εντοπίζεται ο κατάλληλος οδηγός μπλοκ για τον εικονικό δίσκο και δημιουργούνται τα αντικείμενα `BlockDriverState` και `BDRV<disk_format>State`

με περιέχουν χρήσιμες πληροφορίες για τον οδηγό αυτό. Η σύνδεση ανάμεσα στο guest-point-of-view και το host-point-of-view πραγματοποιείται με την δημιουργία ενός στιγμιοτύπου της δομής `BlockBackend`, στην οποία συνδέεται κάποια επέκτασή του `DeviceState` και η `BlockDriverState` δομή του οδηγού. Χάρη σε αυτήν την σύνδεση είναι δυνατόν στη συνέχεια να δρομολογούνται οι αιτήσεις από τον guest στον specific block driver, μέσα από αλυσίδες συναρτήσεων που διατρέχουν τα επίπεδα και τις δομές που περιγράψαμε.

Εδώ παραθέτουμε και τα βασικά είδη δομών (αντικειμένων) που χρησιμοποιούνται από τα ανωτέρω στρώματα:

- `BlockBackend`: στιγμιότυπο που συνδέει τα στιγμιότυπα των συσκευών, όπως προβάλλονται στον guest, με τους οδηγούς τους. Η δομή αυτή έχει κυρίως δείκτες προς άλλες δομές.

- `BlockDriver`: ο αφηρημένος οδηγός που χειρίζεται εικονικούς δίσκους μιας δεδομένης μορφής. raw, qcow2, qed, nfs, κτλ. Η δομή αυτή μοιάζει με μια κλάση, καθώς περιέχει συναρτήσεις που υλοποιούν την λογική διαχείρισης ενός format εικονικών δίσκων.

- `BlockDriverState`: ένα στιγμιότυπο ενός από τους `BlockDriver`, στο οποίο είναι συνδεδεμένο ένα συγκεκριμένο ένα συγκεκριμένο virtual disk header file. Η δομή αυτή περιέχει χρήσιμες πληροφορίες για τον εικονικό δίσκο.

- μια ιδιωτική και driver-specific δομή που περιέχει χρήσιμες πληροφορίες που ο οδηγός μας χρειάζεται όσο ο εικονικός δίσκος είναι συνδεδεμένος. Η σύμβαση που ακολουθείται είναι αυτή η δομή να ονομάζεται `BDRV<disk_format>State` (π.χ. `BDRVQcow2State`). Η `BlockDriverState` δείχνει σε αυτήν την ιδιωτική δομή.

Για παράδειγμα μπορούμε να δούμε την αλληλουχία συναρτήσεων κατά την δημιουργία ενός εικονικού δίσκου, κατά την εισαγωγή του σε ένα VM και κατά τη διάρκεια μιας αίτησης read του guest. Με πράσινο οι συναρτήσεις του block backends layer, με μπλε του generic block drivers layer, με κόκκινο του specific block drivers layer και με μαύρο συναρτήσεις άλλων στρωμάτων. Παρατηρούμε ότι μπορούμε να εξάγουμε το στρώμα και από τα προθέματα των συναρτήσεων (blk για το block backend και bdrv για το generic/specific block drivers).

Δημιουργία:

`img_create` → `bdrv_image_create` → `qemu_coroutine_create` → `qemu_coroutine_enter` → `bdrv_create_co_entry` → `drv->bdrv_create`

Εισαγωγή:

`drive_new` → `blockdev_init` → `blk_new_open` → `a. blk_new_with_bs`, `b. bdrv_open` → `bdrv_open_inherit` → `drv->bdrv_open`

Αίτηση read:

`blk_aio_readv` → `bdrv_aio_readv` → `bdrv_co_aio_rw_vectors` → `qemu_coroutine_create` → `qemu_coroutine_enter` → `bdrv_co_do_rw` → `bdrv_co_do_readv` → `bdrv_co_do_preadv` → `bdrv_aligned_preadv` → `drv->bdrv_co_readv`

Συναρτήσεις που υλοποιήθηκαν

Με βάση τα παραπάνω θα πρέπει να υλοποιήσουμε έναν `BlockDriver` ο οποίος θα υιοθετεί το API που ορίζει το generic block drivers layer και θα υλοποιεί με τον τρόπο που θέλουμε το σύνολο λειτουργιών που ορίζει το API. Σε πρώτη φάση επιλέξαμε να υλοποιεί ένα βασικό υποσύνολο του που αναλύουμε παρακάτω:

bdrv_probe(): σε περίπτωση που ο χρήστης δεν έχει διατυπώσει ρητά την μορφή που έχει το αρχείο εικονικού δίσκου (raw, qcow2 κλπ) και άρα δεν μπορεί να προσδιοριστεί ο κατάλληλος οδηγός για αυτό, καλείται η εν λόγω συνάρτηση κάθε οδηγού. Η συνάρτηση αυτή, επιθεωρώντας συνήθως κάποια επικεφαλίδα και κάποιο magic number του αρχείου, επιστρέφει ένα σκορ συμβατότητας, που περιγράφει το πόσο πιθανό είναι ο συγκεκριμένος οδηγός να είναι υπεύθυνος να διαχειριστεί το αντίστοιχο αρχείο.
Στον οδηγό μας εξετάζουμε το magic number που υπάρχει στα πρώτα 4 bytes της επικεφαλίδας συνταγής και αν είναι πανομοιότυπο επιστρέφουμε το μέγιστο σκορ συμβατότητας, ειδάλλως 0.

bdrv_open(): κάθε φορά που εντοπίζεται ένα μια εικονική συσκευή μπλοκ, στο τέλος των κλήσεων για την ενσωμάτωσή της συσκευής στο μοντέλο που περιγράψαμε παραπάνω, καλείται και αυτή η συνάρτηση. Κυρίως αναλαμβάνει να δημιουργήσει και να αρχικοποιήσει την ιδιωτική driver-specific δομή που θα περιέχει απαραίτητες πληροφορίες, να επεξεργαστεί τυχόν επιλογές σχετικές με τον οδηγό που έχουν δοθεί από

τον χρήστη, αλλά και να επιτελέσει τυχόν άλλες χρήσιμες λειτουργίες που πρέπει να προηγηθούν πριν από την χρήση του οδηγού και είναι απαραίτητες για την ομαλή προσομοίωση του εικονικού δίσκου.
Στον οδηγό μας δημιουργούμε και αρχικοποιούμε την ιδιωτική δομή `BDRVDtreeState`, συμβουλευόμενοι πεδία που υπάρχουν στην επικεφαλίδα συνταγής. Θέτουμε επίσης στην `BlockDriverState` δομή το μέγεθος του δίσκου.

bdrv_close(): όταν μια εικονική συσκευή μπλοκ γίνεται unplugged, όπως για παράδειγμα κατά τον τερματισμό του VM, καλείται αυτή η συνάρτηση για να αποδεσμεύσει μνήμη που πιθανώς να χρησιμοποιεί αλλά και για να επιτελέσει άλλες λειτουργίες απαραίτητες για την ομαλή αποσύνδεση του εικονικού δίσκου.
Στον οδηγό μας απλώς αποδεσμεύουμε την μνήμη κάποιων δυναμικών μεταβλητών.

bdrv_create(): όταν ο χρήστης αιτείται την δημιουργία ενός εικονικού δίσκου, τότε καλείται αυτή ή συνάρτηση ώστε να δημιουργήσει το αρχείο στον host και να αρχικοποιήσει τις δομές του με πιθανώς κατάλληλες πληροφορίες. Εάν υπάρχουν διάφορες επιλογές για την παραμετροποίηση του εικονικού δίσκου, αυτή η συνάρτηση δέχεται την λίστα των επιλογών και επιτελεί τις απαιτούμενες ρυθμίσεις. Συνήθως, σε αυτήν την φάση γράφεται και η επικεφαλίδα του εικονικού δίσκου και γίνεται, εάν έχει δοθεί ως επιλογή, και πρωθύστερη δέσμευση χώρου (preallocation) για τα δεδομένα ή μεταδεδομένα του δίσκου.
Στον οδηγό μας δημιουργούμε όσους καταλόγους και αρχεία της ιεραρχικής δομής που περιγράφτηκαν στην 4.1, δεν υπάρχουν ήδη. Εν συνεχεία επεξεργαζόμαστε τις επιλογές που μπορεί να έχει δώσει ο χρήστης και, αφού καταλήξουμε στις τιμές των παραμέτρων μας, τις γράφουμε μαζί με άλλα χρήσιμα δεδομένα στο αρχείο επικεφαλίδας συνταγής.

bdrv_co_read(): κάθε αίτηση ανάγνωσης του guest, καταλήγει μετά από διάσχιση πολλών ανώτερων επιπέδων, σε αυτήν την συνάρτηση του οδηγού μας. Αυτή οφείλει με βάση το εύρος τομέων που δόθηκε, να συμβουλευτεί την δομή αντιστοίχισης, να επενεργήσει στο κατάλληλο αρχείο του host μέσω κλήσεων συστήματος, και να γεμίσει έναν buffer με τα κατάλληλα δεδομένα. Ο buffer αυτός αντανακλά μνήμη του guest λειτουργικού συστήματος. Περιέχει το υποθέμα «co», γιατί η συνάρτηση αυτή τρέχει σε coroutine contex. Ένα coroutine είναι ένα cooperative userspace thread και αποτελεί ένα ειδικό τύπο συνάρτησης. Μόνο ένα coroutine μπορεί να τρέχει κάθε στιγμή,

δεν μπορεί να διακοπεί από άλλα coroutines, αλλά μπορεί να τρέχει παράλληλα με άλλες κανονικές συναρτήσεις. Ένα coroutine κατά βούληση μόνο, ή όταν τερματίσει, παραχωρεί τον έλεγχο σε μια άλλη συνάρτηση, που δεν είναι απαραίτητα, όπως στον κλασσικό προγραμματισμό, η συνάρτηση που κάλεσε το coroutine. [Knu97]

bdrv_co_write(): κάθε αίτηση εγγραφής του guest καταλήγει μετά από διάσχιση πολλών ανώτερων επιπέδων σε αυτήν την συνάρτηση του οδηγού μας. Αυτή οφείλει με βάση το εύρος τομέων που δόθηκε, να συμβουλευτεί την δομή αντιστοίχισης, να επενεργήσει στο κατάλληλο αρχείο του host μέσω κλήσεων συστήματος, και να γεμίσει τις κατάλληλες θέσεις στο αρχείο με τα δεδομένα ενός buffer. Ο buffer αυτός αντανακλά μνήμη του guest λειτουργικού συστήματος. Παρομοίως με την `bdrv_co_read()`, λειτουργεί σε coroutine context.

bdrv_co_flush_to_disk(): η συνάρτηση αυτή καλείται κάθε φορά που θέλουμε να διασφαλίσουμε ότι όλα τα δεδομένα του εικονικού δίσκου βρίσκονται αποθηκευμένα στον φυσικό δίσκο και δεν διατηρούνται απλώς στην πτητική μνήμη του host. Αυτό συμβαίνει γιατί κατά κανόνα οι κλήσεις συστήματος `write()` σε ένα αρχείο στον host δεν γράφουν τα δεδομένα κατευθείαν στον φυσικό δίσκο αλλά χρησιμοποιούν την host page cache για βελτιστοποίηση των λειτουργιών Ε/Ε.
Ο οδηγός μας υλοποιεί την παραπάνω σημασιολογία, με μια `sync()` κλήση συστήματος για όλο το σύστημα αρχείων στο οποίο βρίσκεται η ιεραρχική δομή καταλόγων και τα άρχεία του συστήματός μας, εφόσον βέβαια έχουν προηγηθεί κάποιες μεταβολές στα δεδομένα ή στα μεταδεδομένα του εικονικού δίσκου. Περισσότερα για το θέμα του συγχρονισμού με τον δίσκο και των κρυφών μνημών αναφέρουμε στην ενότητα 4.3.

bdrv_snapshot_create(): η συνάρτηση αυτή καλείται κάθε φορά που ο χρήστης αιτείται την δημιουργία ενός στιγμιοτύπου για έναν εικονικό δίσκο. Πέρα από την διαχείριση της επικεφαλίδας συνταγής του στιγμιοτύπου με γενικές πληροφορίες που ο QEMU προτείνει να διατηρούν όλα τα στιγμιότυπα, οι υπόλοιπες ενέργειες είναι ειδικές ανά τον εκάστοτε οδηγό.
Ο οδηγός μας, αφού ελέγξει ότι δεν υπάρχει στιγμιότυπο με το ίδιο όνομα, δημιουργεί μια επικεφαλίδα συνταγής που θα αντιπροσωπεύει το στιγμιότυπο, γράφει τις κατάλληλες επικεφαλίδες και στη συνέχεια δημιουργεί ένα νέο κόμβο-ρίζα και αντιγράφει σε αυτόν τα δεδομένα του κόμβου-του ρίζας πρωτεύοντος δίσκου φροντίζοντας να

θέσει όλα τα δικαιώματα των εγγραφών σε μόνο για ανάγνωση. Κατόπιν ενημερώνει την επικεφαλίδα συνταγής του πρωτεύοντος δίσκου με το νέο αυξημένο στιγμιότυπο.

bdrv_snapshot_goto(): η συνάρτηση αυτή καλείται κάθε φορά που ο χρήστης αιτείται την επαναφορά του εικονικού δίσκου στα δεδομένα που είχε κατά την λήψη ενός στιγμιοτύπου.
Ο οδηγός μας αφού ελέγξει ότι το στιγμιότυπο αναφέρεται στον ίδιο εικονικό δίσκο, αντιγράφει τις εγγραφές του ριζικού κόμβου του στιγμιοτύπου στον κόμβο-ρίζα του πρωτεύοντος δίσκου, θέτοντας όλα τα δικαιώματά σε μόνο για ανάγνωση, ώστε να μην αλλοιωθούν τα δεδομένα του στιγμιοτύπου.

bdrv_snapshot_delete(): η συνάρτηση αυτή καλείται κάθε φορά που ο χρήστης αιτείται την διαγραφή ενός στιγμιοτύπου ενός εικονικού δίσκου.
Ο οδηγός μας απλώς διαγράφει την επικεφαλίδα συνταγής του στιγμιοτύπου. Εάν είχαμε εφαρμόσει garbage-collection, εδώ θα διαγράφονταν όσοι κόμβοι δεν ήταν πλέον μοιραζόμενοι ή προσπελάσιμοι.

bdrv_snapshot_list(): η συνάρτηση αυτή καλείται κάθε φορά που ο χρήστης θέλει ναι δει μια λίστα με τα στιγμιότυπα που υπάρχουν για έναν εικονικό δίσκο.
Ο οδηγός μας δημιουργεί την λίστα με τα διαθέσιμα στιγμιότυπα για τον εικονικό δίσκο, διατρέχοντας τον ομώνυμο κατάλογο του εικονικού δίσκου, ο οποίος βρίσκεται στον κατάλογο recipe-store. Για κάθε επικεφαλίδα συνταγής που βρίσκει διαβάζει τα περιεχόμενά του SnapshotHeader που το πρώτο περιέχει, ώστε να εξάγει κάποιες χρήσιμες πληροφορίες, πέρα από το όνομα του στιγμιοτύπου. Από την παραπάνω διάσχιση εξαιρεί την επικεφαλίδα συνταγής του πρωτεύοντος δίσκου, βασιζόμενος στο όνομά του.

bdrv_get_info(): όταν ο χρήστης θέλει να δει κάποιες πληροφορίες του εικονικού δίσκου, η συνάρτηση αυτή καλείται μετά από την αντίστοιχη generic, για να τυπώσει πληροφορίες που αφορούν τις λεπτομέρειες της υλοποίησης και σχετίζονται συνήθως με την ιδιωτική δομή.
Ο οδηγός μας επιστρέφει το επιλεγμένο μέγεθος τεμαχίου.

Κάθε `BlockDriver` έχει ένα συγκεκριμένο όνομα (format_name) που περιγράφει την μορφή του εικονικού δίσκου και τον διαχωρίζει από τους υπόλοιπους οδηγούς. Εμείς δώσαμε στον δικό μας το όνομα dtree, αφορμώμενοι από την σύνθεση του deduplication και της δενδρικής δομή μας.

Τέλος, δημιουργήσαμε ένα στιγμιότυπο του `BlockDriver` και συνδέσαμε τα πεδία του, που αποτελούν συναρτήσεις του generic block drivers API, με την αντίστοιχη υλοποίηση κάθε συνάρτησης από τον οδηγό μας. Για να εισάγουμε τον οδηγό μας στο σύνολο οδηγών του QEMU, καλούμε την `bdrv_register()` του generic block drivers layer, δίνοντας ως όρισμα το παραπάνω στιγμιότυπο.

Ως προς την δημιουργία των κλώνων, προείπαμε ότι θεωρούνται ισοδύναμοι με πρωτεύοντες εικονικούς δίσκους που στην αρχή μοιράζονται το δέντρο ενός στιγμιοτύπου και εν συνεχεία διαφοροποιούνται πραγματοποιώντας σταδιακά COW. Για αυτό το σκοπό συμμορφωθήκαμε με το πρότυπο κατασκευής κλώνων του qcow2 (δηλαδή COW images), οπότε η διαδικασία δημιουργίας κλώνου από ένα στιγμιότυπο πραγματοποιείται στην συνάρτηση `bdrv_create()`, σε περίπτωση που ο χρήστης ορίσει με την επιλογή -b ένα αρχικό στιγμιότυπο (ή ένα αρχείο υποστήριξης στην ορολογία του qcow2). Η διαδικασία απλώς αντιγράφει τον κόμβο-ρίζα του στιγμιοτύπου εφαρμόζοντας δικαιώματα μόνο για ανάγνωση, και δημιουργεί μια νέα επικεφαλίδα συνταγής που δείχνει στον νέο ριζικό κόμβο του κλώνου, σε έναν νέο κατάλογο με το όνομα του κλώνου.

4.3 Κρυφές μνήμες και συγχρονισμός με δίσκο

Για να διαβάσει δεδομένα από τον δίσκο μια διεργασία του χώρου χρήστη, εκτελεί μια κλήση συστήματος `read()`, η οποία διασχίζοντας όλα τα επίπεδα της στοίβας Ε/Ε του Linux καταλήγει να διαβάζει από τον φυσικό δίσκο τους κατάλληλους τομείς και με μια DMA λειτουργία να γεμίζει κάποιες σελίδες στην host page cache με τα δεδομένα.

Για να γράψει δεδομένα στον δίσκο μια διεργασία του χώρου χρήστη, εκτελεί μια κλήση συστήματος `write()`, η οποία καταλήγει αυτή τη φορά να αντιγράφει δεδομένα από ένα userspace buffer, όχι στον φυσικό δίσκο αλλά σε σελίδες (pages) στην page cache [Ker10]. Το πότε τελικά θα δημιουργηθεί μια αίτηση Ε/Ε και θα γίνει η DMA λειτουργία για την μεταφορά των δεδομένων σε τομείς του φυσικού δίσκου εξαρτάται από το πότε ο πυρήνας ή κάποιος χρήστης αιτηθεί να συγχρονιστούν οι βρώμικες σελίδες της page cache στον δίσκο.

Ο οδηγός του εικονικού δίσκου που υλοποιήσαμε, είναι κατ᾽ ουσία μια τέτοια διεργασία που τρέχει στον χώρο χρήστη του host και καταλήγει να διαβάζει ή να γράφει

δεδομένα από τον ή στον φυσικό δίσκο του host, προσομοιώνοντας έτσι έναν φυσικό δίσκο για τον guest. Επειδή όμως μέχρι να γίνει αυτό, μεσολαβεί η page cache του host, ο οδηγός του εικονικού μας δίσκου μερικές φορές μπορεί να μην χρειαστεί να διαβάσει δεδομένα από τον host φυσικό δίσκο, μιας και αυτά υπάρχουν ήδη στην host page cache. Αυτή είναι μια επιθυμητή λειτουργία αφού έχουμε επιτάχυνση στην μεταφορά δεδομένων σε σχέση με έναν φυσικό δίσκο. Από την άλλη, σε μια αίτηση εγγραφής η κλήση συστήματος `write()` που εκτελεί ο οδηγός μας επιστρέφει μόλις τα δεδομένα εισαχθούν σε σελίδες της host page cache. Παρόλο που και πάλι έχουμε επιτάχυνση σε σχέση με την εγγραφή δεδομένων στον φυσικό δίσκο, εδώ εγείρεται το ερώτημα της σημασιολογίας μιας λειτουργίας εγγραφής του εικονικού μας δίσκου. Αυτό γιατί προσομοιώνοντας ένα φυσικό δίσκο, ο οδηγός του εικονικού μας δίσκου, οφείλει να εγγυάται ότι από την στιγμή που θα ολοκληρωθεί η ενέργεια εγγραφής, όλα τα δεδομένα έχουν γραφτεί σε με ένα μόνιμο μη-πτητικό τρόπο. Επειδή όμως η host page cache διατηρείται στη RAM, ένα σχήμα στο οποίο ο οδηγός μας την εκμεταλλεύεται για να ολοκληρώσει σε αυτήν ένα αίτημα εγγραφής, καταστρατηγεί την προηγούμενη εγγύηση. Στην πραγματικότητα της σύγχρονης βιομηχανίας δίσκων όμως, το σημασιολογικό πλαίσιο που περιγράψαμε δεν είναι τόσο αυστηρό και η εγγύηση μόνιμης εγγραφής των δεδομένων δεν είναι πάντα αληθής ακόμα και για τους φυσικούς δίσκους.

Πολλές φορές οι σύγχρονοι δίσκοι για ακόμα μεγαλύτερη βελτιστοποίηση της απόδοσης και για απεξάρτηση από την δρομολόγηση των λειτουργιών Ε/Ε που επιτελεί ο πυρήνας, έχουν ενσωματωμένη μια δική τους κρυφή μνήμη (disk cache) η οποία είναι πολύ μικρότερη (<128MB) και πολύ ταχύτερη (ταχύτητες RAM) από το φυσικό δίσκο. Η μόνιμη αποθήκευση των δεδομένων στους τομείς του φυσικού δίσκου, διεκπεραιώνεται σε κάποια μεταγενέστερη και πιο βολική στιγμή. Έτσι, ακόμα και όταν ο ελεγκτής του φυσικού δίσκου λάβει μια οδηγία για εγγραφή δεδομένων σε τομείς του δίσκου, δεν αποθηκεύει τα δεδομένα στον μη-πτητικό φυσικό δίσκο αλλά σε μια πτητική cache, της οποίας τα δεδομένα επίσης χάνονται σε περίπτωση αποτυχίας (π.χ. διακοπή τροφοδοσίας). Η cache αυτή χρησιμοποιείται επίσης για επιτάχυνση των αναγνώσεων από τον δίσκο, αφού σε μια αίτηση ανάγνωση μπορούν να μεταφερθούν εκ των προτέρων στην disk cache και διπλανοί τομείς που είναι πιθανό να ζητηθούν σύντομα (prefetching).

Οι παραπάνω αναβολές στην εγγραφή των δεδομένων στον φυσικό δίσκο γίνονται

γιατί, όπως προαναφέραμε στην αρχή, η μετακίνηση της κεφαλής εγγραφής/ανάγνωση ενός HDD είναι μια χρονοβόρα διαδικασία που μπορεί να κυριαρχήσει στο latency των αιτημάτων. Έτσι σε διάφορα επίπεδα υπάρχει εκμετάλλευση του buffering και μιας πολιτικής αναδιάταξης των αιτημάτων εγγραφής, έτσι ώστε όσο είναι εφικτό, η κεφαλή να μετακινείται προς μια συγκεκριμένη κατεύθυνση (εντός/εκτός) και σε κοντινά σημεία ανά αίτηση.

Συνεπώς από τα παραπάνω προκύπτει ότι ο οδηγός ενός εικονικού δίσκου, έχει την δυνατότητα για αύξηση της απόδοσης να χρησιμοποιήσει την host page cache, η οποία στην πραγματικότητα θα λειτουργεί σαν μια πολύ μεγάλη εικονική disk cache για τον εικονικό δίσκο.

Η παραπάνω δυνατότητα ισχύει για κάθε οδηγό εικονικού δίσκου και έτσι οι developers του QEMU αποφάσισαν να προτυποποιήσουν τις διάφορες επιλογές συμμετοχής της host page cache. Ο χρήστης λοιπόν που κατά την εκκίνηση του qemu εισάγει ένα εικονικό δίσκο (με την επιλογή -drive) , μπορεί μέσω της επιλογής -cache, να ορίσει τον τρόπο χρήσης της host page cache που επιθυμεί. Οι επιλογές είναι οι εξής:

writethrough η host page cache είναι ενεργοποιημένη και η πραγματική disk cache είναι απενεργοποιημένη. Αυτό σημαίνει ότι οι αιτήσεις εγγραφής οφείλουν να γράφονται και στον φυσικό δίσκο πέρα από την page cache, ακολουθώντας την σημασιολογία `O_SYNC` [Ker10]. Έτσι επιταχύνονται όσες αιτήσεις ανάγνωσης βρουν τα δεδομένα τους στην page cache, αλλά αυξάνεται το latency των εγγραφών, αφού κάθε αίτηση πρέπει να καταλήγει στον πολύ αργό φυσικό δίσκο.

writeback η host page cache είναι ενεργοποιημένη και η πραγματική disk cache είναι ενεργοποιημένη. Αυτό σημαίνει ότι οι αιτήσεις εγγραφής θεωρούνται ολοκληρωμένες μόλις ολοκληρωθεί η μεταφορά των δεδομένων στην host page cache. Με αυτήν την επιλογή επιταχύνονται τόσο οι αναγνώσεις όσο και οι εγγραφές αλλά τα δεδομένα δεν είναι διασφαλισμένα σε περίπτωση αστοχίας.

none η host page cache είναι απενεργοποιημένη και η πραγματική disk cache είναι ενεργοποιημένη. Αυτό σημαίνει ότι όλες οι αιτήσεις προσπερνούν την host page cache και άρα οι user space buffers του qemu που αφο-

ρούν τα δεδομένα της guest αίτησης, αλληλεπιδρούν κατευθείαν με τον δίσκο, ακολουθώντας την σημασιολογία `O_DIRECT` [Ker10]. Έτσι βελτιστοποιούνται στο μέγιστο οι αιτήσεις εγγραφής, εάν υπάρχει μεγάλη disk cache στον φυσικό δίσκο, ειδάλλως η απόδοση υποβαθμίζεται. Οι αιτήσεις ανάγνωσης υστερούν, αφού δεν μπορούν να χρησιμοποιήσουν την host page cache.

unsafe όπως το writeback, με την διαφορά ότι αγνοούνται εντολές για μεταφορά των δεδομένων από την disk cache στον πραγματικό δίσκο, όπως θα δούμε παρακάτω. Δεν προσφέρει εγγύηση για την ακεραιότητα και την μόνιμη αποθήκευση των δεδομένων και θα πρέπει να χρησιμοποιείται μόνο προσωρινά, όταν υπάρχει ανάγκη για πολύ ψηλή απόδοση (π.χ. guest install)

directsync συνδυασμός των σημασιολογιών `O_SYNC` και `O_DIRECT`, όπου οι αιτήσεις παρακάμπτουν την host page cache, ενώ οι εγγραφές θεωρούνται ολοκληρωμένες μόνο όταν γραφούν στους τομείς του φυσικού δίσκου

Στην προκειμένη περίπτωση, ο οδηγός για την μορφή του εικονικού δίσκου που υλοποιήσαμε επιτρέπει μόνο τις επιλογές writeback, writethrough και unsafe, χωρίς να λαμβάνουμε ειδική μέριμνα για να υλοποιήσουμε τις επιλογές none και directsync που απαιτούν `O_DIRECT` σημασιολογία, για λόγους πολυπλοκότητας. Αυτό γιατί οι κλήσεις συστήματος write με `O_DIRECT` σημασιολογία εμπλέκουν κάποιες στρυφνές προϋποθέσεις όπως ειδική ευθυγράμμιση στις περιοχές μνήμης που εμπλέκονται. Προτείνεται μάλιστα ένθερμα η χρήση της writeback mode για λόγους απόδοσης που θα εξηγήσουμε στην συνέχεια.

Ένα μείζον θέμα που φυσικά προκύπτει από τις disk caches, είναι πως ακόμα και στα περιβάλλοντα που απουσιάζει η εικονικοποίηση και έχουμε απλώς φυσικούς δίσκους με πραγματικές disk caches, αναδύεται κάποιες φορές η ανάγκη τα δεδομένα να αποθηκευτούν με ένα μόνιμο τρόπο στον κανονικό δίσκο και όχι στην disk cache του, έτσι ώστε να είναι εξασφαλισμένα σε περίπτωση αποτυχίας. Αυτή είναι μια εγγύηση που θέλουν πολλές εφαρμογές και λειτουργίες του λειτουργικού συστήματος, οπότε προέκυψε ένα μοντέλο με το οποίο μια εγγραφή στο δίσκο μπορεί, αν εξαναγκαστεί, να γράψει τα δεδομένα στους κανονικού τομείς του, παρακάμπτοντας την disk cache. Με βάση αυτό το μοντέλο υπάρχουν 2 τρόποι με τους οποίους η στοίβα Ε/Ε του Linux

δίνει στα συστήματα αρχείων την δυνατότητα να παρακάμψουν μια disk cache που πιθανώς να υπάρχει στον δίσκο. Αυτοί είναι το explicit cache flush και οι FUA σημαίες στις αιτήσεις Ε/Ε. Σύμφωνα με αυτούς κατά την δημιουργία ενός bio, τίθεται η σημαία `REQ_FLUSH` (για explicit cache flush) ή `REQ_FUA` (για FUA) στο αντίστοιχο πεδίο σημαιών. Έτσι διασφαλίζεται ότι η disk cache θα γράψει τα δεδομένα της στον δίσκο πριν ολοκληρωθεί η αίτηση Ε/Ε. Η `REQ_FLUSH` σημαία μπορεί να τεθεί και σε ένα κατά τα άλλα άδειο bio, μόνο και μόνο για να υλοποιήσει την λειτουργία μεταφοράς των δεδομένων του δίσκου από την disk cache στους κανονικούς τομείς.

Ο τρόπος με τον οποίο υλοποιείται στους φυσικούς δίσκους η σημασιολογία του παραπάνω μοντέλου του Linux για άδειασμα (flush) των disk caches, είναι συνήθως μέσω μιας ειδικής εντολής που υποστηρίζει η εκάστοτε διεπαφή (interface). Έτσι μόλις ο οδηγός συσκευής λάβει ένα bio με κάποια από τις 2 σημαίες που οφείλουν να πυροδοτήσουν flush, στέλνει μια ειδική εντολή στον ελεγκτή του δίσκου, ο οποίος αναλαμβάνει να αδειάσει τα περιεχόμενα της cache. Εάν το bio αφορά και αίτηση εγγραφής τότε αυτή πραγματοποιείται παρακάμπτοντας την disk cache. Για παράδειγμα στο πρότυπο SCSI, η εντολή αυτή είναι η `SYNCHRONIZE CACHE (10)` με τον δεκαεξαδικό κωδικό 35 ενώ στο SATA η `ATA_CMD_FLUSH` ή `ATA_CMD_FLUSH_EXT`.

Ο τρόπος με τον οποίο οι διεργασίες μπορούν να αιτηθούν την εγγύηση της μηπτητικής αποθήκευση των δεδομένων τους, είναι μέσω των κλήσεων συστήματος `sync(), syncfs(), fsync(), fdatasync()`. Οι δύο πρώτες αφορούν τα βρώμικα (dirty) δεδομένα ολόκληρου του συστήματος αρχείων που υπάρχουν στην page cache, ενώ οι επόμενες αφορούν τα βρώμικα δεδομένα ενός συγκεκριμένου αρχείου που υπάρχουν στην page cache. Επειδή οι κλήσεις αυτές υλοποιούνται από το εκάστοτε σύστημα αρχείων, είναι ευθύνη του συστήματος αρχείων να χρησιμοποιήσει το παραπάνω μοντέλο ώστε να εγγυηθεί όχι απλώς την μεταφορά των βρώμικων, σελίδων από την page cache στον δίσκο, αλλά και την εγγραφή τους στους φυσικούς τομείς του δίσκου και όχι στην disk cache. Το ext4 στην παρούσα μορφή του, πράγματι, για κάθε μια από τις παραπάνω κλήσεις χρησιμοποιεί ένα explicit cache flush και εγγυάται την μη-πτητική αποθήκευση των δεδομένων.

Μεταπηδώντας πάλι στο δικό μας σενάριο εικονικοποίησης, είναι απαραίτητο να μεταφέρουμε την παραπάνω σημασιολογία και στον εικονικό μας δίσκο. Έτσι, κάθε φορά που το guest λειτουργικό σύστημα κάνει μια κλήση `fsync(), sync()` κτλ, το σύ-

στημα αρχείων του guest αναλαμβάνει να δημιουργήσει ένα bio με την κατάλληλη `REQ_FLUSH` σημαία και το οποίο θα αναγκάσει τον guest frontend οδηγό του εικονικού δίσκου να εκδώσει μια εντολή τύπου `SYNCHRONIZE CACHE`, ανάλογα με της εκάστοτε διεπαφής. Από αυτό το σημείο και μετά αναλαμβάνει ο QEMU, ο οποίος λαμβάνει την εντολή αυτήν που δόθηκε για το προσομοιούμενο `DeviceState`, την αποκωδικοποιεί με βάση τον κώδικα που προσομοιώνει την διεπαφή αυτού του και μεταφέρει την ανάγκη για flush στο `BlockBackend`. Γι' αυτό το σκοπό, καλεί την `blk_flush()` η οποία καλεί την generic `bdrv_flush()`, η οποία μπαίνει σε coroutine context μέσω της `bdrv_co_flush()`. Μπορούμε να καταλήξουμε στην ίδια συνάρτηση μέσω άλλων aio ή coroutine εκδοχών των συναρτήσεων αλλά και πάλι διασχίζοντας το block backend layer και το block drivers generic layer. Τελικά στην `bdrv_co_flush` καλείται η specific `bdrv_flush_to_disk()` συνάρτηση του οδηγού μας, ενώ για άλλες μορφές εικονικών δίσκων υπάρχει μέριμνα ώστε να γίνουν flush και δεδομένα αρχείων υποστήριξης ή caches που πραγματοποιούν οι οδηγοί εικονικών δίσκων στην μνήμη τους.

Το παραπάνω συμβαίνει σε cache=writeback mode, όπου θέλουμε τα δεδομένα να φτάνουν στον δίσκο μόνο μετά από ένα ρητό αίτημα του guest. Σε cache=writethrough mode η generic `bdrv_flush()` καλείται μετά από κάθε αίτηση εγγραφής που φτάνει στον QEMU, οπότε υλοποιεί την απαίτηση για απαραίτητη εγγραφή των δεδομένων στον δίσκο μετά από κάθε αίτηση εγγραφής στον guest. Σε cache=unsafe mode η `bdrv_flush()` απλώς δεν πραγματοποιεί καμία ενέργεια.

Εν ολίγοις, ο οδηγός μας θα πρέπει να υλοποιήσει κατάλληλα την `bdrv_flush_to_disk()` ούτως ώστε τα δεδομένα να φτάνουν στους τομείς του φυσικού δίσκου και να είναι εξασφαλισμένα σε περίπτωση αποτυχίας. Προφανώς γι' αυτόν τον σκοπό το μόνο που χρειαζόμαστε είναι να καλέσουμε μια εκ των `fsync(), fdatasync(), sync(), syncfs()` και το λειτουργικό του host θα αναλάβει την μεταφορά των δεδομένων στον φυσικό δίσκο του host με τον ίδιο τρόπο που περιγράψαμε παραπάνω.

Μια πρώτη προσέγγιση θα ήταν να καλέσουμε μια εκ των `fsync(), fdatasync()` για όλα τα αρχεία που έχουν υποστεί αλλαγές και άρα είναι βρώμικα. Αυτά μπορεί να είναι τεμάχια ή κόμβοι των οποίων τα δεδομένα άλλαξαν εξαιτίας μιας αίτησης εγγραφής από τον guest. Επειδή όμως σε έναν μεγάλο δίσκο με μικρό μέγεθος τεμαχίου, μπορεί να έχουν αλλάξει χιλιάδες αρχεία στο μεσοδιάστημα των αιτήσεων του

guest, αυτή η προσέγγιση καταντάει απελπιστικά αργή. Αυτό συμβαίνει τόσο λόγω της επιβάρυνσης τόσων πολλών κλήσεων συστήματος όσο και κυρίως γιατί κάθε τέτοια κλήση οφείλει να γράψει στο δίσκο πολλά διαφορετικά μπλοκ, περιμένοντας την ολοκλήρωση της εγγραφής τους. Εδώ επικεντρωνόμαστε στην περίπτωση του ext4, το οποίο και χρησιμοποιήσαμε στην υλοποίησή μας, αλλά παρεμφερή συμπεράσματα ισχύουν και για άλλα συστήματα αρχείων. Για κάθε αρχείο πρέπει να ενημερωθεί το inode table, τα data block του αρχείου, το superblock και πιθανώς τα inode και data bitmaps εάν έχουμε και δημιουργία αρχείου. Αυτά τα μπλοκ είναι απομακρυσμένα στον φυσικό δίσκο, ενώ πρέπει να υπάρξει και εγγραφή στο journal, οπότε όλα τα παραπάνω οδηγούν σε πολλές φυσικά απομακρυσμένες αιτήσεις Ε/Ε, που προκαλούν συνεχείς κινήσεις της κεφαλής του δίσκου και άρα σημαντική επιβράδυνση. Αυτή δεν μπορεί να μετριαστεί από κατάλληλη συγχώνευση αιτημάτων ή δρομολόγηση από τον χρονοδρομολογητή Ε/Ε, από την στιγμή που κάθε flush πρέπει να εκτελεστεί σύγχρονα. Εκτελέσαμε κάποιες δοκιμές υιοθετώντας αυτήν την προσέγγιση και τα αποτελέσματα ήταν περισσότερο από απογοητευτικά. Επίσης απαιτείται και μνήμη, αφού είτε πρέπει να αφήνουμε ανοικτά τα αρχεία που έχουν υποστεί αλλαγές μέχρι να κληθεί η `bdrv_flush_to_disk()`, είτε να κρατήσουμε σε μια λίστα τα ονόματά τους και μετά να τα ξανανοίγουμε ένα-ένα και να καλούμε την `fdatasync()`.

Για να αποφύγουμε τα παραπάνω προβλήματα, αποφασίσαμε να εκτελούμε την κλήση συστήματος `sync()`, μεταφέροντας στον δίσκο όλα τα βρώμικα δεδομένα του συστήματος αρχείων. Αν και αυτό εκ πρώτης όψεως μοιάζει χειρότερη επιλογή, μιας και μπορεί στο σύστημα αρχείων να επενεργούν και άλλες εφαρμογές και άρα αναπόφευκτα θα μεταφέρονται και τα δικά τους δεδομένα, έχει θεαματικά καλύτερη απόδοση, γιατί είτε απαλείφει πλεοναστικές ενημερώσεις είτε τις ομαδοποιεί μειώνοντας τον αριθμό των λειτουργιών Ε/Ε. Το superblock και τα inode και data bitmaps ενημερώνονται μόνο μία φορά. Τα inode tables ενημερώνονται μαζικά με μία ή έστω λίγες λειτουργίες Ε/Ε για όλα τα αρχεία, αφού οι τομείς που τα περιέχουν είναι συνεχόμενα τοποθετημένοι στο δίσκο για ένα μεγάλο αριθμό από inodes. Έτσι αντί για 131.000 αιτήσεις Ε/Ε, έχουμε μόλις 1. Τα data blocks που πρέπει να ενημερωθούν είναι τα ίδια στον αριθμό, αλλά είναι πολύ πιθανό πολλά από αυτά να είναι σειριακά τοποθετημένα και άρα ο χρονοδρομολογητής Ε/Ε να συγχωνεύσει τις αντίστοιχες αιτήσεις Ε/Ε. Τα δεδομένα του journal επίσης γράφονται με λίγες λειτουργίες Ε/Ε και όχι με 131.000. Ακόμη, είναι πιθανό να εφαρμοστεί κατάλληλη πολιτική από τον ελεκτή του δίσκου,

ώστε όλα αυτοί οι τομείς να ενημερωθούν με μια σειρά που θα είναι βολική για τον δίσκο. Αυτή η δρομολόγηση δεν ήταν δυνατή νωρίτερα, όταν ο δίσκος έπρεπε να ολοκληρώσει λίγες διάσπαρτες αιτήσεις Ε/Ε, πριν καλέσουμε την επόμενη `fdatasync()`. Με το μαζικό flush γλιτώνουμε επίσης και την επιβάρυνση των κλήσεων συστήματος, που δεν είναι απαραίτητα αμελητέα για ένα τόσο μεγάλο αριθμό κλήσεων.

Προφανώς μπορεί το μαζικό flush να συνοψίζει τις αλλαγές στα αρχεία και επιτρέπει καλύτερη αναδιάταξη των εγγραφών στον φυσικό δίσκο, αλλά έχει και το τίμημά του. Σε ένα περιβάλλον υπολογιστικού νέφους όπου το σύστημα αρχείων του host μας θα διατηρεί αρχεία κόμβων/τεμαχίων για πολλούς δίσκους, μια κλήση `sync()` από τον ένα εικονικό δίσκο θα αναγκάζει και τα δεδομένα του άλλου να συγχρονιστούν, αποδυναμώνοντας το προηγούμενο επιχείρημα υπέρ της χρήσης του `sync()` για όλο το σύστημα αρχείων αντί των `fsync()/fdatasync()`. Η παραπάνω παρενέργεια δεν είναι αμελητέα, αφού καταστρατηγεί ένα βασικό χαρακτηριστικό της εικονικοποίησης, το οποίο είναι η απομόνωση και ανεξαρτησία του ενός VM από τα υπόλοιπα. Εδώ έχουμε σχέσεις αλληλεξάρτησης που σε ακραίες περιπτώσεις μπορεί να οδηγήσουν σε αυξημένη καθυστέρηση στην πρόσβαση στο δίσκο για ένα VM, μόνο και μόνο επειδή το διπλανό του (στον ίδιο host) εκτελεί ένα μακροσκελές flush.

Χρησιμοποιούμε πάντως το μαζικό flush των δεδομένων ολόκληρου του συστήματος αρχείων ως μοναδική βιώσιμη επιλογή για τον οδηγό μας.

Ως πρόσθετη επέκταση, αναφέρουμε ότι θα μπορούσε να εισαχθεί μεγαλύτερη ευελιξία ως προς το σύστημα αρχείων στο οποίο κάνουμε `sync()`. Η παρούσα ιεραρχική δομή που περιγράψαμε, υποθέτει ότι τόσο τα αρχεία κόμβων όσο και τα αρχεία τεμαχίων βρίσκονται στο ίδιο σύστημα αρχείων. Μια επέκταση στην οποία για επιτάχυνση οι κόμβοι θα βρισκόντουσαν σε ένα ξεχωριστό SSD, απαιτεί ένα μηχανισμό ώστε να ανιχνεύονται και να εκτελούνται δυο διαφορετικά flush στα 2 διαφορετικά συστήματα αρχείων. Αυτό είναι πολύ εύκολο να υλοποιηθεί, με την χρήση μάλιστα της `syncfs()`.

4.4 Κλειδώματα και ταυτοχρονισμός

Προβλήματα ταυτοχρονισμού προκύπτουν όταν περισσότερες από μια οντότητες έχουν πρόσβαση σε κοινά δεδομένα. Οπότε στην περίπτωσή μας ας ορίσουμε πρώτα τις οντότητες και τα μοιραζόμενα δεδομένα, πριν προσπαθήσουμε να αντιμετωπίσουμε

καταστάσεις όπου η ταυτόχρονη προσπέλαση θα οδηγήσει σε ασυνεπή δεδομένα ή ασυνεπή εικόνα μιας από τις οντότητες για την κατάσταση των δεδομένων.

Αρχικά αναφέρουμε ότι εν γένει δεν είναι δυνατόν δυο διαφορετικές εικονικές μηχανές να έχουν πρόσβαση στον ίδιο εικονικό δίσκο κατά την διάρκεια λειτουργίας τους, εκτός και αν τον προσπελάσουν με κάποιο σύστημα αρχείων (π.χ. NFS) ή storage system (π.χ rados) το οποίο αναλαμβάνει να επιλύσει ζητήματα σειριοποίησης της πρόσβασης στο δίσκο. Σε κάθε άλλη περίπτωση, τα δεδομένα του δίσκου θα αλλοιώνονταν, αφού οι δυο εικονικές μηχανές όχι μόνο θα αλλοίωναν με αμοιβαίο τρόπο η μία τα δεδομένα της άλλης, αλλά θα είχαν στις page caches τους και διαφορετικές εικόνες του εικονικού δίσκου. Έτσι, παραγκωνίζοντας τα πρωτόκολλα που επιτρέπουν διαμοιρασμό του εικονικού δίσκου, για τα οποία υπάρχουν και κατάλληλοι οδηγοί στον QEMU (nfs, sheepdog, rados), θεωρούμε ότι κάθε εικονικός δίσκος είναι λογικά συνδεδεμένος με το πολύ ένα running VM.

Άρα οι διάφορες οντότητες που μπορεί ταυτόχρονα να έχουν πρόσβαση σε κάποια κοινά δεδομένα είναι:

1. Μια διεργασία qemu-system για την προσομοίωση ενός VM (π.χ. qemu-sysem-x86_64). Προείπαμε ότι μόνο μια τέτοια διεργασία μπορεί να υπάρχει ανά εικονικό δίσκο. Είναι δυνατόν όμως να πεπλεχθούν πολλές αιτήσεις Ε/Ε του guest για τον ίδιο εικονικό δίσκο.
2. Μια διεργασία qemu-img για την δημιουργία, επαναφορά, διαγραφή και απαρίθμηση στιγμιοτύπων. Εδώ μπορεί να έχουμε περισσότερες από μια διεργασίες τύπου qemu-img.
3. Μια διεργασία qemu monitor για την αποθήκευση της κατάστασης μιας εικονικής μηχανής, διαδικασία που καταλήγει σε λήψη στιγμιοτύπου. Δυνατή σε ένα στιγμιότυπο.
4. Μια διεργασία για την δημιουργία και διαγραφή εικονικών δίσκων (πρωτεύοντες ή κλώνοι), και πάλι δυνατή σε πολλά στιγμιότυπα
5. Μια offline διεργασία απαλοιφής διπλοτύπων που θα προσπελάζει τους immutable κόμβους/τεμάχια που έχουν προκύψει από στιγμιότυπα
6. Μια διεργασία συγχρονισμού των περιεχομένων ενός εικονικού δίσκου με ένα απομακρυσμένο αντίγραφο ασφαλείας του

Τα κοινά δεδομένα στα οποία οι παραπάνω οντότητες μπορούν να έχουν πρόσβαση είναι: οι επικεφαλίδες συνταγής, οι κόμβοι και το αρχείο info (κρατάει το επόμενο διαθέσιμο uid των δίσκων).

Παραδόξως, τα τεμάχια δεν αντιμετωπίζουν πρόβλημα ταυτόχρονων προσπελάσεων. Η μοναδική οντότητα που μπορεί να προσπελάζει τα τεμάχια είναι ένα στιγμιότυπο της διεργασίας qemu-system, αφού η διεργασία απαλοιφής διπλοτύπων προσπελάζει τεμάχια που είναι μόνο για ανάγνωση και χωρίς να αλλάζει τα περιεχόμενά τους, ενώ η qemu-img για την δημιουργία και διαγραφή στιγμιοτύπων δεν επιδρά με τεμάχια. Ακόμα και η συλλογή σκουπιδιών που θα διέγραφε το τεμάχιο, στην χειρότερη περίπτωση θα μπορούσε να αποτύχει στην διαγραφή όταν μια qemu-system διεργασία είχε ανοικτό τον file descriptor. Η μοναδική διεργασία που ίσως χρειάζεται lock, θα ήταν η διεργασία απομακρυσμένου συγχρονισμού, ώστε να προστατέψουμε την περίπτωση που η αντιγραφή θα διακόπτονταν από κάποια εγγραφή στο αρχείου τεμαχίου και έτσι τα δεδομένα που θα αντιγράφονταν θα ήταν ασυνεπή. Η διεργασία συγχρονισμού όμως, είναι συνηθέστερο να συγχρονίζει ένα στιγμιότυπο, ενώ ακόμα και στην περίπτωση που συγχρονίζει έναν ζωντανό εικονικό δίσκο που επιτρέπεται να δέχεται παράλληλα λειτουργίες Ε/Ε, είναι δυνατόν να παρακάμψει την ασυνέπεια, λαμβάνοντας ένα write lock στο entry που δείχνει στο τεμάχιο που αντιγράφεται, όπως και γίνεται.

Να σημειώσουμε ότι η ιδιωτική δομή `BDRVDtreeState`, που ανάμεσα στα υπόλοιπα διατηρεί και πληροφορίες της επικεφαλίδας συνταγής, δεν αποτελεί πραγματικά μοιραζόμενο πόρο, αφού κάθε qemu-img ή qemu-system διεργασία που προσπελάζει τον εικονικό δίσκο, δημιουργεί μια τέτοια ιδιωτική δομή κατά την έναρξή της και την έχει στον ιδιωτικό χώρο μνήμης της για όσο τρέχει. Παρακάτω θα εστιάσουμε στο θέμα που ανακύπτει από την ανάγκη διαμοιρασμού των πληροφοριών αυτής της δομής.

Επίσης, ο προτυποποιημένος τρόπος που ακολουθεί το QEMU για λήψη online στιγμιοτύπων είναι η δημιουργία ενός COW image που με αυξητικό τρόπο θα εμπερικλείει τις αλλαγές από την αρχική εικόνα, η οποία πλέον γίνεται αρχείο υποστήριξης κατ’ αντιστοιχία με όσα παρουσιάσαμε στην 3.2. Η online διαδικασία λήψης, εκκινείται από την qemu monitor διεργασία. Ο σχεδιασμός μας επιλέγει να μην ακολουθήσει αυτό το αυξητικό πρότυπο στιγμιοτύπων, καθώς δεν ενθαρρύνει την έννοια της εξαρτώμενης COW εικόνας. Αντιθέτως, υποστηρίζει online στιγμιότυπα κανονικά μέσω

της χρήσης της qemu-img διεργασίας, με τον ίδιο τρόπο που λαμβάνονται και offline στιγμιότυπα, με την διαφορά ότι τώρα ότι η εικονική μηχανή θα είναι ενεργή. Η qemu monitor καλό είναι να χρησιμοποιηθεί βέβαια, για να παγώσει όλες τις λειτουργίες Ε/Ε στον guest.

Επειδή τόσο η qemu-img όσο και η qemu monitor καταλήγουν να καλούν την συνάρτηση του οδηγού μας για λήψη στιγμιοτύπων, κάθε επόμενη αναφορά σε qemu-img διεργασία, υπονοεί ισοδύναμα την πιθανότητα ύπαρξης μια qemu monitor διεργασίας.
Τέλος, η μόνη άλλη διεργασία που είναι πιθανό να επιχειρήσει να προσπελάσει τον εικονικό δίσκο είναι η QEMU Guest Agent (qemu-ga), η οποία επιτρέπει στον host να εκδώσει κάποιες εντολές προς τον guest, ώστε να εξασφαλιστεί η ομαλή συνεργασία τους για την αποπεράτωση κάποιων ενεργειών. Και πάλι η χρήση του guest agent μπορεί να χρησιμοποιηθεί για λήψη online στιγμιοτύπων, οπότε και αυτή η περίπτωση ανάγεται στην χρήση μιας διεργασίας qemu monitor ή qemu-img.

Όπως βλέπουμε και οι τρεις τύποι δεδομένων που μπορεί να προσπελαστούν ταυτόχρονα αφορούν αρχεία στο σύστημα αρχείων του host, των οποίων η εγγραφή και ανάγνωση περνάει διαμέσου της host page cache. Έτσι οφείλουμε να προστατεύσουμε αυτά τα αρχεία από ταυτόχρονη προσπέλαση ή ταυτόχρονη μεταβολή των δεδομένων τους. Στις περιπτώσεις που μια οντότητα τροποποιεί δεδομένα, καμία άλλη οντότητα δεν θα πρέπει να μπορεί ούτε να τα τροποποιήσει ούτε να τα προσπελάσει, ενώ σε περίπτωση που μια ή περισσότερες οντότητες διαβάζουν δεδομένα, καμία άλλη οντότητα δεν θα πρέπει να μπορεί να τα τροποποιήσει. Για να το πετύχουμε αυτό σε επίπεδο αρχείων χρησιμοποιούμε τα POSIX fcntl locks [Ker10].

fcntl locks

Αποτελούν advisory locks, υπό την έννοια ότι απαιτούν την συνεργασία όλων των διεργασιών που εμπλέκονται με τα αρχεία που κλειδώνονται. Ένα fcntl lock σε αντίθεση με άλλες τεχνικές mandatory locking, δεν θα αποτρέψει την προσπέλαση και τροποποίηση του αρχείου από μια διεργασία που δεν θα ελέγξει πρώτα εάν το αρχείο είναι κλειδωμένο. Κάθε εμπλεκόμενη διεργασία λοιπόν, οφείλει να σέβεται τα κλειδώματα (locks), κάνοντας αίτηση για κλείδωμα πριν προσπελάσει ή τροποποιήσει το αρχείο. Τα fcntl locks επιτρέπουν το κλείδωμα όχι μόνο ολόκληρου του αρχείου αλλά και περιοχών bytes αυτού, γεγονός που μας είναι χρήσιμο ώστε να εφαρμόσουμε το

κλείδωμα στα απολύτως απαραίτητα δεδομένα και να περιορίσουμε τον παραλληλισμό όσο λιγότερο γίνεται. Υπάρχουν δυο τύποι κλειδωμάτων, το read και το write lock, και είναι αντίστοιχοι με την σημασιολογία των shared και exclusive lock. Πολλές διεργασίες μπορούν να έχουν ένα read lock σε μια περιοχή δεδομένων αλλά μόνο μια διεργασία μπορεί να έχει write lock, αποκλείοντας άλλες διεργασίας από το να πάρουν read ή write locks.
Μια διεργασία λοιπόν, μπορεί να αιτηθεί να πάρει ένα write lock, να πάρει ένα read lock, να αναβαθμίσει κάποιο read lock σε write ή να ξεκλειδώσει κάποιο lock, όλα για μια περιοχή δεδομένων. Η αίτηση αυτή μπορεί να γίνει σε blocking mode (`F_SETLKW`), όπου, εάν η απόκτηση του κλειδώματος δεν είναι δυνατή η διεργασία θα μπλοκάρει μέχρι να το αποκτήσει, ή σε non-blocking mode, όπου η συνάρτηση λήψης του κλειδώματος θα επιστρέψει με κωδικό αποτυχίας `-EAGAIN` (`F_SETLK`). Υιοθετούμε την πρώτη προσέγγιση, αφού η διεργασία που αναμένει να πάρει το κλείδωμα ούτε μπορεί να εκτελέσει κάποιο άλλο χρήσιμο έργο, ούτε φιλοδοξούμε να εφαρμόσουμε μια πολιτική παραχώρησης lock σε διεργασίες, διαφορετική από την επιλογή του δρομολογητή του πυρήνα του host. Επαφιόμαστε σε αυτόν ώστε να μην υπάρξουν επίσης περιπτώσεις λιμοκτονίας.

Θα πρέπει να είμαστε προσεκτικοί για να αποφύγουμε αδιέξοδα (deadlocks), ενώ μια κλήση `close()` σε οποιονδήποτε από τους file descriptors του αρχείου που έχει ένα κλείδωμα, οδηγεί στην ελευθέρωση όλων των κλειδωμάτων για το αρχείο, ακόμα και αν αυτά έχουν ληφθεί μέσω διαφορετικού file descriptor. Καμία από αυτές τις περιπτώσεις δεν παρουσιάζεται στον οδηγό μας.

Χρησιμοποιούμε το fcntl σχήμα κλειδωμάτων και όχι το flock, αφού το τελευταίο προσφέρει μόνο κλείδωμα ολόκληρων αρχείων και δεν υποστηρίζεται από το NFS. Επίσης αποφεύγουμε το mandatory locking, αφού έχουμε την δυνατότητα να επιβάλουμε σε όλες τις οντότητες να συμμορφωθούν με το μηχανισμό κλειδωμάτων και δεν αναμένουμε από άλλες εξωτερικές την προσπέλαση των αρχείων μας. Με αυτόν τον τρόπο γλιτώνουμε την σημαντική επιβάρυνση που εισάγεται όταν για κάθε κλήση συστήματος σχετική με Ε/Ε, ο πυρήνας πρέπει να ελέγχει εάν το αρχείο είναι κλειδωμένο.

Τώρα θα παρουσιάσουμε ποιες από τις παραπάνω οντότητες προσπελάζουν τα κοινά δεδομένα και με ποιο τρόπο το επιχειρούν, σκιαγραφώντας κάθε φορά την λύση κλειδώματος που λύνει το εκάστοτε πρόβλημα ταυτοχρονισμού.

info

Το αρχείο αυτό προσπελάσεται και αλλάζει μόνο στην περίπτωση δημιουργίας ενός νέου εικονικού δίσκου, για να προμηθεύσει την συνάρτηση δημιουργίας με το κατάλληλο uid του δίσκου. Προφανώς πρέπει να προστατέψουμε την προσπέλαση στο αρχείο με write lock, ώστε να μην έχουμε δίσκους με το ίδιο uid. Επειδή η δέσμευση του uid και η αύξησή του για το επόμενο διαθέσιμο, πρέπει να πραγματοποιείται μόνο όταν είμαστε σίγουροι ότι έχουν επιτύχει όλες οι άλλες απαιτούμενες ενέργειες για την δημιουργία του εικονικού δίσκου, το κλείδωμα του αρχείου αποτελεί το πρώτο πράγμα της συνάρτησης δημιουργίας και το ξεκλείδωμα είναι το τελευταίο. Έτσι εξασφαλίζουμε και την ατομικότητα της δημιουργίας ενός εικονικού δίσκου. Επίσης, σε περίπτωση που είναι η πρώτη φορά που δημιουργούμε εικονικό δίσκο, το σύστημα δημιουργεί ένα νέο αρχείο info και θεωρεί ως πρώτο uid το 1. Για να είμαστε βέβαιοι ότι ανάμεσα στον έλεγχο για το εάν υπάρχει το αρχείο και στην δημιουργία του και απόκτηση του πρώτου uid δεν θα εισέλθουν 2 διεργασίες δημιουργίας, χρησιμοποιούμε την σημαία `O_EXCL` στην κλήση συστήματος `open()` που δημιουργεί το αρχείο. Έτσι, εάν μετά τον έλεγχο προχωρήσουν δυο διεργασίες, μία από τις δύο θα αποτύχει.

Κόμβοι

Γενικώς κάθε οντότητα κλειδώνει τα entries του κόμβου που την ενδιαφέρουν, και χρησιμοποιεί read lock αν απλώς θέλει να τα διαβάσει, και write lock αν πρόκειται γράψει σε αυτά.

Μια οντότητα δημιουργίας στιγμιοτύπων κλειδώνει με write lock τους ριζικούς κόμβους και του εικονικού δίσκου και του στιγμιοτύπου, αφού και στους δύο υπάρχει εγγραφή ή ενημέρωση δεδομένων. Παρομοίως οι οντότητες αποκατάστασης ενός εικονικού δίσκου και δημιουργίας κλώνου, κλειδώνουν με read lock τον κόμβο-ρίζα του στιγμιοτύπου και με write lock τον κόμβο-ρίζα του πρωτεύοντος εικονικού δίσκου/κλώνου. Τα παραπάνω κλειδώματα των ριζικών κόμβων πραγματοποιούνται σε ολόκληρους τους κόμβους, δηλαδή σε όλα τα entries τους.

Η διεργασία qemu-system που εξυπηρετεί τις αιτήσεις του guest, για κάθε κόμβο που προσπελάζει κλειδώνει μόνο τα entries που εμπλέκονται, με read lock αν έχουμε αίτηση ανάγνωσης και με write αν έχουμε αίτηση εγγραφής. Η αναδρομική διάσχιση των κόμβων/τεμαχίων στα οποία δείχνουν τα entries που ενδιαφέρουν γίνεται με τα

κλειδώματα των entries κρατημένα. Η παραπάνω υλοποίηση κλειδώνει περιοχές κόμβων σταδιακά από το υψηλότερο προς το χαμηλότερο επίπεδο και τις ξεκλειδώνει πάλι σταδιακά κατά την ολοκλήρωση της επεξεργασίας των κόμβων, από το χαμηλότερο προς το ψηλότερο επίπεδο. Τα εμπλεκόμενα entries του ριζικού κόμβου δηλαδή, είναι κλειδωμένα καθ' όλη τη διάρκεια εξυπηρέτησης μιας αίτησης Ε/Ε του guest. Επειδή μια οντότητα λήψης ή αποκατάστασης στιγμιοτύπου χρειάζεται να κλειδώσει με write lock όλα τα entries του ριζικού κόμβου, συνεπάγεται ότι όσο όσο εξυπηρετείται μία αίτηση Ε/Ε δεν μπορεί να ληφθεί ένα στιγμιότυπο, αλλά και ότι όσο διαρκεί η λήψη στιγμιοτύπου δεν μπορεί να εξυπηρετηθεί καμία αίτηση Ε/Ε. Αυτός ο αμοιβαίος αποκλεισμός εξυπηρέτησης αιτήσεων και λήψης/αποκατάστασης στιγμιοτύπων, προκύπτει χάρη στην διατήρηση των κλειδωμάτων κατά την αναδρομική διάσχιση, και είναι απαραίτητος ώστε μετά από τη λήψη ενός στιγμιοτύπου, τα δεδομένα να είναι πράγματι immutable. Σε περίπτωση που δεν διατηρούμε τα locks και είναι δυνατόν να πεπλεχθεί η εξυπηρέτηση αιτήσεων με την λήψη στιγμιοτύπων, τότε είναι πιθανόν να παρουσιαστεί το εξής παθολογικό σενάριο:
Έστω ότι έχουμε μια αίτηση εγγραφής, και ο χρονοδρομολογητής του host αποφασίζει να διακόψει την διεργασία qemu-system αφού έχει επεξεργαστεί ο ριζικός κόμβος. Στη συνέχεια, η διεργασία λήψης ενός στιγμιοτύπου καταφέρνει να πάρει το κλείδωμα που χρειάζεται για όλο τον ριζικό κόμβο και να δημιουργήσει το στιγμιότυπο. Εάν τώρα εμφανιστεί μια αίτηση ανάγνωσης για τα δεδομένα του στιγμιοτύπου, τότε σε περίπτωση που δεν έχει ολοκληρωθεί η αρχική αίτηση εγγραφής, θα διαβαστούν παρωχημένα δεδομένα. Στην πραγματικότητα το πρόβλημα είναι ότι μετά από λίγο -μόλις ολοκληρωθεί η αρχική αίτηση εγγραφής-, τα δεδομένα αυτά θα αλλάξουν και έτσι θα υπάρχουν δύο εκδοχές για θεωρητικά immutable δεδομένα. Συνεπώς, η διατήρηση του write lock σε entries του ριζικού κόμβου κατά την διάρκεια εξυπηρέτησης μιας αίτησης, μας εγγυάται ότι για να επιτραπεί η λήψη ενός στιγμιοτύπου θα έχουν πρώτα ολοκληρωθεί όλες οι εκρεμμούσες αιτήσεις.

Οι αιτήσεις ανάγνωσης σε στιγμιότυπα, χρησιμοποιούν πάλι read locks στα entries που ενδιαφέρουν, καθώς παρόλο που οι κόμβοι είναι immutable, είναι πιθανόν να μεταβληθούν τα περιεχόμενά τους από μια διεργασία απαλοιφής διπλοτύπων. Η διεργασία αυτή προείπαμε ότι κάνει μια bottom-up διάσχιση του δέντρου, ενημερώνοντας τα nid με αποτυπώματα αντί για ονόματα αρχείων. Αφού βρεθεί το αποτύπωμα των κόμβων/τεμαχίων παιδιών ενός κόμβου, τότε λαμβάνεται ένα write lock σε ολό-

κληρο των κόμβο και ενημερώνονται τα entries του. Το κλείδωμα αυτό εξασφαλίζει ότι δεν θα υπάρξει απόπειρα ανάγνωσης των περιεχομένων του κόμβου κατά τη διαδικασία ενημέρωσης, αφού η διεργασία qemu-system δεν θα μπορέσει να λάβει read lock. Για τον ίδιο λόγο, κατά το COW ενός κόμβου πρέπει να λαμβάνεται read lock στον immutable κόμβο/τεμάχιο που πρόκειται να αντιγραφεί.

Ο αμοιβαίος αποκλεισμός αιτήσεων και λήψης/αποκατάστασης στιγμιοτύπου που προαναφέραμε θα μπορούσε να επιτευχθεί εναλλακτικά μόνο με ένα κεντρικό κλείδωμα στον κόμβο-ρίζα ή στην επικεφαλίδα συνταγής, καθ' όλη την διάρκεια μιας αίτησης και μιας λήψης/αποκατάστασης στιγμιοτύπου, χωρίς να λαμβάνονται κλειδώματα στα entries των κόμβων. Ο μηχανισμός που επιλέξαμε όμως δίνει την δυνατότητα αρχικά για χρησιμοποίηση του ίδιου κώδικα για αιτήσεις ανάγνωσης στιγμιοτύπων και πρωτευόντων δίσκων, μιας και μπορεί στις αιτήσεις ανάγνωσης πρωτευόντων δίσκων να μην απαιτείται read lock στους κόμβους, αλλά σε αυτές στα στιγμιότυπα είναι απαραίτητο λόγω της ύπαρξης του deduplication. Επίσης, με τον παρόντα μηχανισμό μπορούμε να εξασφαλίσουμε την ακεραιότητα στην περίπτωση που μια πιο επιθετική προσέγγιση επιλέξει να εφαρμόσει απαλοιφή διπλοτύπων σε κάποια περιοχή του πρωτεύοντος εικονικού δίσκου. Το μόνο που απαιτείται είναι η διεργασία απαλοιφής διπλοτύπων να κλειδώσει με ένα write lock τα entries που αναφέρονται σε υποδέντρα που θα υποστούν deduplication. Έτσι δεν θα επιτραπούν αιτήσεις εγραφής ή ανάγνωσης προς αυτά τα entries, και άρα προς αυτήν την περιοχή του δίσκου, μέχρι να ολοκληρωθεί η διαδικασία. Προφανώς επόμενες αιτήσεις εγγραφής σε deduplicated entries οφείλουν να προκαλούν COW.

Η διεργασία συγχρονισμού με απομακρυσμένο αντίγραφο συνήθως επιδρά σε στιγμιότυπα, λαμβάνοντας ένα read lock για κάθε κόμβο που πρόκειται να αποσταλεί στο δίκτυο, ώστε να μην έρθει σε σύγκρουση με τις ενημερώσεις της διεργασίας απαλοιφής διπλοτύπων. Με τον τρέχοντα σχεδιασμό, ο συγχρονισμός μπορεί να διενεργηθεί και για μια περιοχή του πρωτεύοντος εικονικού δίσκου με τον ίδιο ακριβώς τρόπο, αφού πιθανές αιτήσεις εγγραφής δεν θα μπορούν να πάρουν το write lock και άρα να μεταβάλουν τα δεδομένα, παρά μόνο πριν ή μετά από την αποστολή τους από την διεργασία συγχρονισμού.

Η οντότητα δημιουργίας του ριζικού κόμβου για έναν πρωτεύοντα εικονικό δίσκο δεν χρησιμοποιεί κάποιο κλείδωμα, αφού στη χειρότερη περίπτωση που έχουμε απόπειρα

για λειτουργία Ε/Ε στον δίσκο πριν από την δημιουργία του ριζικού κόμβου, η λειτουργία απλώς θα αποτύχει.

Επίσης, παρόλο που η deduplication διαδικασία εφαρμόζει bottom-up διάσχιση του δέντρου, ενώ οι άλλες εφαρμόζουν top-down διάσχιση, δεν θα έχουμε καταστάσεις αδιεξόδου (deadlocks). Αυτό γιατί η διεργασία απαλοιφής διπλοτύπων για να πάρει ένα write lock μπορεί να χρειαστεί να περιμένει μια αίτηση ανάγνωσης να απελευθερώσει το read lock, αλλά δεν κρατάει ταυτόχρονα κάποιο άλλο lock που εμποδίζει την αίτηση ανάγνωσης να προχωρήσει την καθοδική της διάσχιση και να ολοκληρωθεί.

Επικεφαλίδες συνταγής

Αρχικά πρέπει να διασφαλίσουμε ότι θα δημιουργηθεί μόνο μια επικεφαλίδα συνταγής για ένα εικονικό δίσκο, στιγμιότυπο ή κλώνο, και ότι εάν δυο οντότητες qemu-img προσπαθήσουν ταυτόχρονα να δημιουργήσουν την ίδια, τότε μια από τις δύο θα αποτύχει. Αυτό το επιτυγχάνουμε με την χρήση της σημαίας `O_EXCL` στην κλήση συστήματος `open()`, η οποία διασφαλίζει ατομική δημιουργία του αρχείου από την πρώτη διεργασία και αποτυχία στην εκ νέου δημιουργία που θα επιχειρήσει μια δεύτερη διεργασία. Εφεξής, υπάρχουν τρείς περιπτώσεις ασυνεπειών που μπορούν να προκύψουν.

Η πρώτη είναι να έχει δημιουργηθεί το αρχείο, και μια οντότητα να επιχειρήσει να προσπελάσει την επικεφαλίδα προτού ξεκινήσει η εγγραφή για την αρχικοποίηση της επικεφαλίδας ή πρωτού ολοκληρωθεί ολόκληρη. Η εγγραφή της επικεφαλίδας όμως γίνεται ατομικά, αφού το μέγεθος της επικεφαλίδας είναι 512bytes, και συνπεώς η εγγραφή αναγκαστικά θα διεκπερεωθεί σε μια λειτουργία Ε/Ε. Έτσι η λειτουργία ανάγνωσης από την δεύτερη οντότητα θα πραγματοποιηθεί είτε μετά τη λειτουργία εγγραφής, οπότε όλα τα δεδομένα της επικεφαλίδας θα είναι έγκυρα, είτε πριν, οπότε το αρχείο της επικεφαλίδας θα είναι κενό. Μπορούμε να αποτρέψουμε την πιθανότητα μια οντότητα να θεωρήσει έγκυρα τα δεδομένα που έχουν προκύψει από το διάβασμα μιας κενής επικεφαλίδας, εάν μετά από κάθε ανάγνωσή της, ελέγχουμε το magic number της επικεφαλίδας.

Η δεύτερη περίπτωση ασυνέπειας ανακύπτει εάν επιχειρηθεί η αποκατάσταση, η κλωνοποίηση ή το backup ενός στιγμιοτύπου ενόσω εφαρμόζεται η απαλοιφή διπλοτύπων. Αφού τελειώσει η διαδικασία απαλοιφής διπλοτύπων για όλο το δέντρο γίνεται η ενημέρωση της επικεφαλίδας συνταγής του στιγμιοτύπου με το νέο root nid, που δεν

είναι άλλο από το αποτύπωμα του κόμβου-ρίζας. Πάλι λόγω ατομικότητας της εγγραφής της επικεφαλίδας συνταγής, η διεργασία αποκατάστασης/κλωνοποίησης/backup, θα διαβαστεί είτε το παλιό root nid, είτε το καινούργιο. Ακόμα και αν διαβαστεί το παρωχημένο όνομα του κόμβου-ρίζας του στιγμιοτύπου αντί για το αποτύπωμά του, θεωρούμε ότι δεν υπάρχει πρόβλημα, αφού η διττή ύπαρξη αποτυπωμάτων και ονομάτων αρχείων είναι ένα γενικότερο πρόβλημα που πρέπει να επιλυθεί. Αυτό ανακύπτει γιατί μετά την απαλοιφή διπλοτύπων, το στιγμιότυπο έχει entries που περιέχουν αποτυπώματα ενώ ο πρωτεύων εικονικός δίσκος ή κάποιοι κλώνοι του έχουν entries με ονόματα αρχείων. Η τρίτη περίπτωση ασυνέπειας είναι τα online στιγμιότυπα.

Υποστήριξη online στιγμιοτύπων

Όπως προαναφέραμε, η δομή `BDRVDtreeState` είναι ορατή μόνο στο εσωτερικό μιας qemu-system ή qemu-img διεργασίας και κάθε νέο στιγμιότυπο μιας τέτοιας διεργασίας έχει ένα ξεχωριστό ανεξάρτητο αντίγραφο. Το πρόβλημα είναι ότι αυτή η δομή περιέχει πληροφορίες όπως το `next_snapshot_num`, οι οποίες πρέπει να είναι μοιραζόμενες. Ισοδύναμα, ο οδηγός μας αρχικοποιεί την δομή αυτή κατά την έναρξή του, με βάση πληροφορίες που υπάρχουν στην επικεφαλίδα συνταγής, αλλά μετά δεν ενημερώνει αυτές τις πληροφορίες. Για παράδειγμα, εάν μία διεργασία qemu-system εκκινήσει ένα VM που έχει προσαρτήσει έναν εικονικό δίσκο, τότε θα δημιουργηθεί ένα στιγμιότυπο `BDRVDtreeState` με βάση το `next_snapshot_num` που είναι γραμμένο κατά την στιγμή της εκκίνησης. Εάν όμως, κατά την διάρκεια λειτουργίας του VM, όπου η ίδια qemu-system με το ίδιο `BDRVDtreeState` συνεχίζουν να υπάρχουν, εκκινηθεί μια qemu-img διεργασία για την λήψη ενός στιγμιοτύπου του εικονικού δίσκου, τότε αφού ολοκληρωθεί το στιγμιότυπο, η επικεφαλίδα συνταγής θα έχει `next_snapshot_num` κατά ένα μεγαλύτερο από την πληροφορία που διατηρεί η δομή `BDRVDtreeState` του qemu-system. Για να καταπολεμήσουμε αυτήν την ασυνέπεια, είτε δεν θα επιτρέπουμε online δημιουργία στιγμιοτύπων με αυτόν τον τρόπο, είτε θα πρέπει να βρούμε ένα τρόπο να ενημερώνεται η πληροφορία `next_snapshot_num` της δομής `BDRVDtreeState`. Επιλέξαμε να υλοποιήσουμε την δεύτερη προσέγγιση, μέσω μιας πολιτικής ανάκρισης κάθε φορά που χρειαζόμαστε την εν λόγω πληροφορία. Για κάθε διάσχιση κόμβου που εμπίπτει σε ένα αίτημα εγγραφής του guest, διαβάζουμε τα περιεχόμενα του αρχείου της επικεφαλίδας συνταγής, ώστε σε περίπτωση που έχει αλλάξει το `next_snapshot_num` από μια διεργασία qemu-img, να ενημε-

ρώσουμε και την αντίστοιχη πληροφορία στην δομή `BDRVDtreeState`. Αυτό γίνεται μόνο για τις αιτήσεις εγγραφής στις οποίες θα χρειαστεί να κάνουμε δέσμευση νέου κόμβου/τεμαχίου, και άρα θα χρειαστούμε το `next_snapshot_num`, και γίνεται μόνο μια φορά σε κάθε αίτηση, την πρώτη φορά που αυτό θα απαιτηθεί. Το `next_snapshot_num` άλλωστε, είναι δυνατό να αλλάξει μόνο πριν ή μετά από κάποια αίτηση και όχι κατά την διάρκεια εξυπηρέτησής της από τον οδηγό μας, καθώς μια αίτηση εγγραφής κρατάει write lock σε κάποια entries του κόμβου-ρίζα μέχρι και την ολοκλήρωσή της, ενώ η λήψη στιγμιοτύπου απαιτεί write lock σε όλα τα entries της ρίζας. Επειδή υπάρχει αυτός ο αμοιβαίος αποκλεισμός ανάμεσα στην εξυπηρέτηση αιτήσεων και στην λήψη στιγμιοτύπου, δεν χρειαζόμαστε locks στην επικεφαλίδα συνταγής κατά την ανάγνωση και ενημέρωσή της. Αρκεί η λειτουργία ενημέρωσης κατά την λήψη του στιγμιοτύπου να πραγματοποιηθεί όσο είναι κρατημένο το write lock όλων των entries του ριζικού κόμβου. Έτσι θα διασφαλιστεί η ατομικότητα δημιουργίας του στιγμιοτύπου και της ενημέρωσης του `next_snapshot_num`, και θα είμαστε σίγουροι ότι μια αίτηση θα διαβάσει την πιο πρόσφατη τιμή του.

Η επίδραση στην απόδοση είναι μικρή αν αναλογιστούμε ότι έχουμε μόνο μία επιπρόσθετη λειτουργία Ε/Ε ανά αίτηση, η οποία πρακτικά πιθανότατα θα αποφύγει την προσπέλαση στο δίσκο. Μετά την πρώτη φορά που διαβάζουμε την επικεφαλίδα συνταγής, τα περιεχόμενα της, που είναι και μόλις 512bytes, θα βρίσκονται σε κάποια σελίδα της page cache και έτσι η ανάγνωση θα είναι πολύ γρήγορη. Ακόμα και αν έχουν φύγει κάποια στιγμή μετά την εκκίνηση του VM, η λήψη στιγμιοτύπου θα έχει απαιτήσει ενημέρωση της επικεφαλίδας συνταγής και άρα θα τα έχει επαναφέρει στην host page cache.

Σε έναν πιο ολοκληρωμένο σχεδιασμό, θα ήταν ιδανικό να παραμετροποιήσουμε το πρότυπο του QEMU για λήψη online στιγμιοτύπων, έτσι ώστε όταν ο εικονικός δίσκος είναι της μορφής μας, να καλεί την συνάρτηση λήψης στιγμιοτύπου, αντί για να δημιουργεί ένα COW image. Καθ' αυτόν τον τρόπο, δεν θα χρειαζόταν να ελέγχουμε κάθε φορά εάν έχει ληφθεί ένα online στιγμιότυπο και άρα πρέπει να ενημερωθεί η δομή `BDRVDtreeState`, αλλά η λήψη στιγμιοτύπου θα πυροδοτούσε μια συνάρτηση `bdrv_reopen()`, η οποία θα ενημέρωνε την δομή.

Προστασία από πεπλεγμένες αιτήσεις

Αν και υπάρχει μόνο μία διεργασία qemu-system που εξυπηρετεί αιτήσεις για έναν εικονικό δίσκο, είναι πιθανό να υπάρχουν ταυτόχρονα πολλές εκρεμμείς αιτήσεις που διεκδικούν πρόσβαση στους κόμβους και τα τεμάχια. Δεν μας πειράζει εάν αιτήσεις ανάγνωσης πεπλεχθούν με αιτήσεις εγγραφής, καθώς η μόνη εγγύηση ατομικότητας που καλείται να παρέχει ο εικονικός δίσκος μας είναι της αδιάσπαστης ανάγνωσης ή εγγραφής ενός τομέα των 512 bytes, πράγμα που ούτως ή άλλως μας εγγυάται ο φυσικός δίσκος του host. Αυτό που καλούμαστε να διαφυλάξουμε είναι η συνέπεια των μεταδεδομένων μας, δηλαδή να αποτρέψουμε περιπτώσεις που η παρεμβολή αιτήσεων με άλλες θα αλλοιώσει κάποια δεδομένα των κόμβων. Κάτι τέτοιο είναι δυνατό μόνο από πεπλεγμένες αιτήσεις εγγραφής που αναγκάζονται να γράφουν στους κόμβους, δηλαδή από αιτήσεις που προκαλούν COW. Συγκεκριμένα, εάν δύο αιτήσεις εγγραφής προσπελάσουν ένα entry που είναι μόνο για ανάγνωση, τότε και οι δύο θα επιχειρήσουν να κάνουν COW και να ενημερώσουν το όνομα και τα δικαιώματα του entry. Παρόλο που η ενημέρωση και των δύο θα καταλήξει στο ίδιο αποτέλεσμα για τον γονεϊκό κόμβο, τα περιεχόμενα του κόμβου/τεμαχίου-παιδί που θα γίνει COW, θα είναι διαφορετικά. Αυτό προφανώς δεν θα συμβεί στην αρχική αντιγραφή από τον immutable κόμβο/τεμάχιο, αλλά από την μετέπειτα διάσχιση του νεοσύστατου. Αν λοιπόν και οι δύο αιτήσεις επιχειρήσουν να δημιουργήσουν τον κόμβο/τεμάχιο-παιδί, και μετά την πρώτη δημιουργία προλάβει να γίνει κάποια αλλαγή των περιεχομένων του κόμβου/τεμαχίου-παιδί, η δεύτερη δημιουργία είτε θα αποτύχει είτε θα ξανακάνει COW και θα γράψει πάνω από τα περιεχόμενα της πρώτης. Έτσι θα έχουμε ένα race condition ανάλογα με την σειρά που οι αιτησεις θα δημιουργήσουν τον νέο κόμβο/τεμάχιο. Για να αποτρέψουμε την περίπτωση που δυο αιτήσεις προσπαθήσουν να κάνουν COW τον ίδιο κόμβο/τεμάχιο, υπάρχουν διάφορες προσεγγίσεις.

Σύμφωνα με την πρώτη, οι αναγνώσεις και οι εγγραφές θα πραγματοποιούνται σε έναν ιδιωτικό χώρο μνήμης του χώρου χρήστη (cache του οδηγού), όπου θα φορτώνονται όλοι οι κόμβοι. Σε αυτήν την ιδιωτική cache λοιπόν, η οποία θα αντανακλά τις on-disk δομές του σχεδιασμού μας στην μνήμη χώρου χρήστη, θα έχουν πρόσβαση όλα τα νήματα των διαφόρων αιτημάτων Ε/Ε, και πάνω σε αυτές τις δομές στην μνήμη θα χρησιμοποιήσουμε αποκλειστικά κλειδώματα (π.χ. posix mutexes) για να αποτρέψουμε δυο νήματα από το να δημιουργήσουν τον ίδιο κόμβο/τεμάχιο δύο φορές. Το πρώτο νήμα που θα προλάβει να πάρει το κλείδωμα, θα δημιουργήσει τον

κόμβο/τεμάχιο και θα κρατήσει το κλείδωμα μέχρις ότου να ενημερώσει τα δικαιώματα του entry στην ιδιωτική cache. Από εκεί και στο εξής, το δεύτερο νήμα που θα προσπαθήσει να δημιουργήσει τον κόμβο/τεμάχιο, θα ξαναελέγξει τα δικαιώματα, και βλέποντας ότι αυτά δεν είναι πλέον μόνο για ανάγνωση, θα παραβλέψει την COW λειτουργία. Κάθε entry θα έχει το δικό του κλείδωμα προσθέτοντας στην δομή κόμβος-στην-μνήμη, πέρα από τα πεδία nid και permissions του κόμβου-στο-δίσκο, και ένα πεδίο lock.

Σύμφωνα με την δεύτερη προσέγγιση, η δεύτερη αίτηση που προσπαθεί να δημιουργήσει το αρχείο, μόλις αντιληφθεί την ύπαρξή του αποτυγχάνει, και επανεκκινείται η διαδικασία εξυπηρέτησής της από τον οδηγό. Θεωρούμε ότι μέχρι να ξαναφτάσει στο σημείο ανάγνωσης του κόμβου θα έχει ολοκληρωθεί η δέσμευση του νέου αρχείου και το COW. Εφαρμόζουμε δηλαδή ένα επιθετικό και πιο χρονοβόρο ίσως σχήμα συγχρονισμού, με την λογική ότι οι πεπλεγμένες αιτήσεις εγγραφής δεν θα είναι συχνό φαινόμενο στον εικονικό μας δίσκο. Εναλλακτικά, θα μπορούσαμε να σειριοποιήσουμε τις δημιουργίες κόμβων/τεμαχίων προσθέτοντάς τις σε μια ουρά, λογική που υιοθετεί και το qcow2 κατά την δέσμευση των clusters του. Κάθε φορά που μία αίτηση, και άρα ένα νήμα, ήθελε να δημιουργήσει μια αίτηση, θα ήλεγχε εάν αυτή η αίτηση υπήρχε στην ουρά. Εάν ναι θα περίμενε σε μια ουρά αναμονής την ολοκλήρωσή της και εάν όχι θα την πρόσθετε με προστατευμένο τοπο στην ουρά και θα προχωρούσε με την δημιουργία. Θεωρούμε όμως ότι θα υπήρχε συμφόρηση και αυξημένη καθυστέρηση οπότε επιλέγουμε να υιοθετήσουμε την δεύτερη προσέγγιση.

Ο παραπάνω μηχανισμός κλειδωμάτων που υιοθετήθηκε, καταλήγει σε μια μορφή εικονικού δίσκου που είναι ανθεκτική σε αστοχίες (crash-consistent). Αυτό συμβαίνει γιατί τα μεταδεδομένα παραμένουν συνεπή μετά από τη βίαιη διακοπή κάποιας λειτουργίας. Οι κόμβοι ενημερώνονται αφού έχει γίνει η ολοκλήρωση του COW για τα παιδιά τους, οπότε δεν υπάρχει περίπτωση να διακοπεί μια αίτηση και να καταλήξουμε σε μια ασυνεπή δομή.

4.5 Περιγραφή βασικών λειτουργιών

Τώρα που ολοκληρώσαμε την περιγραφή όλων των λεπτομερειών της υλοποίησης, προχωράμε σε μια ολοκληρωμένη περιγραφή των βασικών λειτουργιών του οδηγού.

Αίτηση ανάγνωσης ή εγγραφής από τον guest

Για κάθε κόμβο που προσπελάσεται παρέχονται οι σχετικοί και όχι οι απόλυτοι αριθμοί τομέων που μας ενδιαφέρουν, δηλαδή η αρίθμηση των τομέων δεν αναφέρεται σε ολόκληρο τον εικονικό δίσκο αλλά στο υποδέντρο που αυτός ο κόμβος ορίζει. Αρχικά προσπελάσεται ο κόμβος-ρίζα. Για αυτόν και για κάθε άλλο κόμβο ακολουθείται η εξής διαδικασία:

1. Ανοίγει το αρχείο του κόμβου με μια κλήση συστήματος `open()`.
2. Με βάση τον σχετικό αριθμό τομέων, προκύπτουν υπολογιστικά τα entries του κόμβου που πρέπει να διαβαστούν για την περαιτέρω πλοήγηση στους κόμβους-παιδιά ή τεμάχια-παιδιά.
3. Λαμβάνουμε read lock για αίτηση ανάγνωσης ή write lock για αίτηση εγγραφής μόνο για την περιοχή bytes που αναφέρεται στα entries που μας ενδιαφέρουν.
4. Για κάθε entry ξεχωριστά:
    (a) Υπολογίζονται οι νέοι σχετικοί τομείς που θα πρέπει να προσπελαστούν στον κόμβο-παιδί ή τεμάχιο-παιδί και αποθηκεύονται σε ένα πίνακα.
    (b) Αν έχουμε μια αίτηση εγγραφής και κάποιο entry έχει δικαιώματα 0, δηλαδή είτε δεν έχει δεσμευτεί ποτέ, είτε απαιτεί COW:
        i. Εάν είναι η πρώτη φορά που γίνεται δέσμευση νέου κόμβου/τεμαχίου για αυτήν την αίτηση, διαβάζουμε το `next_snapshot_num` από την επικεφαλίδα συνταγής, για να δούμε εάν δημιουργήθηκε κάποιο στιγμιότυπο και άρα αυξήθηκε η τιμή του.
        ii. Λαμβάνεται η απόφαση για το εάν θα πρέπει να γίνει COW, δηλαδή πέρα από την δέσμευση του κόμβου/τεμαχίου να υπάρξει αντιγραφή των δεδομένων από κάποιον μοιραζόμενο κόμβο ή τεμάχιο. COW δεν γίνεται μόνο εάν έχουμε προσπέλαση κόμβου/τεμαχίου για πρώτη φορά (NID_-TYPE_NULL) ή εάν το τεμάχιο πρέπει να γραφτεί εξ' ολοκλήρου.
        iii. Δημιουργείται ο νέος κόμβος/τεμάχιο με μια κλήση συστήματος `open()`. Εάν χρειάζεται COW, γίνεται αντιγραφή όλων των δεδομένων από τον παλιό κόμβο/τεμάχιο, και εάν έχουμε κόμβο, μετατρέπουμε όλα τα δικαιώματα σε μόνο για ανάγνωση. Εάν δεν χρειάζεται COW, δεσμεύονται συνεχόμενα μπλοκ του συστήματος αρχείων με την `posix_fallocate()`,

και εάν έχουμε κόμβο, γράφουμε επιπλέον και την επικεφαλίδα του. Εάν κατά την κλήση της open διαπιστωθεί ότι έχει ήδη δημιουργηθεί ο κόμβος/τεμάχιο, τότε η αίτηση επανεκκινείται.

5. Αφού ολοκληρωθεί η προσπέλαση όλων των entries, και εάν έχει υπάρξει έστω και μία δημιουργία νέου κόμβου/τεμαχίου, ενημερώνονται τα ονόματα και τα δικαιώματα των entries με τα νέα. Η εγγραφή γίνεται με μία λειτουργία Ε/Ε.
6. Εφαρμόζεται αναδρομικά η εν λόγω διαδικασία για όλους τους κόμβους/τεμάχια στους οποίους δείχναν τα entries ή που μόλις δημιουργήθηκαν.
7. Ξεκλειδώνουμε όλα τα read και write locks.

Εάν φτάσουμε σε κάποιο τεμάχιο, απλώς μεταφέρουμε δεδομένα από τον buffer στο αρχείο τεμαχίου ή αντίστροφα, με βάση πάντα τον αριθμό των τομέων που προέκυψαν από το ανώτερο επίπεδο.

Δημιουργία στιγμιοτύπου

1. Γίνεται έλεγχος για το εάν το όνομα του νέου στιγμιοτύπου υπάρχει ήδη, διαβάζοντας τον κατάλογο με τις επικεφαλίδες συνταγής.
2. Δημιουργείται με `open()` μια επικεφαλίδα συνταγής για το στιγμιότυπο, και γράφονται οι δομές `RecipeHeader`, `SnapshotHeader`, το όνομα του δίσκου και το νέο root nid.
3. Δημιουργείται με `open()` ένα αρχείο κόμβου-ρίζας με βάση το νέο root nid.
4. Λαμβάνουμε write lock και στους δύο κόμβους-ρίζες, σε όλα τα entries τους.
5. Αντιγράφουμε τα δεδομένα από τον κόμβο-ρίζα του πρωτεύοντος δίσκου στον κόμβο-ρίζα του στιγμιοτύπου, μετατρέποντας τα entries και των δυο κόμβων-ρίζας σε μόνο για ανάγνωση. Για διατήρηση της συνέπειας πρώτα γίνεται η εγγραφή στον κόμβο-ρίζα του στιγμιοτύπου και μετά η ενημέρωση των δικαιωμάτων στον κόμβο-ρίζα του πρωτεύοντος εικονικού δίσκου.
6. Αυξάνουμε το `next_snapshot_num` και ενημερώνουμε την δομή `RecipeHeader`.
7. Απελευθερώνεται το write lock στους κόμβους-ρίζες.

Δημιουργία κλώνου

1. Γίνεται έλεγχος για το εάν το όνομα του νέου κλώνου υπάρχει ήδη ως εικονικός δίσκος, διαβάζοντας τον κατάλογο με τις επικεφαλίδες συνταγής.
2. Ανοίγουμε την επικεφαλίδα συνταγής του στιγμιοτύπου από το οποίο θα προκύψει ο κλώνος, και γίνεται έλεγχος ότι όντως πρόκειται για στιγμιότυπο και όχι πρωτεύοντα δίσκο.
3. Κλειδώνουμε με write lock το αρχείο info και λαμβάνεται το επόμενο διαθέσιμο uid για τον κλώνο.
4. Δημιουργείται με `open()` μια επικεφαλίδα συνταγής για τον κλώνο, και γράφεται η δομή `RecipeHeader`, το όνομα του κλώνου και το νέο root nid. Πρόκειται για μια επικεφαλίδα συνταγής πανομοιότυπη με αυτό ενός πρωτεύοντος εικονικού δίσκου.
5. Δημιουργείται με `open()` ένα αρχείο κόμβου-ρίζας για τον κλώνο, με βάση το νέο root nid.
6. Αντιγράφουμε τα δεδομένα από τον κόμβο-ρίζα του στιγμιοτύπου στον κόμβο-ρίζα του κλώνου, μετατρέποντας τα entries μόνο του κόμβου-ρίζας του κλώνου, σε μόνο για ανάγνωση. Λαμβάνεται read lock στον κόμβο-ρίζα του στιγμιοτύπου και write lock στον κόμβο-ρίζα του κλώνου, καθ’ όλη την διάρκεια της αντιγραφής.
7. Ενημερώνουμε το αρχείο info, αυξάνοντας το επόμενο διαθέσιμο αφήνουμε το write lock.

Αποκατάσταση εικονικού δίσκου από στιγμιότυπο

Ανοίγουμε την επικεφαλίδα συνταγής του στιγμιοτύπου και τους κόμβους-ρίζα του στιγμιοτύπου και του πρωτεύοντος δίσκου και αντιγράφουμε τα δεδομένα από τον κόμβο-ρίζα του στιγμιοτύπου σε αυτόν του πρωτεύοντος δίσκου. Δεν χρειάζεται μετατροπή των δικαιωμάτων, αφού στον κόμβο-ρίζα του στιγμιότυπου αυτά έχουν ήδη τεθεί κατά τη λήψη του, σε μόνο για ανάγνωση. Για κάθε εγγραφή στον πρωτεύοντα δίσκο μετά την αποκατάσταση θα χρειάζεται COW, το οποίο όμως θα οδηγεί στην δημιουργία hot κόμβων/τεμαχίων που θα έχουν το ίδιο όνομα όπως και οι hot κόμβοι/τεμάχια που υπήρχαν πριν από την αποκατάσταση. Συνεπώς, η διαφορά με το COW μετά από ένα στιγμιότυπο είναι ότι τα αρχεία δεν θα δημιουργούνται στο σύστημα αρχείων, αλλά καθώς θα προϋπάρχουν, απλώς θα γίνονται overwrite με τα δεδομένα του immutable κόμβου/τεμαχίου.

5 Πειραματική Αξιολόγηση

5.1 Ρύθμιση πλατφόρμας μετρήσεων

Η πειραματική αξιολόγηση χωρίζεται σε τρία κομμάτια:

- Micro benchmarks: εδώ έχουμε σαφώς καθορισμένα φορτία (workload) που αναφέρονται σε συγκεκριμένες περιοχές του δίσκου, με συγκεκριμένα μοτίβα προσπελάσεων και συγκεκριμένες λειτουργίες Ε/Ε. Χρησιμεύουν για να μετρήσουμε την απόδοση κάθε παραμετροποίησης ανάλογα με την λειτουργία Ε/Ε και να καταλήξουμε σε κάποια σύνολα παραμέτρων που επιτυγχάνουν την ισορροπία που επιθυμούμε ανάμεσα στην απόδοση και στην εξοικονόμηση χώρου. Μας επιτρέπουν επίσης να απομονώσουμε την συμπεριφορά του συστήματός μας σε στοιχειώδεις λειτουργίες Ε/Ε και να μετρήσουμε τις επιδράσεις του φορτίου σε πολύ συγκεκριμένα κομμάτια του συστήματός μας.
- Μεταδεδομένα: εδώ αναλύουμε θεωρητικά και πρακτικά την επίδραση των διαφόρων παραμέτρων στην ποσότητα μεταδεδομένων που απαιτούνται.
- Macro benchmarks: εδώ έχουμε φορτία που προσομοιώνουν κάποιο πραγματικό φορτίο που μπορεί να υπάρξει σε ένα πραγματικό σύστημα και αφορούν όχι μεμονωμένες λειτουργίες Ε/Ε, αλλά συμπεριφορικά μοτίβα Ε/Ε σε ένα ανώτερο επίπεδο εφαρμογών. Αυτά τα benchmarks μας προσφέρουν μια μακροσκοπική εποπτεία της απόδοσης του συστήματός μας, για έναν αριθμό διαφορετικών προτύπων φορτίου που συναντώνται σε πραγματικά συστήματα στην παραγωγή. Μας επιτρέπουν να δούμε πώς το σύστημά μας αποκρίνεται σε σενάρια πραγματικής χρήσης.

Οι παράμετροι που θα μεταβάλλουμε στις διάφορες δοκιμές με σκοπό να μελετήσουμε την απόδοση του συστήματός μας είναι οι εξής:

- κατάσταση του δίσκου / φάση δεδομένων δίσκου (αδέσμευτα, δεσμευμένα, COW): περιγράφει την κατάσταση των δεδομένων της περιοχή του δίσκου που εξετάζουμε, ως προς το εάν η περιοχή του δίσκου είναι άδεια, δηλαδή είναι η πρώτη προσπέλαση που γίνεται σε αυτά και που θα επιφέρει την δημιουργία κόμβων και τεμαχίων (αδέσμευτα ή clean), μια κανονική προσπέλαση μετά την δημιουργία των κατάλληλων κόμβων και τεμαχίων για αυτά (δεσμευμένα ή allocated) ή μια προσπέλαση που θα επιφέρει COW σε κόμβους και τεμάχια, αφού τα δεδομένα αυτά θα έχουν γίνει immutable από την λήψη κάποιου στιγμιοτύπου (COW).
- μέγεθος τεμαχίου (4KB-4MB): περιγράφει το μέγεθος του τεμαχίου και ανάλογα με το πείραμα θα χρησιμοποιήσουμε διαφορετικό υποδιάστημα της ενδεικτικής περιοχής. Παραπάνω από 4MB θεωρούμε ότι είναι ασύμβατο εάν θέλουμε να επιτύχουμε απαλοιφή διπλοτύπων.
- ύψος δέντρου / αριθμός επιπέδων δέντρου συνταγής (1-10): περιγράφει πόσα επίπεδα θα έχει το δέντρο των κόμβων και αρχίζει από 1, όπου υπάρχει μόνο η ρίζα και έχουμε ουσιαστικά την επίπεδη δομή που περιγράψαμε στην 3.5, μέχρι 10. Για ύψος μεγαλύτερο του 10 θεωρούμε ότι εισάγουμε αδικαιολόγητα λειτουργίες Ε/Ε, χωρίς να αποκομίζουμε τα ανάλογα οφέλη.

Το μέγεθος του δίσκου είναι μια παράμετρος που δεν επιδρά άμεσα στην απόδοση αλλά σε συνδυασμό με το ύψος του δέντρου, καθορίζουν το μέγεθος και το πλήθος των κόμβων του δέντρου και άρα επιφέρει έμμεσες μεταβολές. Για λόγους οικονομίας χρόνου εκτελέσαμε όλα τα πειράματά μας με δίσκο 10GB, θεωρώντας τον ως αντιπροσωπευτικό. Το macro-benchmark postmark που εκτελέστηκε και με δίσκους μεγέθους 20GB και 30GB, δεν παρουσίασε ουσιαστικές διαφοροποιήσεις.

Τα τεχνικά χαρακτηριστικά των μηχανημάτων που χρησιμοποιήσαμε για τις μετρήσεις είναι τα εξής:
Guest:
CPU: 2 of host cores, L2 cache 4K
memory: 1GB
OS: Linux 3.2.0 amd64, Debian 7.6 64bit wheezy
πρόσβαση μέσω τερματικού (no graphical mode)

Hypervisor: Qemu 2.5.0 with KVM kernel module in Linux enabled, full virtualization

Host:
hard disk: ST9640320AS, SATA, 5400 rpm, 8MB cache, 640GB
CPU: micro-benchmarks: {Intel® Core™ i5 CPU M 450 @ 2.40GHz × 4 , x86_64, L1 32KB, L2 256KB, L3 3MB}, macro-benchmarks: {Intel® Core™ i7-6500U @ 2.50GHz x 4, x86_64, L1 32KB, L2 256KB, L3 4MB }
memory: {micro benchmarks: 5,7GB, DDR3, 1333MHz}, macro benchmarls: {7,5GB, DDR3, 1600MHz}
OS: Linux 3.16.0 generic, 14.04 LTS Ubuntu 64bit trusty (on 95GB partition with extra swap partition)

Η διαφοροποίηση στον επεξεργαστή και την κύρια μνήμη ανάμεσα στα micro και στα macro benchmarks δεν οφείλεται σε κάποια σχεδιαστική επιλογή, αλλά στο ατυχές γεγονός βλάβης στον επεξεργαστή μας.

Η ιεραρχική δομή των καταλόγων που χρησιμοποιεί το σύστημά μας, όπως παρουσιάστηκε στην ενότητα 4.1, τοποθετήθηκε σε ένα ξεχωριστό φυσικό δίσκο από αυτόν που βρίσκεται εγκατεστημένο το host λειτουργικό σύστημα, έτσι ώστε να απαλλαγούμε από τον θόρυβο που θα εισήγαγαν λειτουργίες Ε/Ε άσχετες με το σύστημά μας.

Ο εικονικός δίσκος εισάγεται στον guest με την επιλογή writeback για το cache mode, έτσι ώστε να εκμεταλλευόμαστε την host page cache αλλά να μην έχουμε μετά από κάθε αίτηση εγγραφής την υποχρέωση να συγχρονίζουμε τα δεδομένα στον δίσκο. Σημειώνουμε ότι μόνο αυτή η περίπτωση cache mode μπορεί να παράγει ένα λειτουργικό εικονικό δίσκο για την υλοποίησή μας, καθώς στην writethrough περίπτωση εγγραφών, ο συντριπτικός αριθμός των ξεχωριστών αιτήσεων συγχρονισμού για κάθε αρχείο, οδηγεί σε υπέρογκες καθυστερήσεις ακόμα και για μικρά μεγέθη αιτήσεων. Ο συγχρονισμός πραγματοποιείται μόνο μετά από ρητή απαίτηση του guest, όταν αυτός θέλει να διασφαλίσει ότι τα δεδομένα είναι αποθηκευμένα με μόνιμο τρόπο. Επίσης, ως διεπαφή του εικονικού δίσκου επιλέγεται η VirtIO με τον παραεικονικοποιημένο οδηγό της, καθώς έτσι επιταχύνεται η επικοινωνία guest και host για μεταφορά δεδομένων, χωρίς να επηρεάζονται όμως οι συσχετισμοί απόδοσης ανάμεσα στις διαφορετικές παραμετροποιήσεις.

Πριν από κάθε μέτρηση, και εκτός αν διατυπώνεται το αντίθετο, αδειάζουμε την page cache, την inode cache και την dentry cache, τόσο στον host όσο και στον guest. Στο

ext4 σύστημα έχουμε ενεργοποιήσει τις επιλογές noatime και nodriatime για αποφυγή αχρείαστων εγγραφών, και την επιλογή directory idndexing για ταχύτερη εύρεση των αρχείων στους καταλόγους.

Το journaling σύστημά μας είναι writeback. Το journal δεν μας εγγυάται ότι δεν θα χαθούν δεδομένα αλλά μας εγγυάται ότι δεν θα έρθει σε ασυνεπή κατάσταση το σύστημα αρχείων. Προβαίνουμε σε αυτή την επιλογή, ώστε σε περίπτωση που σε κάποια αστοχία το σύστημα αρχείων του host έρθει σε ασυνεπή κατάσταση, να μην χρειαστεί να τρέξουμε `fsck` για επιδιόρθωση. Δεν υπάρχει λόγος να βάλουμε ordered αντί για writeback mode. Στην ordered αποκλείεται να έχουμε file corruption, δηλαδή αρχείο που να έχει λάθος δεδομένα. Στο writeback υπάρχει περίπτωση ένα αρχείο να έχει λάθος δεδομένα και άρα για τον guest αυτό μεταφράζεται στο να έχουν κάποιοι τομείς του εικονικού δίσκου λάθος δεδομένα. Αυτό μας πειράζει μόνο εάν το σύστημα αρχείων του guest νομίζει ότι αυτά τα δεδομένα είναι σωστά. Όμως το σύστημα αρχείων του guest είναι υπεύθυνο ώστε αυτό να διαχειρίζεται την εικόνα του για την συνέπεια των δεδομένων και όπως και ένας πραγματικός δίσκος, ο φυσικός δίσκος μας δεν οφείλει να παρέχει οποιαδήποτε εγγύηση για την συνέπεια των δεδομένων σε υψηλότερο επίπεδο.

Δεν εκτελέστηκαν πειράματα για μέτρηση της αποτελεσματικότητας της απαλοιφής διπλοτύπων γιατί πέρα από την πολυπλοκότητα κατασκευής ενός συστήματος απαλοιφής διπλοτύπων, είναι πρακτικά δύσκολη η εξεύρεση ή δημιουργία φορτίων τα οποία είναι και αντιπροσωπευτικά και αρκούντως μεγάλα, ώστε να προσφέρουν αξιόπιστα αποτελέσματα για το deduplication ratio. Επίσης υπάρχουν ήδη δεδομένα άλλων ερευνών, για την αποτελεσματικότητα του deduplication σε σχέση με το μέγεθος του τεμαχίου, και τα οποία μπορούμε να χρησιμοποιήσουμε ως οδηγό.

5.2 Micro-Benchmarks

Για τα micro-benchmarks χρησιμοποιούμε το fio (flexible IO tester) [Agc], το οποίο είναι μια προηγμένη και πλήρως παραμετροποιήσιμη γεννήτρια φορτίων I/O με πλούσια έξοδο.

Όταν κάνουμε τις μετρήσεις των micro-benchmarks δεν χρησιμοποιούμε σύστημα αρχείων στον guest αλλά γράφουμε κατευθείαν στην συσκευή μπλοκ (block device) του guest που αντιστοιχεί στον εικονικό δίσκο, χρησιμοποιώντας direct I/O. Αυτό γιατί θέλουμε να μετρήσουμε την απόδοση του οδηγού μας απομονώνοντας τα υπόλοιπα στοιχεία που μπορεί να την επηρεάζουν. Τόσο στα σειριακά όσο και στα τυχαία μοτίβα, ο guest εκδίδει αιτήσεις μεγέθους 4KB. Χρησιμοποιήθηκε η libaio με βάθος ουράς 4. Η ολοκλήρωση μιας αίτησης Ε/Ε σηματοδοτείται από την ολοκλήρωση της bdrv_co_write/bdrv_co_read συνάρτησης του οδηγού μας. Τέλος, αφού εκδωθούν όλες οι λειτουργίες Ε/Ε, χρησιμοποιούμε την επιλογή end_fsync η οποία συνεπάγεται μια κλήση `fsync()` πάνω στη συσκευή μπλοκ και άρα εγγυάται ότι όλα τα δεδομένα θα γραφτούν στον πραγματικό δίσκο, χάρη στην αλυσίδα κλήσεων που περιγράψαμε στην ενότητα 4.3.

Πριν φτάσουμε στις μετρήσεις, αναλύουμε τις πρωτογενείς αιτίες που θεωρούμε ότι κρίνουν την απόδοση. Αυτές ανάγονται στον αριθμό των αρχείων τα οποία προσπελάζουμε και στην ποσότητα των δεδομένων που συνολικά γράφουμε/διαβάζουμε. Αυτές είναι δυο διαφορετικές πτυχές, με την αύξηση κάθε μιας να προκαλεί και μείωση της απόδοσης. Η αύξηση του αριθμού των αρχείων που γράφουμε/διαβάζουμε οδηγεί σε περισσότερες ή μεγαλύτερες αιτήσεις Ε/Ε για την ενημέρωση του inode table και των bitmaps, σε προσπέλαση περισσότερων directory data blocks για εύρεση των λόγω αρχείων στον κατάλογο, σε περισσότερες κλήσεις συστήματος, ενώ για αιτήσεις εγγραφής αυξάνεται και ο χρόνος των εγγραφών στο journal, αφού αυξάνεται ο αριθμός των συναλλαγών. Η αύξηση των δεδομένων και άρα των data blocks που πρέπει να γράψουμε, οδηγεί λογικά σε αύξηση του χρόνου που χρειάζεται ο δίσκος για να φέρει εις πέρας την εγγραφή των περισσότερων δεδομένων. Προφανώς σημαντικό είναι και το εάν οι αιτήσεις Ε/Ε για αυτά τα μπλοκ είναι σειριακές (sequential) ή τυχαίες (random).

Όπως αναλύσαμε και στην ενότητα 3.8, κυρίαρχη παράμετρος στην απόδοση είναι ο

αριθμός των αρχείων (κόμβων και τεμαχίων) που προσπελάσονται και όχι η ποσότητα των δεδομένων, η οποία είναι για κάθε λειτουργία Ε/Ε σχετικά μικρή. Σε αυτήν την ενότητα θα επεκτείνουμε τις θεωρητικές παρατηρήσεις της 3.8, αφού τώρα στις βασικές λειτουργίες Ε/Ε προστίθονται λειτουργίες εξαιτίας της διαχείρισης των οντοτήτων από το σύστημα αρχείων. Παράλληλα, διαπιστώνουμε την συμπεριφορά του εικονικού δίσκου με την χρήση της host page cache και είμαστε σε θέση να εκτιμήσουμε τους πραγματικούς συσχετισμούς ανάμεσα στις διάφορες περιπτώσης του δίσκου και στα διαφορετικά μοτίβα προσπέλασης.

Επειδή το δέντρο παραμέτρων που δημιουργείται από τον συνδυασμό όλων των δυνατών τιμών των τεσσάρων παραμέτρων, είναι εξαιρετικά μεγάλο, θα εφαρμόσουμε μια τακτική σταδιακού κλαδέματος, όπου εξετάζοντας μια παράμετρο κάθε φορά, θα εξάγουμε συμπεράσματα για τις τιμές της που μοιάζουν ασύμφορες ή ανούσιες. Έτσι θα τις απαλείφουμε από τον επόμενο γύρο μετρήσεων που θα εκτελούμε, ώστε να αντιμετωπίσουμε την εκθετική πολυπλοκότητα του δέντρου.

Αρχίζουμε τώρα να παρουσιάζουμε τις διάφορες μετρήσεις που εκτελέσαμε. Μετράμε την απόδοση τόσο με βάση το bandwidth που παρουσίασε ο δίσκος, δηλαδή τις αιτήσεις που μπορούσε να διεκπεραιώσει στην μονάδα του χρόνου, όσο και με βάση το latency, δηλαδή την καθυστέρηση εκπλήρωσης μιας αίτησης, από την στιγμή που αυτή προστέθηκε στην ουρά του guest. Όπου δεν διατυπώνουμε ρητά μια διαφοροποίηση των δυο παραπάνω μετρικών, θεωρούμε ότι αυτές συμφωνούν, κάτι που άλλωστε ήταν και η κυρίαρχη τάση στις μετρήσεις που εκτελέσαμε.

5.2.1 Μέγεθος τεμαχίου

Θα μετρήσουμε την απόδοση του οδηγού μας για διαφορετικά μεγέθη τεμαχίων, τόσο για επίπεδη συνταγή όσο και για δενδρική τριών επιπέδων. Η αύξηση του μεγέθους των τεμαχίων επιφέρει για σταθερό μέγεθος δίσκου, μείωση στον αριθμό των τεμαχίων και έμμεσα των κόμβων (λιγότεροι κόμβοι για να δεικτοδοτήσουν τα λιγότερα τεμάχια). Αναμένουμε όσο αυξάνεται το μέγεθος του τεμαχίου, τόσο να αυξάνεται και η απόδοση, αφού για την ίδια αίτηση του guest εμπλέκονται λιγότερα αρχεία δεδομένων και μεταδεδομένων και άρα έχουμε λιγότερες λειτουργίες Ε/Ε. Επίσης μικρότερα τεμάχια συνεπάγονται μικρότερες ανάγκες δεικτοδότησης και άρα μικρότερες συνταγές. Η μείωση στο μέγεθος των συνταγών, επιφέρει και αυτή αύξηση της απόδοσης.

Σειριακή εγγραφή 1GB:
Στην clean περίπτωση (αδέσμευτα μεταδεδομένα, κενός δίσκος) από 64KB και μετά οι διαφορές στο bandwidth είναι ελάχιστες και κρίνονται περισσότερο από τον θόρυβο. Παρόμοια συμπεριφορά έχουμε και για την COW περίπτωση, όπου η σχετική σταθεροποίηση είναι για 128KB.
Στην allocated περίπτωση (δεσμευμένα μεταδεδομένα) παρατηρούμε μια αύξουσα τάση η οποία είναι ελαφρώς πιο έντονη μέχρι τα 16KB.
Τα παραπάνω ισχύουν τόσο για 1 επίπεδο όσο και για 3 επίπεδα, καθώς οι δυο καμπύλες είναι γενικά παρόμοιες, ειδικά έως τα 32KB. Μάλιστα οι διαφορές μεταξύ του ενός και των τριών επιπέδων είναι υπαρκτές αλλά μικρές σε σχέση με την αύξηση του μεγέθους των τεμαχίων, ειδικά για την περίπτωση των clean και COW. Τα παραπάνω είναι λογικά, αφού μειώνονται τα εμπλεκόμενα αρχεία, και επίσης μειώνεται και το μέγεθος των μεταδεδομένων (entries) που προσπελάσονται.

Τυχαία εγγραφή και ανάγνωση 50MB σε τυχαία σημεία που βρίσκονται σε εύρος 1GB στον δίσκο:
Αρχικά εκτελέσαμε το πείραμα σε ένα κενό δίσκο, με αιτήσεις των 4KB. Εδώ για 1 επίπεδο και στις τρεις περιπτώσεις φαίνεται να έχουμε μια καμπύλη «καμπάνας» στην οποία η μέγιστη απόδοση παρατηρείται για 16KB. Αυτό οφείλεται στο ότι τα τεμάχια που αντιστοιχούσαν σε απομακρυσμένους τομείς του εικονικού δίσκου δεσμεύτηκαν σειριακά και άρα κοντά στον φυσικό δίσκο. Έτσι έχουμε τον συγκερασμό δυο αλληλοσυγκρουόμενων παραμέτρων: από τη μία, για χαμηλό μέγεθος τεμαχίων έχουμε πολλά αρχεία που αυξάνουν τις λειτουργίες Ε/Ε και άρα μειώνουν την απόδοση, και από την άλλη, για υψηλό μέγεθος τεμαχίων έχουμε εγγραφές σε μια αρκετά εκτενέστερη περιοχή του δίσκου, κάτι που λόγω της αργής μετακίνησης της κεφαλής ενός HDD μειώνει επίσης την απόδοση. Η περιοχή του δίσκου θα είναι εκτενέστερη γιατί ο διπλασιασμός του μεγέθους τεμαχίων δεν ακολουθείται από υποδιπλασιασμό των τεμαχίων-αρχείων που προσπελάσονται, καθώς οι αιτήσεις των 4KB γίνονται σε τυχαία σημεία του δίσκου.

Για να άρουμε τον 2ο παράγοντα εξετάζουμε τα πραγματικά τυχαίες εγγραφές σε έναν εικονικό δίσκο του οποίου τα τεμάχια έχουν δεσμευτεί σειριακά. Έτσι οι τυχαίες αιτήσεις που εκτελούμε αναφέρονται στο ίδιο εύρος του δίσκου.
Πράγματι, στην allocated περίπτωση τώρα διαπιστώνουμε μια συνεπή αύξηση της απόδοσης και στις αναγνώσεις και στις εγγραφές όσο αυξάνεται το μέγεθος τεμαχίου,

πράγμα που οφείλεται στην μείωση των αρχείων. Τα δεδομένα και είναι κατανεμημένα στην ίδια περιοχή του δίσκου και είναι σταθερά. Η μείωση των αρχείων συνεπάγεται αύξηση της απόδοσης γιατί για μεγάλο μέγεθος τεμαχίου περισσότερες αιτήσεις θα πέφτουν στο ίδιο τεμάχιο. Αυτό σημαίνει αφενός ότι το entry για αυτό το τεμάχιο θα έχει ξαναπροσπελαστεί και άρα θα υπάρχει στην host page cache, και αφετέρου ότι το inode block του τεμαχίου θα υπάρχει στην inode cache. Και οι δυο caches μας γλιτώνουν από λειτουργίες Ε/Ε. Στην COW περίπτωση όμως, η αντιγραφή του τεμαχίου παίρνει χρόνο ανάλογο με το μέγεθος του τεμαχίου, κάτι που δεν ήταν αλήθεια για απλές εγγραφές, καθώς γραφόταν μόνο η περιοχή του τεμαχίου που άλλαζε. Η αύξηση του χρόνου προσπέλασης συμβαίνει μάλιστα εις διπλούν, αφού πρέπει να γίνει και μεγαλύτερη ανάγνωση και μεγαλύτερη εγγραφή. Επειδή λοιπόν έχουμε τυχαίες εγγραφές 4KB, για μεγάλα μεγέθη τεμαχίων θα αντιγράφονται πολύ περισσότερα δεδομένα απ' όσα πραγματικά προσπελάσονται. Έτσι αυτός ο παράγοντας συγκρούεται με αυτόν της μείωσης των αρχείων και για αυτό προκύπτει η καμπανοειδής καμπύλη.

Οι παραπάνω παρατηρήσεις επεκτείνονται και για δέντρο μεγαλύτερου ύψους. Για 3 επίπεδα μάλιστα, αναμένουμε όσο μειώνεται το μέγεθος τεμαχίων τόσο θα αυξάνονται και τα μεταδεδομένα, καθώς οι κόμβοι αυξάνονται και σε αριθμό και σε μέγεθος, και άρα τα μεγάλα μεγέθη αναμένεται να είναι καλύτερα από τα μικρά με μεγαλύτερη διαφορά απ' ό,τι στην επίπεδη δομή. Πράγματι, στις τυχαίες προσπελάσεις φαίνεται ότι για μικρά μεγέθη τεμαχίων τα παραπάνω ισχύουν, αν εξαιρέσουμε τα 4KB και 8KB, όπου η διαφορά απόδοσης σε σχέση με την επίπεδη δομή κρύβεται εξαιτίας της ήδη μεγάλης επιβάρυνσης του τελευταίου επιπέδου. Ενδεικτικά στα 8KB-64KB έχουμε 15%-35% υπεροχή του δέντρου ύψους 1 σε σχέση με το δέντρο ύψους 3, ενώ στα 128KB-1MB η διαφορά απόδοσης περιορίζεται σε ποσοστό μικρότερο του 10%.

Τα παραπάνω αποτελέσματα είναι βασισμένα στο bandwidth. Το μέσο latency μιας αίτησης ακολουθεί γενικώς αυτά τα αποτελέσματα. Η μόνη ειδική παρατήρηση που μπορεί να γίνει, είναι ότι για τυχαίες εγγραφές, το latency για 64KB είναι πολύ καλύτερο από αυτό για μεγέθη μικρότερα ή ίσα των 32KB.

Γενικά, λοιπόν, ισχύει το αναμενόμενο ότι η αύξηση του μεγέθους του τεμαχίου επιφέρει και αύξηση της απόδοσης. Από τη γενική επισκόπηση των διαγραμμάτων μας, η μόνη κατηγορία λειτουργιών Ε/Ε όπου μετριάζεται η επίδραση του μεγέθους τεμαχίων στην απόδοση είναι αυτή του sequential read/write σε allocated δίσκο. Αυτό

ισχύει γιατί σε αντίθεση με τις clean και COW περιπτώσεις, δεν χρειάζεται να εκτελέσει εργασίες που είναι επιφορτισμένες με δημιουργία αρχείων και οι οποίες εξαρτώνται έντονα από τον αριθμό των αρχείων και άρα από το μέγεθος τεμαχίου. Επίσης στα τυχαία μοτίβα, το μεγαλύτερο μέγεθος τεμαχίου αυξάνει τις πιθανότητες κάποιες αιτήσεις Ε/Ε του guest να καταλήξουν στο ίδιο τεμάχιο και άρα να εξυπηρετηθούν γρηγορότερα γιατί το entry της συνταγής και το inode block του τεμαχίου θα είναι ήδη φορτωμένα στη RAM. Από την άλλη, μεγαλύτερο μέγεθος τεμαχίου συνεπάγεται μικρότερες ανάγκες δεικτοδότησης και άρα στα σειριακά μοτίβα μπορεί να γίνει αισθητή η ανάγκη για προσπέλαση λιγότερων μεταδεδομένων (entries).

Συνοψίζοντας τα παραπάνω έχουμε τα εξής πιο συγκεκριμένα συμπεράσματα:

Α. Ακόμα και από πλευράς απόδοσης δεν υπάρχει ουσιαστικός λόγος να έχουμε τεμάχια πάνω από 128KB, αφού οι βελτιώσεις είναι ανεπαίσθητες (εκτός ίσως από την sequential write allocated περίπτωση).

Β. Εάν μας ενδιαφέρει μεγάλη απόδοση στην COW περίπτωση πρέπει να επιλέξουμε κάπου μεταξύ 32-128KB, όπου στα σειριακά μοτίβα μειώνεται η αύξηση της απόδοσης και στα τυχαία εμφανίζει την κορυφή της καμπάνας (στα 64KB).

Γ. Αν θέλουμε να έχουμε ψηλή απόδοση και χαμηλό deduplication → 128KB.
Αν θέλουμε χαμηλή απόδοση και υψηλό deduplication → 16KB
Αν θέλουμε μέτρια απόδοση και μέτριο deduplication → 32KB ή 64KB
Οι επιβαρύνσεις για 4KB, 8KB θεωρούμε ότι είναι πολύ υψηλές για πρωτεύουσα αποθήκευση, παρόλο που στην βιβλιογραφία ενδείκνυνται περισσότερο για υψηλό deduplication.

Δ. Αν μας ενδιαφέρει το latency ενός random read/write request πρέπει να προτιμήσουμε μέγεθος 64KB, καθώς έχει μεγάλη διαφορά από τα μικρότερά του.

Ε. Η μόνη περίπτωση που μετριάζεται η επίδραση του μεγέθους τεμαχίων στην απόδοση είναι αυτή των σειριακών αναγνώσεων/εγγραφών σε allocated δίσκο.

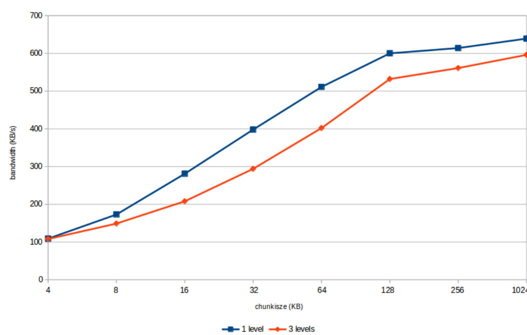
Ο λόγος που συμβαίνουν τα παραπάνω είναι κυρίως γιατί με την αύξηση του μεγέθους τεμαχίου εμπλέκονται λιγότερα αρχεία (τεμάχια και κόμβοι) τόσο στα σειριακά όσο και στα τυχαία μοτίβα. Επίσης, μειώνεται και ο αριθμός των entries της συνταγής που προσπελάσονται. Στα σειριακά μοτίβα, η αύξηση δεν είναι βέβαια γραμμική αφού παρά τον υποδιπλασιασμό των τεμαχίων, μεγαλώνει το μέγεθος κάθε λειτουρ-

γίας Ε/Ε. Μετά από 128KB η διαφορά στην ελάφρυνση είναι μικρή, αφού σταματά να γίνεται αισθητή η εξοικονόμηση διαχειριστικών λειτουργιών του host (inode table, εύρεση σε directory data blocks κτλ). Στην COW περίπτωση έχουμε καμπάνα στα τυχαία μοτίβα λόγω του φαινομένου του εσωτερικού κατακερματισμού, όπου αντιγράφονται περισσότερα δεδομένα απ᾿ όσα πρέπει.

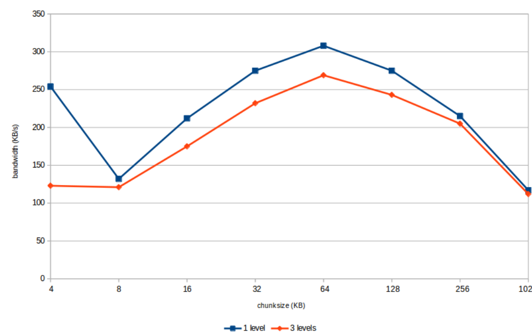
Ως επιπρόσθετη παρατήρηση, εισαγωγική για το επόμενο κομμάτι μετρήσεων, σημειώνουμε πως:

ΣΤ. για όλα τα μοτίβα Ε/Ε πέραν των σειριακών εγγραφών, η απόδοση των επιπέδων 1 και 3 είναι σχετικά κοντά.

Ως ενδεικτικό των παραπάνω παραθέτουμε τα διαγράμματα τυχαίων εγγραφών σε allocated και COW περίπτωση.



Σχήμα 5.1: *Απόδοση τυχαίων εγγραφών με δεσμευμένα μεταδεδομένα δίσκου*



Σχήμα 5.2: *Απόδοση τυχαίων εγγραφών για COW δεδομένων*

5.2.2 Αριθμός επιπέδων

Με βάση τις μετρήσεις που κάναμε για το μέγεθος του τεμαχίου, παραβλέπουμε μεγέθη μικρότερα των 16KB ως υπερβολικά επιβαρυντικά και μεγαλύτερα των 128KB ως ασύμφορα αφού χάνουμε σε deduplication και κερδίζουμε ελάχιστα σε απόδοση. Οι μετρήσεις μας θα επικεντρωθούν λοιπόν στα μεγέθη των 16KB, 32KB, 64KB, 128KB.

Εδώ αναμένουμε όσο αυξάνεται ο αριθμός των επιπέδων να μειώνεται και η απόδοση. Αυτό συμβαίνει γιατί για το ίδιο μέγεθος τεμαχίου και άρα για τον ίδιο αριθμό τεμαχίων έχουμε ότι το μεγαλύτερο ύψος του δέντρου συνεπάγεται προσπέλαση περισσότερων κόμβων και άρα περισσότερες λειτουργίες Ε/Ε. Η ανάγκη για προσπέλαση

περισσότερων κόμβων προκύπτει αφ' ενός επειδή για την προσπέλαση ενός τεμαχίου πρέπει πρώτα να προσπελάσουμε τόσους κόμβους όσους και το ύψος του δέντρου, δηλαδή με κάθε επιπλέον ύψος έχουμε αύξηση μονοπατιού για την προσπέλαση ενός τεμαχίου. Αφετέρου στη γενική περίπτωση που θέλουμε να προσπελάσουμε ένα μεγάλο αριθμό από τεμάχια, χρειάζεται να προσπελάσουμε στο επίπεδο των κόμβων-φύλλων τα κατάλληλα entries τους. Επειδή όμως για σταθερό μέγεθος δίσκου, με την αύξηση του ύψους μειώνεται ο αριθμός των entries ενός κόμβου (epn), τα entries που πρέπει να προσπελάσουμε είναι διασκορπισμένα σε περισσότερους κόμβους και αναπόφευκτα πρέπει να προσπελάσουμε περισσότερους κόμβους. Αυτός είναι ο ίδιος δομικός λόγος που καθιστά δυνατή την εξοικονόμηση μεταδεδομένων μεταξύ των στιγμιοτύπων. Οι περισσότεροι κόμβοι-φύλλα για ένα μεγαλύτερο ύψος, επιφέρουν και αυξημένες ανάγκες δεικτοδότησης στο δεύτερο επίπεδο το οποίο ισχύει και αναδρομικά για το τρίτο κ.ο.κ. Έτσι έχουμε και περισσότερα μεταδεδομένα (entries) σε σχέση με το αντίστοιχο επίπεδο του χαμηλότερου δέντρου και τα μεταδεδομένα αυτά είναι κατανεμημένα σε περισσότερους κόμβους εξαιτίας του μικρότερου epn. Περισσότερες λεπτομέρειες για αυτήν εξάρτηση στην 5.3.1.

Με βάση τις πρωτογενείς αιτίες που αναλύσαμε στην αρχή έχουμε ότι προφανώς η προσπέλαση αρχείων αυξάνεται για τους λόγους που προαναφέραμε, ενώ η αύξηση των μεταδεδομένων των εσωτερικών κόμβων συνηγορεί στην αύξηση και της ποσότητας των data blocks που διαβάζουμε/γράφουμε.

Πράγματι οι μετρήσεις μας, τόσο για το bandwidth όσο και για το latency, επιβεβαιώνουν ότι για όλα τα μεγέθη τεμαχίων που εξετάσαμε, η απόδοση για ένα ύψος x είναι υψηλότερη από αυτήν για ύψη μεγαλύτερα του x.

Ας μελετήσουμε όμως τώρα τον ρυθμό με τον οποίο πέφτει αυτή η απόδοση για κάθε φάση του δίσκου και για κάθε μέγεθος τεμαχίου ξεχωριστά, με την ελπίδα ότι θα εντοπίσουμε σε ένα συμφέροντα συνδυασμό.

Clean περίπτωση:

Σειριακή εγγραφή και ανάγνωση 1GB.

16KB:	μέχρι και 3 επίπεδα δεν χάνεται απόδοση, μέχρι και 5 μικρή πτώση
32KB:	μέχρι και 3 επίπεδα δεν χάνεται απόδοση, μέχρι και 5 μικρή πτώση
64KB:	συνεχόμενη πτώση απόδοσης, ειδικά από 1-3 και από 5-10 επίπεδα
128KB:	μέχρι και 5 επίπεδα δεν χάνεται απόδοση

Επίσης η μείωση του μεγέθους τεμαχίου έχει πιο δραματικές συνέπειες για την απόδοση από την αύξηση των επιπέδων. Ενδεικτικό είναι ότι κατά κανόνα, η απόδοση για 10 επίπεδα ενός μεγέθους είναι καλύτερη από την απόδοση του προηγούμενου μεγέθους για 1 επίπεδο. Αυτό οφείλεται στο γεγονός ότι τα αρχεία των τεμαχίων που πρέπει να δημιουργηθούν είναι πολύ περισσότερα από τους κόμβους, οπότε ο χρόνος δημιουργίας τους και ο χρόνος για την εγγραφή των entries του τελευταίου επιπέδου, υποσκελίζουν τον χρόνο δημιουργίας και εγγραφής των κόμβων.

Με βάση τα παραπάνω καταλήγουμε στα εξής συμπεράσματα για την clean φάση του δίσκου:

A. Το μέγεθος τεμαχίου επιδρά στον clean δίσκο πολύ πιο καθοριστικά απ' ό,τι τα επίπεδα, λόγω επικράτησης του χρόνου δημιουργίας των τεμαχίων και των αντίστοιχων entries τους. Αν θέλουμε να έχουμε γρήγορη δέσμευση τεμαχίων πρέπει να διαλέξουμε 128KB ή 64KB και μέχρι 5 level. To latency επιτρέπει και 32KB, όχι όμως και 16KB.

B. Με εξαίρεση τα 64KB, δεν υπάρχει μεγάλη μείωση απόδοσης μέχρι τα 3 level

Allocated περίπτωση:

Σειριακή εγγραφή και ανάγνωση 1GB:

16KB: τεράστια πτώση από 1 στα 2 επίπεδα μόνο για write αιτήσεις, μικρή μόνο πτώση μέχρι τα 3

32KB: τεράστια πτώση από 1 στα 2 επίπεδα μόνο για write αιτήσεις, μικρή μόνο πτώση μέχρι τα 3

64KB: τεράστια πτώση από 2 στα 3 επίπεδα μόνο για write αιτήσεις, μικρή μόνο πτώση μέχρι τα 3, για 3 και πάνω πέφτουν μόνο οι write αιτήσεις

128KB: σταδιακή πτώση για 1-3 επίπεδα, μικρή μόνο πτώση μέχρι τα 3

Εδώ δεν παρατηρείται το να έχει μεγαλύτερη βαρύτητα το μέγεθος τεμαχίου, κάτι που οφείλεται στο ότι δεν έχουμε δημιουργία τεμαχίων και ενημέρωση των αντίστοιχων entries τους.

Οι τεράστιες πτώσεις που παρατηρούνται στην περίπτωση των σειριακών εγγραφών οφείλονται κυρίως στο γεγονός ότι σε μια επίπεδη συνταγή προσπελάζεται μόλις ένα αρχείο, κάτι που μπορεί να εκμεταλλευτεί πλήρως την τοπικότητα και την ταχύτητα του δίσκου σε σειριακές προσπελάσεις. Αντίθετα σε μεγαλύτερο ύψος, επιμερίζομε τις προσπελάσεις για την εύρεση των τεμαχίων σε πολλά αρχεία, κάτι που ανακόπτει την

μεγάλη ταχύτητα. Αυτό μάλιστα είναι ακόμα πιο έντονο για μικρά μεγέθη τεμαχίων, κάτι που είναι λογικό, αφού έχουμε περισσότερα αρχεία στο τελευταίο επίπεδο.
Το παραπάνω φαινόμενο μάλιστα παρατηρείται μόνο στις λειτυργίες εγγραφής, κάτι το οποίο είναι απολύτως φυσικό. Η παραπάνω διάσπαση των προσπελάσεων σε πολλά αρχεία αντί για ένα δεν θα έπρεπε να επιδράσει τρομερά στην απόδοση, αφού η σειριακή προσπέλαση γίνεται με τον ίδιο τρόπο που έγινε και η δέσμευση των αρχείων, οπότε το μοτίβο αυτό θα εκμεταλλευτεί πλήρως την τοπικότητα, με την κεφαλή του δίσκου να μετακινείται λίγο και προς μια μόνο κατεύθυνση. Πιθανώς μάλιστα οι προσπελάσεις να ομαδοποιούνται σε ένα ή λίγα αιτήματα Ε/Ε από τον I/O scheduler. Για αυτό στις αναγνώσεις δεν έχουμε ουσιαστική πτώση. Η πτώση στις εγγραφές οφείλεται στο ότι για την εγγραφή κάθε αρχείου κόμβου θα πρέπει επίσης να εγγραφούν τα μεταδεδομένα, τόσο στο journal, όσο και στα inode block. Αυτό επιφέρει την μετακίνηση της κεφαλής σε απομακρυσμένες θέσεις στο δίσκο, εισάγοντας καθυστέρηση.

Τυχαία εγγραφή και ανάγνωση 50M σε περιοχή 1GB:

- 16KB: ίδια απόδοση σε 1 και 2 επίπεδα, υπάρχει πτώση απόδοσης στα 3 και από εκεί και μετά πέφτει σταδιακά
- 32KB: ίδια απόδοση σε 1 και 2 επίπεδα, υπάρχει πτώση απόδοσης στα 3 και από εκεί και μετά πέφτει σταδιακά
- 64KB: πολύ κοντινές αποδόσεις για 1 και 2 επίπεδα, μικρή μόνο πτώση μέχρι τα 3 (μετά πέφτει έντονα μόνο το write)
- 128KB: πολύ κοντινές αποδόσεις για 1-3 επίπεδα, μικρή μόνο πτώση μέχρι τα 3

Αρχικά να τονίσουμε ότι εξαιτίας του φαινομένου της έλλειψης τοπικότητας για τυχαίες εγγραφές σε κενό δίσκο, το οποίο περιγράψαμε προηγουμένως, εκτελέσαμε την μέτρηση για τεμάχια δεσμευμένα με σειριακό και όχι με τυχαίο τρόπο.

Στις τυχαίες εγγραφές έχουμε επίσης το φαινόμενο της πολύ γρήγορης αύξησης του latency από τα 3 επίπεδα και μετά, ενώ για 1 ή 2 είναι σχεδόν πανομοιότυπο. Είναι λογικό να μην αυξάνεται το latency μέχρι τα δύο επίπεδα, αφού για δίσκο 10GB που εξετάσαμε, ο κόμβος-ρίζα είναι μικρότερος από το 4KB-μπλοκ του συστήματος αρχείων του host, οπότε μετά την πρώτη προσπέλαση τοποθετείται στην RAM και δεν επιβαρύνει το latency. Το παραπάνω γενικεύεται και για μεγαλύτερους δίσκους, αφού για παράδειγμα για 1TB αρκούν ένα-δυο μπλοκ (ανάλογα με το μέγεθος τεμαχίων) για να φορτωθεί όλος ο κόμβος στην RAM. Η υλοποίησή μας αποφεύγει και επανειλημμένες κλήσεις συστήματος `open()` για τον ριζικό κόμβο, αφού τον κρατάει ανοικτό.

Με βάση τα παραπάνω καταλήγουμε ότι στην allocated περίπτωση:

Α. Στα σειριακά μοτίβα εγγραφής, υπάρχει πτώση απόδοσης από το 1 στα 2 επίπεδα εξαιτίας του επιμερισμού των προσπελάσεων μεταδεδομένων σε πάνω από 1 αρχεία, κάτι που πυροδοτεί πολλαπλές εγγραφές στο journal και περιορίζει την σειριακή λειτουργία του δίσκου.

Β. Στα τυχαία μοτίβα δεν παρατηρείται ουσιαστική πτώση απόδοσης από 1 στα 2 επίπεδα, αφού ο κόμβος-ρίζα φορτώνεται στην κύρια μνήμη.

Γ. Η απόδοση είναι ανεκτή μέχρι και τα 3 επίπεδα, αν και για τυχαία μοτίβα και μεγέθη τεμαχίων 16KB και 32KB πέφτει αισθητά σε σχέση με τα 2, ενώ για σειριακά μοτίβα πέφτει μόνο για 64KB.

COW:

Σειριακή εγγραφή 1GB:

16KB: μικρή μόνο πτώση απόδοσης μέχρι 3 επίπεδα, γενικώς σταδιακή πτώση από 1-5, πιο γρήγορα αυξάνεται το latency

32KB: ίδια απόδοση για 1 και 2 επίπεδα, από 3 και πάνω πέφτει σταδιακά

64KB: παρόμοια απόδοση για 1 και 2 επίπεδα, σταδιακή πτώση από 3-5

128KB: κοντινή απόδοση για 1-3 επίπεδα, μικρή πτώση μέχρι 3, γενικώς σταδιακή πτώση 1-5

Τυχαία εγγραφή 50M σε περιοχή 1GB:

16KB: ίδια απόδοση για 1 και 2 επίπεδα, από 3 και πάνω πέφτει κατακόρυφα σε μη ανεκτά επίπεδα

32KB: ίδια απόδοση για 1 και 2 επίπεδα, από 3 και πάνω πέφτει σταδιακά

64KB: παραπλήσια απόδοση για 1-3 επίπεδα, από εκεί και μετά πέφτει σταδιακά

128KB: σχετικά μικρή πτώση για 1-3 επίπεδα,από εκεί και μετά πέφτει σταδιακά

Να σημειώσουμε πως, όπως αναφέραμε και προηγουμένως, στα τυχαία COW έχουμε το φαινόμενο της «καμπάνας» όσο αφορά τα μεγέθη τεμαχίων.

Είναι λογικό να έχουμε παρεμφερείς επιδόσεις στο 1 και στα 2 επίπεδα, αφού εδώ κυριαρχεί ο χρόνος του COW, ο οποίος για λίγα επίπεδα κυριαρχείται κυρίως από το COW των τεμαχίων.

Συμπεράσματα για COW περίπτωση:

Α. Για σειριακά COW έχουμε αποδεκτή απόδοση μέχρι και τα 3 επίπεδα και γενικό-

τερα σταδιακή πτώση μέχρι τα 5 επίπεδα

Β. Για τυχαία COW έχουμε για μικρά μεγέθη τεμαχίων ανεκτή απόδοση μέχρι 2 επίπεδα και για μεγάλα μεγέθη μέχρι 3. Από εκεί και πέρα έχουμε σταδιακή πτώση

Γ. Εν γένει έχουμε παραπλήσιες αποδόσεις μεταξύ των επιπέδων 1 και 2.

Συμπεράσματα για αριθμό επιπέδων

Γενικά, λοιπόν, ισχύει το αναμενόμενο ότι η αύξηση του ύψους επιφέρει και μείωση της απόδοσης, λόγω περισσότερων κόμβων και άρα αυξημένων λειτουργιών Ε/Ε.

Πιο συγκεκριμένα, συνοψίζοντας τα συμπεράσματα για κάθε μία από τις παραπάνω φάσεις του δίσκου έχουμε χονδρικά τα εξής τρία γενικά συμπεράσματα:

Α. Έχουμε μικρή μόνο μείωση απόδοσης μέχρι 3 επίπεδα

Β. Έχουμε παραπλήσιες αποδόσεις μεταξύ των επιπέδων 1 και 2 με εξαίρεση την περιπτώση των σειριακών εγγραφών σε allocated δίσκο

Γ. Οι πτώσεις απόδοσης από τα 2 στα 3 επίπεδα εντοπίζονται σε όλα τα τυχαία μοτίβα για μικρά μεγέθη τεμαχών 16KB, 32KB, στα σειριακά COW εκτός των 128KB και στα σειριακά writes σε allocated ή clean δίσκο για 64KB

Όπως αναφέρθηκε στο κεφάλαιο του σχεδιασμού, η δενδρική δομή (πάνω από 1 επίπεδο) χαρίζει το μεγάλο πλεονέκτημα της ακαριαίας λήψης στιγμιοτύπου. Συνεπώς ωθούμενοι και από το δεύτερο συμπέρασμα που βρίσκει ότι δεν υπάρχουν μεγάλες διαφορές μεταξύ ενός και δύο επιπέδων, περιορίζουμε το πεδίο των παραμέτρων μας για 2 και πάνω επίπεδα. Ακόμη, το πρώτο συμπέρασμα φαίνεται να θέτει ένα άνω φράγμα στον αριθμό των επιπέδων στα 3, οπότε καταλήγουμε ότι ο κατάλληλος αριθμός επιπέδων χωρίς να θυσιάσουμε σημαντικά την απόδοση είναι τα 2 ή 3. Το τρίτο συμπέρασμα μας δίνει μια ιδέα για τις περιπτώσεις που εντοπίζεται πτώση απόδοσης από τα 2 στα 3 επίπεδα. Μάλιστα, επανεξετάζοντας τα διαγράμματα της 5.2.1, μπορούμε να επιβεβαιώσουμε ότι για όλες τις περιπτώσεις Ε/Ε πέραν του sequential write, δεν παρουσιάζεται μεγάλη υπονόμευση της απόδοσης συγκριτικά με τη μη δενδρική δομή, ειδικά σε σχέση με την επίδραση του μεγέθους τεμαχίου.

Ανακεφαλαιώνοντας, στο υπόλοιπο της πειραματικής μας αξιολόγησης θα περιοριστούμε στο εξής εύρος παραμέτρων:

μεγέθη τεμαχίων: 16KB, 32KB, 64KB, 128KB   ύψος δέντρου: 2, 3

5.2.3 Χρόνος λήψης στιγμιοτύπου

Εδώ περιμένουμε μια τρομακτική διαφορά στο χρόνο λήψης στιγμιοτύπου από 1 σε 2 επίπεδα, αφού αντί για ολόκληρη την συνταγή θα πρέπει να αντιγραφεί μόνο ο ριζικός κόμβος. Δεδομένου μάλιστα ότι ήδη με 2 επίπεδα καταφέρνουμε για όλα τα μεγέθη τεμαχίων $\geq$4KB και για όλους τους δίσκους $\leq$1TB ο ριζικός κόμβος να έχει μέγεθος μικρότερο από MB, δεν περιμένουμε και διαφορές για πάνω από 2 επίπεδα.

Παραθέτουμε τα αποτελέσματα για δίσκο 1TB και διάφορα μεγέθη τεμαχίων σε KB (γραμμές) και επίπεδα του δέντρου (διπλές στήλες). Για κάθε επίπεδο δέντρου (διπλή στήλη), η πρώτη στήλη περιέχει το μέγεθος του ριζικού κόμβου σε KB και η δεύτερη τον χρόνο λήψης στιγμιοτύπου σε δευτερόλεπτα.

	1		2		3		5	
4	268435	255.618	361	0.949	14.3	0.711	1.5	0.684
8	134217	129.353	255	0.772	11.8	0.708	1	0.679
16	67108	69.188	181	0.767	9.2	0.705	1	$\sim$
32	33554	33.536	127	0.73	7.2	0.695	1	$\sim$
64	16777	17.004	90	0.718	6.1	0.697	1	$\sim$
128	8388	9.000	64	0.702	4.6	0.685	1	$\sim$

Όπως ήταν αναμενόμενο, από 2 επίπεδα και πάνω οι διαφορές είναι ανύπαρκτες. Η μόνη ουσιαστική διαφορά παρατηρείται στα 4KB που έχουμε κατά 150ms πιο αργή λήψη από το 8KB, ενώ τα 3 επίπεδα είναι ελάχιστα πιο γρήγορα, λόγω του μικρότερου μεγέθους της συνταγής. Από εκεί και πέρα δεν υπάρχουν ουσιαστικές διαφορές στο μέγεθος του ριζικού κόμβου και άρα στον χρόνο λήψης στιγμιοτύπου.

Επίσης όπως είναι λογικό, στην επίπεδη δομή ο υποδιπλασιασμός του μεγέθους τεμαχίου επιφέρει διπλασιασμό του χρόνο λήψης στιγμιοτύπου, αφού έχουμε διπλάσια τεμάχια και άρα διπλάσια entries στη συνταγή και άρα διπλάσιο μέγεθος συνταγής. Για δίσκο μεγέθους 500GB έχουμε περίπου 650ms για όλα μεγέθη εκτός των 4KB που πήρε 780ms. Οι διαφορές μεταξύ των επιπέδων τόσο στο 1TB όσο και στα 500GB είναι ανεπαίσθητες, αφού έχουμε να κάνουμε με μόλις δυο μικρού μεγέθους αιτήσεις Ε/Ε. Στην πραγματικότητα επειδή αδειάζουμε την host page cache πριν τη λήψη, έχουμε και άλλη μια αίτηση ανάγνωσης του ριζικό κόμβου από το δίσκο. Χωρίς αυτήν οι χρόνοι κατεβαίνουν στα 0.4s. Λογικό είναι άλλωστε και το ότι παρατηρούμε μια διακύμανση σε αυτούς του χαμηλούς χρόνους, αφού στον χρόνο ολοκλήρωσης της λήψης

στιγμιοτύπου υπεισέρχονται πια με μη αμελητέο τρόπο και παράγοντες όπως ο χρονοδρομολογητής του λειτουργικού συστήματος, ο I/O scheduler, αλλά ακόμα και η τυχαία αρχική θέση της κεφαλής του δίσκου.

Ως βασικό συμπέρασμα λοιπόν προκύπτει ότι τα 2 επίπεδα λύνουν ουσιαστικά το πρόβλημα της ακαριαίας λήψης στιγμιοτύπου. Ο μοναδικός λόγος που συνηγορεί υπέρ της χρήσης τριών επιπέδων, λοιπόν, είναι αυτός του μεγαλύτερου καταμερισμού των entries σε κόμβους, κάτι που θα βοηθήσει μια διαδικασία συγχρονισμού με απομακρυσμένο backup αλλά και θα συμβάλει στην απαλοιφή μεταδεδομένων μεταξύ συνεχόμενων στιγμιοτύπων, εξαλείφοντας τον εσωτερικό πλεονασμό. Στην ενότητα 5.3.2 εξετάζουμε εάν αυτό το επιχείρημα ευσταθεί και εάν μπορεί να οδηγήσει πράγματι σε μακροπρόθεσμη εξοικονόμηση χώρου.

Προφανώς, πανομοιότυποι είναι και οι χρόνοι δημιουργίας ενός κλώνου, αφού έχουμε ακριβώς τα ίδια βήματα με αυτά της λήψης στιγμιοτύπου, δηλαδή δημιουργία μιας επικεφαλίδας συνταγής και αντιγραφή του ριζικού κόμβου. Η απουσία της μεταβολής των δικαιωμάτων του αρχικού ριζικού κόμβου στην περίπτωση του κλώνου, δίνει ελάχιστα ταχύτερους χρόνος και εξίσου αμελητέες διαφοροποιήσεις για δέντρα δύο και πάνω επιπέδων. Περιττό είναι επίσης να αναφέρουμε ότι ακαριαία είναι και η δημιουργία ενός πρωτεύοντος εικονικού δίσκου χάρη στο sparseness.

5.3 Μεταδεδομένα

Είναι θεμιτό τώρα, να παρουσιάσουμε την επιβάρυνση σε μεταδεδομένα. Για να το κάνουμε αυτό παρουσιάζουμε τα μεταδεδομένα για πλήρως γεμάτους δίσκους, ώστε να δούμε τις διαφορές ανάλογα το μέγεθος τεμαχίου, το ύψος του δέντρου και το μέγεθος του δίσκου.

Υπάρχουν δύο είδη μεταδεδομένων: τα φυσικά, που αναφέρονται στα μεταδεδομένα που απαιτούνται για την αντιστοίχιση μεταξύ των οντοτήτων μας και της τοποθεσίας αποθήκευσής τους στο σύστημα, και τα λογικά, που αναφέρονται στα μεταδεδομένα που περιέχουν οι συνταγές για την αντιστοίχιση τομέων με οντότητες. Στην περίπτωσή μας, αφού οι οντότητες είναι αρχεία, τα φυσικά μεταδεδομένα είναι απλώς ο χώρος που καταλαμβάνουν τα inodes στο σύστημα αρχείων και ο χώρος που καταλαμβάνουν τα ονόματα των αρχείων στα μπλοκ των καταλόγων (directory blocks των

chunk-store και node-store). Κάθε δομή inode ενός αρχείου καταλαμβάνει 256bytes, και κάθε όνομα αρχείου σε ένα μπλοκ καταλόγου καταλαμβάνει 48bytes. Σύνολο δηλαδή τα φυσικά μεταδεδομένα ανά αρχείο είναι 304bytes. Τα λογικά μεταδεδομένα συνίστανται στο μέγεθος που καταλαμβάνουν οι κόμβοι της δεντρικής συνταγής. Οι κόμβοι περιέχουν, πέρα από την επικεφαλίδα, entries, οι οποίες περιγράφουν το όνομα (id ή αποτύπωμα) ενός άλλου κόμβου/τεμαχίου και το δικαίωμα εγγραφής. Το μέγεθος κάθε εγγραφής στην υλοποίησή μας είναι 22bytes.

Επίσης θα υποδιαιρέσουμε τα μεταδεδομένα σε δύο άλλες κατηγορίες. Η πρώτη είναι τα καθαρά μεταδεδομένα, τα οποία συνίστανται στον όγκο που καταλαμβάνουν τα entries που απαιτούνται για την δεικτοδότηση των τεμαχίων ενός δίσκου. Τα καθαρά μεταδεδομένα, συνεπώς, είναι αμιγώς εξαρτώμενα από το μέγεθος τεμαχίου για το εκάστοτε μέγεθος δίσκου. Η δεύτερη κατηγορία είναι τα μεταδεδομένα που εισάγονται εξαιτίας της δενδρικής δομής και συνίστανται στον όγκο των entries των εσωτερικών κόμβων και των επικεφαλίδων όλων των κόμβων.

5.3.1 Υπολογισμός μεταδεδομένων

Καθαρά Μεταδεδομένα

Συγκρίνουμε αρχικά την ποσότητα των μεταδεδομένων που χρειαζόμαστε για να δεικτοδοτήσουμε τα τεμάχια. Αυτά είναι μεταδεδομένα που εξαρτώνται από το μέγεθος τεμαχίου και δεν μπορούμε να τα γλιτώσουμε ανεξαρτήτως επιπέδων. Στην δενδρική δομή συνιστούν το τελευταίο επίπεδο των κόμβων-φύλλων και η μοναδική διαφορά που εισάγει το ύψος του δέντρου είναι σε πόσους κόμβους αυτά τα μεταδεδομένα θα κατανεμηθούν. Παρατηρούμε ότι για κάθε μέγεθος δίσκου, έχουμε υποδιπλασιασμό των μεταδεδομένων με κάθε διπλασιασμό του μεγέθους τεμαχίου, κάτι που είναι λογικό, αφού η σχέση τους είναι: $meta = entry_size * \lceil \frac{disk_size}{chunksize} \rceil$. Βλέπουμε ότι για μεγέθη 4KB τα μεταδεδομένα αποτελούν γύρω στο 0.5% των πραγματικών δεδομένων του δίσκου, για 8KB έχουμε 0.28% και από 16KB έχουμε κάτω από 0.12%.

Τα παραπάνω αποτελέσματα προκύπτουν χωρίς να συνυπολογίσουμε και το χώρο που καταλαμβάνεται από τα inodes και τις εγγραφές με τα ονόματα των αρχείων στους καταλόγους, δηλαδή τα φυσικά μεταδεδομένα. Αν συνυπολογίσουμε και τα 304bytes που προαναφέραμε, έχουμε για 4KB-32KB τα μεταδεδομένα να είναι αντίστοιχα το 8%

4%, 2% και 1% του δίσκου. Ακόμη και αυτά αποτελούν μικρές ποσότητες, εάν εξασφαλιστεί ότι δεν θα αυξάνονται γραμμικά με την λήψη στιγμιοτύπων. Αγνοούμε όμως τα φυσικά μεταδεδομένα, αφού είναι ίδιον της υλοποίησής μας και σίγουρα μπορούμε με τον παρόντα σχεδιασμό να εφαρμόσουμε μια αντιστοίχιση ανάμεσα σε φυσική θέση του τεμαχίου και το αποτύπωμα ή το όνομά του που καταλαμβάνει πολύ λιγότερα bytes. Ενδεικτικά, θεωρώντας ότι το όνομα/αποτύπωμα ενός κόμβου/τεμαχίου καταλαμβάνει 20bytes και την διεύθυνση του κόμβου/τεμαχίου ένα 64bit αριθμό, έχουμε 28bytes, οπότε προστιθέμενα με τα προϋπάρχοντα 22bytes που θέλει μια συνταγή για κάθε τεμάχιο, τα 4KB καταλαμβάνουν το 1.2%, τα 8KB το 0.6% τα 16KB το 0.3% και από τα 32KB και μετά πέφτουμε σε κάτω από 0.15%.

Γίνεται εμφανές λοιπόν, ότι τα μεταδεδομένα των τεμαχίων είναι αρκετά μικρό κομμάτι του δίσκου, ειδικά για τα μεγέθη μεγαλύτερα των 8KB, που επιλέξαμε λόγω απόδοσης.

Μεταδεδομένα εισαγόμενα από το ύψος του δέντρου

Υπάρχουν δύο λόγοι που αυξάνονται τα μεταδεδομένα με το ύψος του δέντρου. Ο πρώτος και προφανής, είναι για κάθε επιπρόσθετο επίπεδο, προστίθενται κόμβοι που πρέπει να δεικτοδοτήσουν τους κόμβους του κατωτέρου επιπέδου. Ο δεύτερος και λίγο πιο σκιώδης, είναι ότι με την αύξηση του ύψους αυξάνεται ο αριθμός των κόμβων στους οποίους κατανέμονται τα μεταδεδομένα του πρώτου επιπέδου (φύλλα), με αυτήν την αύξηση να συμπαρασύρει και την ποσότητα των μεταδεδομένων σε κάθε άλλο επίπεδο. Για να γίνει αυτό κατανοητό, επαναλαμβάνουμε ότι τα entries ανά κόμβο εξαρτώνται από το ύψος του δέντρου και ορίζονται από τον τύπο $epn = \lceil \sqrt[h]{tc} \rceil$ της ενότητας 3.6. Έτσι, το τελευταίο επίπεδο ενός δίσκου με δεδομένο μέγεθος τεμαχίου, αν και δεικτοδοτεί πάντα τα ίδια τεμάχια και άρα έχει πάντα τον ίδιο αριθμό από entries, ανάλογα με το ύψος κατανέμει αυτά τα entries σε διαφορετικό αριθμό κόμβων. Όσο το ύψος αυξάνεται, το epn μειώνεται και άρα αφού τα entries είναι σταθερά, απαιτούνται περισσότεροι κόμβοι. Αυτή η αύξηση των κόμβων του κατωτέρου επιπέδου δημιουργεί αυξημένες ανάγκες δεικτοδότησης για το ανώτερο επίπεδο, το οποίο αφού αυξάνεται με την σειρά του συμπαρασύρει με αναδρομικό τρόπο και τα ανώτερα επίπεδα. Έτσι πέρα από τον επιπλέον κόμβο-ρίζα που προστίθεται κατά την αύξηση του ύψους του δέντρου, προστίθενται και περισσότερα entries σε κάθε άλλο επίπεδο πέραν του τελευταίου, για να καλύψουν τις αυξημένες ανάγκες δεικτοδότησης.

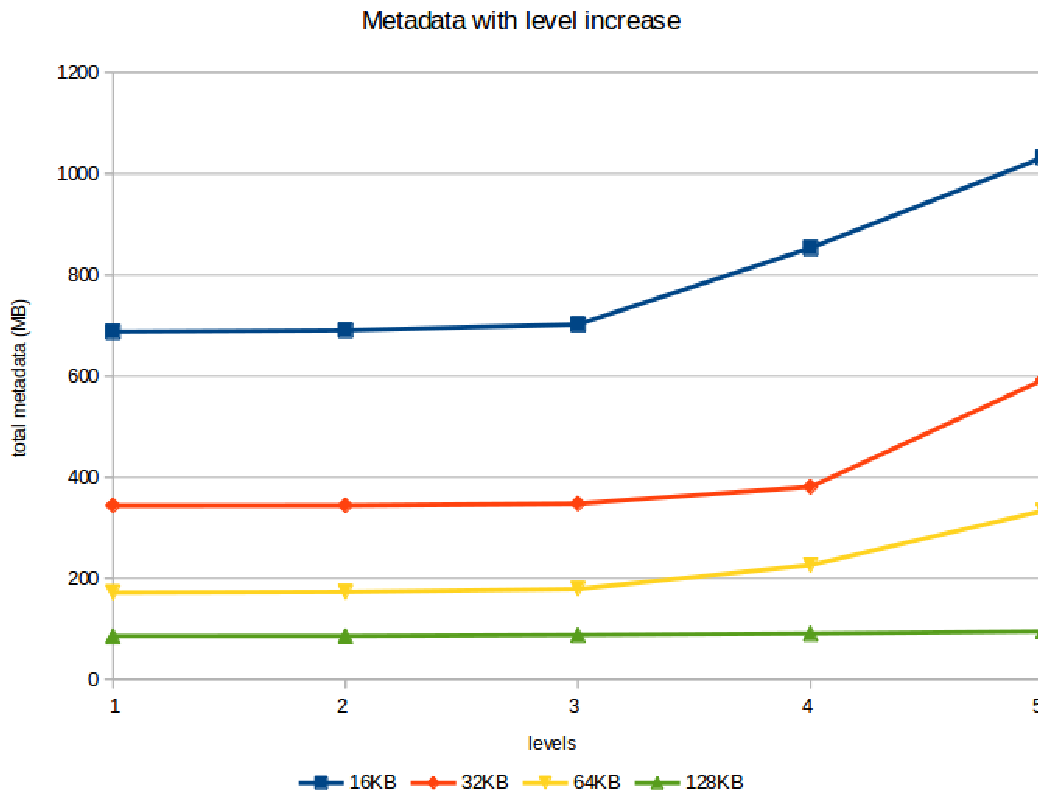
Επίσης δεν πρέπει να παραβλέψουμε και το ότι επιπλέον μεταδεδομένα δεν εισάγονται μόνο από τα επιπλέον entries, αλλά και από τις περισσότερες επικεφαλίδες κόμβων. Προαναφέραμε ότι στο επίπεδο των φύλλων έχουμε περισσότερους κόμβους για ένα υψηλό παρά για ένα χαμηλό δέντρο. Επειδή ένα επίπεδο περιέχει τόσους κόμβους όσους δεικτοδοτεί το προηγούμενό του, έχουμε ότι το πρώτο επίπεδο περιέχει τόσους κόμβους, όσα και τα entries του δευτέρου επιπέδου, δηλαδή epn^{h-1} κόμβους. Κατ' αναλογία το δεύτερο επίπεδο περιέχει epn^{l-2} κόμβους, και το επίπεδο x περιέχει epn^{h-x} κόμβους. Έτσι αναλύοντας το epn έχουμε ότι το επίπεδο έχει $(\lceil tc^h \rceil)^{h-x}$. Παραβλέποντας την στρογγυλοποίηση προς τα πάνω, κάνουμε τις πράξεις και καταλήγουμε στην $tc^{1-\frac{x}{h}}$ που μεταφράζεται ότι με την αύξηση του ύψους h, αυξάνεται ο εκθέτης και άρα επειδή αυτός είναι πάντα θετικός ($x < h$), ο αριθμός των κόμβων αυξάνεται. Αυτός ο ισχυρισμός θα ήταν πράγματι αληθής εάν δεν είχαμε την στρογγυλοποίηση προς τα πάνω για το epn, η οποία διαταράσσει την ομαλή μαθηματική συμπεριφορά. Ακόμα όμως και με την στρογγυλοποίηση, μετά από εξαντλητικούς υπολογισμούς για κάθε μέγεθος δίσκου από 1GB έως 1TB, και για κάθε μέγεθος τεμαχίου <1MB, παρατηρούμε ότι ο αρχικός μας ισχυρισμός είναι και πρακτικά αληθής για όλα τα δέντρα ύψους 2-5. Υπάρχουν κάποιοι συνδυασμοί παραμέτρων για τους οποίους οι κόμβοι ενός επιπέδου x είναι περισσότεροι στο δέντρο ύψους k παρά στο δέντρο ύψους l όπου $l > k$, αλλά αυτές οι περιπτώσεις είναι λίγες και παρατηρούνται συνήθως για $k \geq 7$. Όλο το παραπάνω για τον αριθμό τον κόμβων συνοψίζεται και στην εξής στην απλούστερη διαπίστωση: αφού τα entries κάθε επιπέδου (εκτός του πρώτου) αυξάνονται, και το epn μειώνεται, αναπόφευκτα θα έχουμε τους ίδιους ή περισσότερους κόμβους σε κάθε επίπεδο. Αυτή η αύξηση συμπαρασύρει λοιπόν και την ποσότητα των μεταδεδομένων, αφού για κάθε κόμβο πρέπει να τοποθετήσουμε την επικεφαλίδα του και πρέπει να δεσμεύσουμε ένα inode (φυσικά μεταδεδομένα).

Συνεπώς έχουμε αύξηση των μεταδεδομένων για ένα υψηλότερο δέντρο εξαιτίας:

1. των entries του εξτρα επιπέδου
2. των επιπλέον entries σε κάθε επίπεδο (αναδρομικά αυξημένες ανάγκες δεικτοδότησης)
3. των επιπλέον επικεφαλίδων των κόμβων (εν γένει αυξάνεται ο αριθμός των κόμβων κάθε επιπέδου)

Τώρα θα παρατηρήσουμε πειραματικά την αύξηση των μεταδεδομένων σε σχέση με

το ύψος του δέντρου, θα μελετήσουμε δηλαδή πόσο θα αυξηθούν τα μεταδεδομένα που εισάγονται από τους κόμβους των επιπλέον επιπέδων σε σχέση με τα απαραίτητα μεταδεδομένα για την δεικτοδότηση των τεμαχίων που περιγράψαμε παραπάνω.



Σχήμα 5.3: *Μεταδεδομένα με άυξηση των επιπέδων*

Αρχικά παρατηρούμε ότι για όλα τα μεγέθη δίσκων, η μετάβαση από τα 5 στα 7 επίπεδα πάντοτε αποτελεί σημείο καμπής όπου τα μεταδεδομένα αυξάνονται σημαντικά. Έτσι, σε συνδυασμό με τα προηγούμενα συμπεράσματα για την απόδοση ανάλογα με τα επίπεδα, καταλήγουμε στο ότι είναι τελείως ασύμφορα δέντρα με πάνω από 5 επίπεδα και συνεχίζουμε την ανάλυσή μας εξαιρώντας τα. Επίσης, όπως είναι επόμενο, τα 2 επίπεδα διαφέρουν ελάχιστα από τα καθαρά μεταδεδομένα, αφού προστίθεται μόλις ένας κόμβος μεγέθους μερικών kilobyte, και άρα συνυπολογίζοντας το πλεονέκτημα της ακαριαίας δημιουργίας στιγμιοτύπου, είναι πολύ προτιμότερα από το ένα επίπεδο.

Από εκεί και μετά, πέρα από την γενικότερη τάση αύξησης με την αύξηση των επιπέδων, δεν μπορούμε για όλα τα μεγέθη δίσκων να παρατηρήσουμε μια συγκεκριμένη μορφολογία στον τρόπο αύξησης. Οι ανωμαλίες αυτές και τα διαφορετικά σημεία καμπής προκύπτουν από την ιδιότυπη αλληλεπίδραση του τύπου εύρεσης του epn, εξαι-

τίας της στρογγυλοποίησης. Παραθέτουμε το διάγραμμα 5.3 για δίσκο 500GB, θεωρώντας το ως γενικότερα αντιπροσωπευτικό, αλλά και ως ένα συνηθισμένο μέγεθος δίσκου.

Γενικότερα η αύξηση των μεταδεδομένων που εισάγονται από τους κόμβους των επιπέδων σε σχέση με τα καθαρά μεταδεδομένα που απαιτούνται για δεικτοδότηση των τεμαχίων, είναι περιορισμένη για 3 και κάτω επίπεδα. Συνεπώς αναμφίβολα συμφέρει η χρήση 2 και 3 επιπέδων, αν συλλογιστούμε την ακαριαία λήψη στιγμιοτύπου που προσφέρουν.

Συνολικά συμπεράσματα:

A. Με την αύξηση του μεγέθους τεμαχίου, υπάρχει εκθετική μείωση των καθαρών μεταδεδομένων που απαιτούνται για την δεικτοδότηση των τεμαχίων. Από τα 32KB και μετά το ποσοστό τους επί του δίσκου πέφτει κάτω από 0.15%

B. Η αύξηση από τα καθαρά μεταδεδομένα σε αυτά των 2 επιπέδων είναι αμελητέα, ενώ μικρή ή ανεπαίσθητη είναι και η αύξηση μέχρι και τα 3 επίπεδα.

Η επιλογή του μεγέθους τεμαχίου είναι ο καθοριστικός παράγοντας για την ποσότητα των μεταδεδομένων. Επειδή εδώ έχουμε μια σύγκρουση καθαρά στον τομέα της εξοικονόμησης χώρου, πρέπει να μετρήσουμε τα οφέλη του deduplication σε σχέση με τα επιπλέον μεταδεδομένα του μεγέθους τεμαχίου. Από την βιβλιογραφία, όμως, μπορούμε να εξάγουμε ότι για τεμάχια από 16KB και πάνω που έχουμε εστιάσει, η ποσότητα των μεταδεδομένων που εισάγεται αποτελεί ένα μικρό τίμημα μπροστά στα οφέλη της απαλοιφής διπλοτύπων. Συνεπώς θα λέγαμε ότι τα οφέλη της απαλοιφής διπλοτύπων για μικρότερο μέγεθος τεμαχίου επισκιάζουν την ζημία από τα επιπλέον μεταδεδομένα. Η επιλογή μεγαλύτερου τεμαχίου δικαιολογείται λοιπόν να γίνει κυρίως για λόγους απόδοσης, κάτι που υποστηρίζεται τόσο από τα micro-benchmarks, όσο και από το γεγονός ότι μικρότερο μέγεθος τεμαχίου συνεπάγεται μικρότερο ευρετήριο αποτυπωμάτων κατά την απαλοιφή διπλοτύπων, και άρα καλύτερη απόδοση.

Ένα ενδιαφέρον ερώτημα επίσης είναι το εάν σε βάθος χρόνου μπορούν να εξοικονομηθούν τα επιπλέον μεταδεδομένα που εισάγουν τα δέντρα με ύψη μεγαλύτερα ή ίσα του 3. Αυτή η εξοικονόμηση μπορεί να καταστεί εφικτή στην περίπτωση που υπάρχει απαλοιφή μεταξύ πλεοναστικών μεταδεδομένων μετά από μια σειρά από στιγμιότυπα. Αυτό είναι το ερώτημα που εξετάζει η επόμενη ενότητα.

5.3.2 Εξοικονόμηση μεταδεδομένων

Για να εξαλείψουμε τον εσωτερικό χρονικό πλεονασμό στα δεδομένα, χρησιμοποιούμε την τεχνική COW, η οποία μετά από την λήψη στιγμιοτύπων κατορθώνει να αποθηκεύει μόνο τις αλλαγές που έχουν συμβεί σε επίπεδο τεμαχίων. Πέρα όμως από την απαλοιφή των πλεοναστικών δεδομένων, η τεχνική αυτή σε συνδυασμό με την δενδρική μας δομή, απαλείφει και πλεοναστικά μεταδεδομένα, δηλαδή δεν αποθηκεύει ξανά τους κόμβους που δεν υπέστησαν αλλαγές μεταξύ των στιγμιοτύπων. Η ποσότητα των πλεοναστικών μεταδεδομένων που εξαλείφεται, εξαρτάται προφανώς από το μοτίβο προσπελάσεων αλλά και το μέγεθος των κόμβων, και σε αντιστοιχία με το μέγεθος τεμαχίου, όσο μικρότερος είναι ο κόμβος, τόσο μειώνονται οι πιθανότητες να μεταβληθεί μετά από ένα στιγμιότυπο και να χρειαστεί να αποθηκευτεί ξανά. Συγκεκριμένα με το να έχουμε πιο μικρούς σε μέγεθος κόμβους, καταφέρνουμε να έχουμε μια πιο fine-grained ανάλυση στα entries τους. Έτσι μπορούμε να εντοπίσουμε με μεγαλύτερη ακρίβεια τα entries που μεταβάλλονται και να αντιγράψουμε μόνο αυτά ή τουλάχιστον να αποφύγουμε να αντιγράψουμε πολλά απαράλλαχτα.
Το μέγεθος του κόμβου εξαρτάται από τον αριθμό των entries που υπάρχουν σε αυτόν, και το οποίο δίνεται από τον τύπο $epn = \lceil \sqrt[h]{tc} \rceil$. Συνεπώς, παρατηρούμε ότι για μεγαλύτερο ύψος δέντρου, έχουμε και μικρότερους κόμβους. Αυτό μας οδηγεί στο συμπέρασμα ότι η αύξηση των επιπέδων οδηγεί σε μια μείωση του μεγέθους των κόμβων, η οποία μπορεί δυνητικά να βοηθήσει στην εξοικονόμηση χώρου απαλείφοντας πλεοναστικά μεταδεδομένα μετά από την λήψη μιας ακολουθίας στιγμιοτύπων.

Για παράδειγμα για δίσκο 10GB, έχουμε στο επίπεδων κόμβων-φύλλων, για ύψος δέντρου 2, 810 κόμβους μεγέθους 17920, ενώ για ύψος 3, 7534 κόμβους μεγέθους 2048. Προφανώς, αν μεταβληθούν όλοι οι κόμβοι του επιπέδου, για ύψος 2 θα έχουμε 13.84MB, ενώ για ύψος 3 θα έχουμε 14.88MB, δηλαδή περισσότερα μεταδεδομένα. Όμως το πιθανότερο είναι ότι θα αλλάξουν κάποια από αυτά τα μεταδεδομένα. Στην ακραία περίπτωση που αλλάξουν 5 τεμάχια σε τυχαία σημεία του δίσκου, το πιθανότερο είναι ότι θα αλλάξουν 5 κόμβοι φύλλα και για τα δύο δέντρα. Έτσι τώρα τα μεταδεδομένα που θα αποθηκευτούν θα είναι για δέντρο με ύψος 2, 5*17920 = 87.5KB, ενώ για δέντρο με ύψος 3, 5*2048 + 5*2048= 20KB (προσθέσαμε και 5 διαφορετικούς πατρικούς κόμβους). Από το παραπάνω παράδειγμα γίνεται ορατός ο μηχανισμός με τον οποίο οι μικρότεροι σε μέγεθος κόμβοι καταφέρνουν να αντιμετωπίζουν την ανούσια απο-

θήκευση entries που δεν έχουν μεταβληθεί.

Μικρότερο μέγεθος κόμβων, βέβαια, δεν συνεπάγεται αυτόματα και καλύτερα αποτελέσματα. Δεν πρέπει να παραγνωρίζουμε το γεγονός ότι με την αύξηση του ύψους, το μέγεθος των κόμβων μπορεί να μικραίνει, αλλά, όπως προαναφέραμε, ο αριθμός τους αυξάνεται. Η αύξηση του αριθμού των κόμβων επιφέρει και επιπρόσθετα μεταδεδομένα, λόγω των επικεφαλίδων (NodeHeader). Παράλληλα, η αύξηση του ύψους επιφέρει και επιβάρυνση στην αποθήκευση, εξαιτίας των κόμβων των έξτρα επιπέδων σε σχέση με ένα μικρότερο ύψος. Ακόμη, μπορεί το μοτίβο των προσπελάσεων να είναι διασκορπισμένο στο δίσκο και να μην ευνοεί την απομόνωση των entries που δεν μεταβάλλονται. Συνεπώς ο καθοριστικός παράγοντας είναι η ποσότητα των μεταδεδομένων που μεταβάλλονται ανάμεσα στην λήψη των στιγμιοτύπων. Ελπίζουμε, λοιπόν, ότι καταφέρνουμε να εξοικονομούμε μεταδεδομένα, μιας που η ποσότητα του πλεονασμού που εξαλείφεται χάρη στο μεγαλύτερο επίπεδο λεπτομέρειας υπερισχύει των επιπλέον μεταδεδομένων που εισάγει το ψηλότερο δέντρο.

Πέρα από την εξοικονόμηση αποθηκευτικού χώρου, η αύξηση του αριθμού των επιπέδων μπορεί να ωφελήσει και μια διεργασία συγχρονισμού με ένα απομακρυσμένο αντίγραφο. Αυτό γιατί η ποσότητα των μεταδεδομένων κόμβων ανάμεσα στα στιγμιότυπα αποτελεί και πάλι το επιπλέον φορτίο. Ο συγχρονισμός του εικονικού δίσκου απαιτεί αποστολή των ονομάτων των entries ενός κόμβου και στη συνέχεια αποστολή μόνο των κόμβων των οποίων τα ονόματα δεν βρέθηκαν στον απομακρυσμένο εξυπηρετητή. Κάθε κόμβος λοιπόν που δεν μεταβάλλεται μεταξύ δυο στιγμιοτύπων, υπάρχει ήδη στον απομακρυσμένο εξυπηρετητή από προηγούμενο συγχρονισμό, και άρα, αφού δεν αποστέλλεται, εξοικονομεί bandwidth του δικτύου. Μεγαλύτερο ύψος δέντρου, συνεπάγεται μικρότερους κόμβους και άρα μεγαλύτερη πιθανότητα ένας κόμβος να μην μεταβληθεί και να μην χρειαστεί να αποσταλούν τα μεταδεδομένα του.

Επομένως, οι κόμβοι που δεν μεταβάλλονται μεταξύ των στιγμιοτύπων, εξοικονομούν και αποθηκευτικό χώρο και εύρος ζώνης του δικτύου κατά τον απομακρυσμένο συγχρονισμό. Καλούμαστε τώρα να εξετάσουμε το κατά πόσο τα επιπλέον επίπεδα μπορούν όντως, σε σενάρια πραγματικής χρήσης, να εξοικονομήσουν μεταδεδομένα. Σε περίπτωση που αυτό συμβαίνει θέλουμε να διαπιστώσουμε και το εάν σε βάθος χρόνου ένα υψηλότερο δέντρο μπορεί με την εξοικονόμηση μεταδεδομένων να αποσβέσει την αρχικά υψηλότερη ποσότητα των μεταδεδομένων που απαιτεί για έναν γεμάτο δί-

σκο, σε σχέση με ένα χαμηλότερο δέντρο, όπως φαίνεται στο σχήμα 5.3.

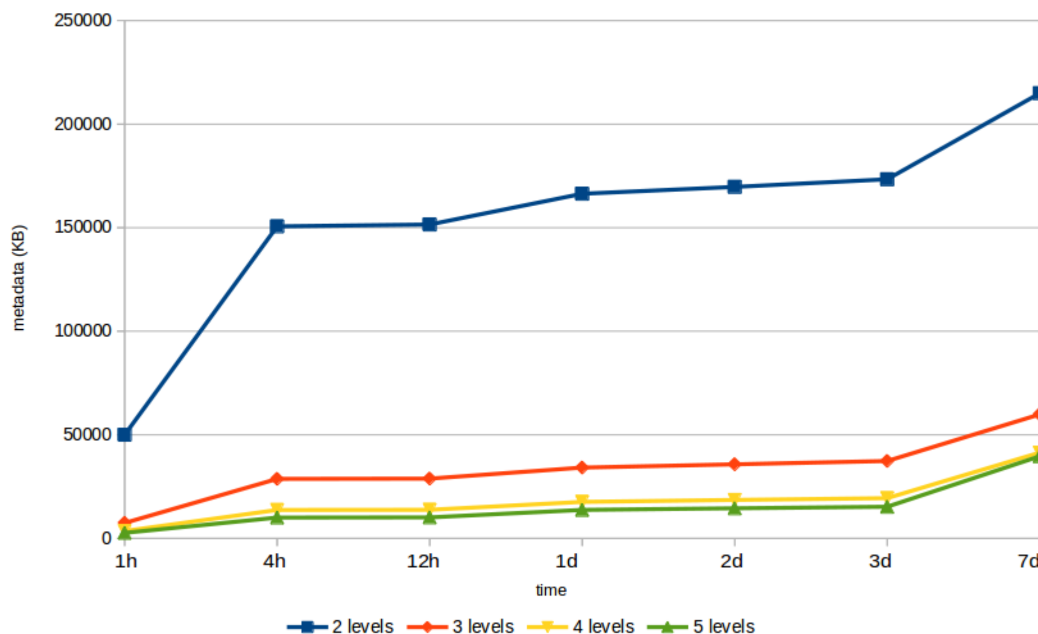
Για να εξετάσουμε τα παραπάνω, βρήκαμε ίχνη (traces) λειτουργιών Ε/Ε σε δίσκους που χρησιμοποιήθηκαν σε πραγματικό περιβάλλον παραγωγής και επομένως αντανακλούν πραγματικά φορτία και πραγματικά μοτίβα μεταβολών. Τα ίχνη αυτά ανακτήθηκαν από το SNIA IOTTA repository με ελεύθερη άδεια, και συγκεκριμένα είναι το πακέτο των MSR Caimbridge Traces που χρησιμοποιήθηκαν στο [NDR08]. Τα ίχνη αφορούν 36 volumes δεκατριών data center servers διαφορετικού προσανατολισμού, σε περιβάλλον Windows, και περιγράφουν τις λειτουργίες Ε/Ε σε χρονικό παράθυρο μιας εβδομάδας.

Για τον υπολογισμό των μεταδεδομένων που εξοικονομούνται, κατασκευάσαμε ένα πρόγραμμα που καταγράφει τους κόμβους που προσπελάζονται σε ένα ίχνος για κάθε διαφορετικό ύψος δέντρου και μέγεθος τεμαχίου και στη συνέχεια υπολογίζει με βάση αυτούς τους κόμβους την ποσότητα των μεταδεδομένων που προσπελάστηκαν. Όσο λιγότερα μεταδεδομένα προσπελάζονται, τόσο λιγότερα μεταδεδομένα σημαίνει ότι θα αποθηκεύονταν στον φυσικό δίσκο ή θα αποστέλλονταν στο δίκτυο, εάν αυτό το ίχνος των λειτουργιών Ε/Ε ακολουθούσε μετά από μια λήψη στιγμιοτύπου.

Αρχικά παρατηρούμε την διαφορά στην απαλοιφή πλεονασμού ανάλογα με το χρονικό διάστημα που μεσολαβεί μεταξύ πιθανών λήψεων στιγμιοτύπου. Χρησιμοποιούμε τα ίχνη από ένα δίσκο δεδομένων χρηστών, ένα δίσκο δεδομένων ενός web/sql server και ένα δίσκο όπου υπάρχει το λειτουργικό σύστημα του χρήστη.

Προφανώς η μορφή του διαγράμματος θα εξαρτηθεί και από την ποσότητα των δεδομένων που μεταβάλλονται. Παραθέτουμε τα αποτελέσματα μόνο για μέγεθος τεμαχίου 16KB, αφού για τα υπόλοιπα μεγέθη τεμαχίου η μορφή του διαγράμματος ήταν η ίδια και άρα οι αλληλοσυσχετίσεις ανάμεσα στα διαφορετικά επίπεδα και ανάμεσα στα διαφορετικά χρονικά διαστήματα ήταν οι ίδιες. Αυτό που κυρίως άλλαζε ήταν η ποσότητα των μεταδεδομένων που μεταβάλλονταν, κάτι που είναι λογικό, αφού έχουμε διαφορετικές ανάγκες σε δεικτοδότηση τεμαχίων.

Βλέπουμε ότι από πολύ νωρίς, δηλαδή ανάμεσα στη μία ώρα και στις 4 ώρες, δημιουργείται μια μεγάλη διαφορά ανάμεσα στην ποσότητα των μεταδεδομένων που αλλάζουν. Το εντυπωσιακό είναι ότι αυτό δεν οφείλεται σε μια μεγάλη ποσότητα δεδομένων που άλλαξαν. Συνεπώς μπορούμε να συμπεράνουμε ότι επειδή αυτό το μοτίβο ήταν και εντονότερο στον δίσκο δεδομένων των χρηστών παρά στον web server,



Σχήμα 5.4: *Μεταβολή μεταδεδομένων με την πάροδο του χρόνου*

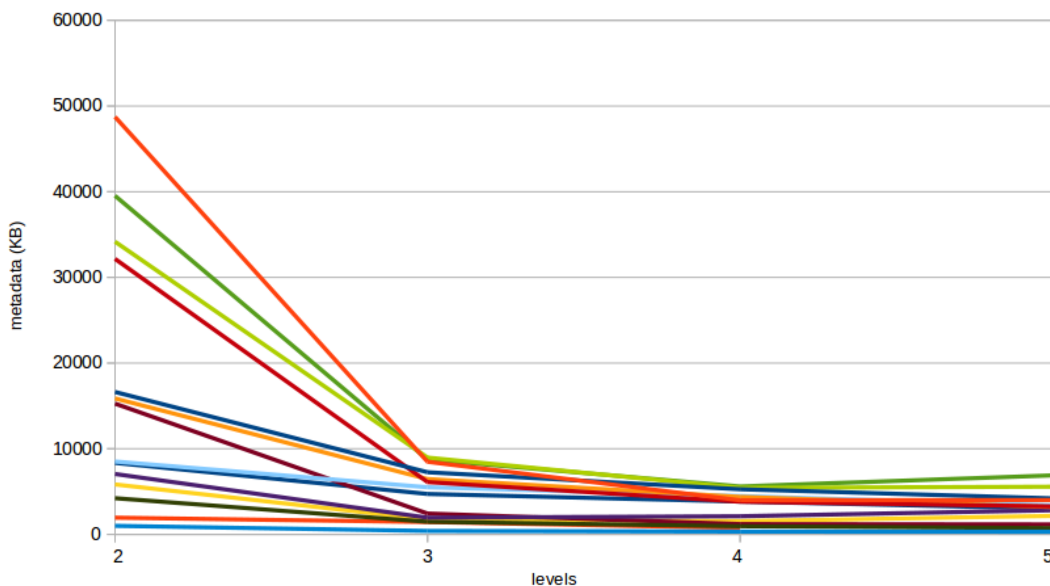
υπάρχει μια περιοχή του δίσκου η οποία υφίσταται επαναληπτικά μεταβολές. Γι' αυτό το λόγο αυξήθηκαν νωρίς τα μεταδεδομένα, αλλά στη συνέχεια παρέμεναν σταθερά, παρόλο που η ποσότητα των δεδομένων που εγγράφονταν αυξάνονταν με σταθερό ρυθμό. Το παραπάνω καταδεικνύει ότι οι μεταβολές στην ποσότητα των μεταδεδομένων ανάμεσα στα στιγμιότυπα, εξαρτώνται μόνο από το χωρικό παράθυρο.

Τα διαγράμματα των άλλων δίσκων έχουν παρόμοια μορφή, αλλά με λίγο ηπιότερη μεταβολή ανάμεσα στη μια και τις 4 ώρες. Ο χρόνος δεν είναι σε γραμμική κλίμακα και είναι λογικό στο τέλος να έχουμε μεγάλη μεταβολή λόγω του μεγάλου χρονικού διαστήματος των 4 ημερών που μεσολαβεί. Τα διάφορα διαστήματα καταδεικνύουν την ποσότητα των μεταδεδομένων μόνο για τους κόμβους, ενώ θεωρείται ότι στιγμιότυπο λαμβάνεται από την αρχή των ιχνών και όχι από την μια χρονική στιγμή στην άλλη.

Τώρα θεωρούμε ότι το μεσοδιάστημα μεταξύ λήψης στιγμιοτύπου είναι μια εβδομάδα και χρησιμοποιούμε τα ίχνη πολλών διαφορετικών servers για να μελετήσουμε την επίδραση του μεγέθους τεμαχίου και του ύψους του δέντρου.

Όσο αφορά το ύψος, από το προηγούμενο διάγραμμα έγινε ήδη αντιληπτή η μεγάλη διαφοροποίηση ανάμεσα στα 2 και στα 3 επίπεδα, χωρίς να παρουσιάζονται εντυπωσιακές ευκαιρίες εξοικονόμησης μεταδεδομένων από τα 3 και πάνω. Εδώ παραθέτουμε

ένα διάγραμμα για 14 διαφορετικούς δίσκους. Όπως βλέπουμε, σε όλων των ειδών servers, παρατηρούμε μια μεγάλη μεταβολή από τα 2 στα 3 επίπεδα και από εκεί και πάνω τα κέρδη μοιάζουν μικρά ή έστω αμελητέα. Σε κάποιες περιπτώσεις χρήσης όπως τους project servers, web staging servers, user servers και media servers, όπου έχουμε εγγραφές σε συγκεκριμένα χωρία του δίσκου, η χρήση τριών επιπέδων είναι ακόμα πιο ωφέλιμη. Οι μοναδικές περιπτώσεις που δεν εμφανίζεται η γνησίως φθίνουσα ποσότητα μεταδεδομένων είναι όταν τα 5 επίπεδα αποδεικνύονται χειρότερα από τα 4.



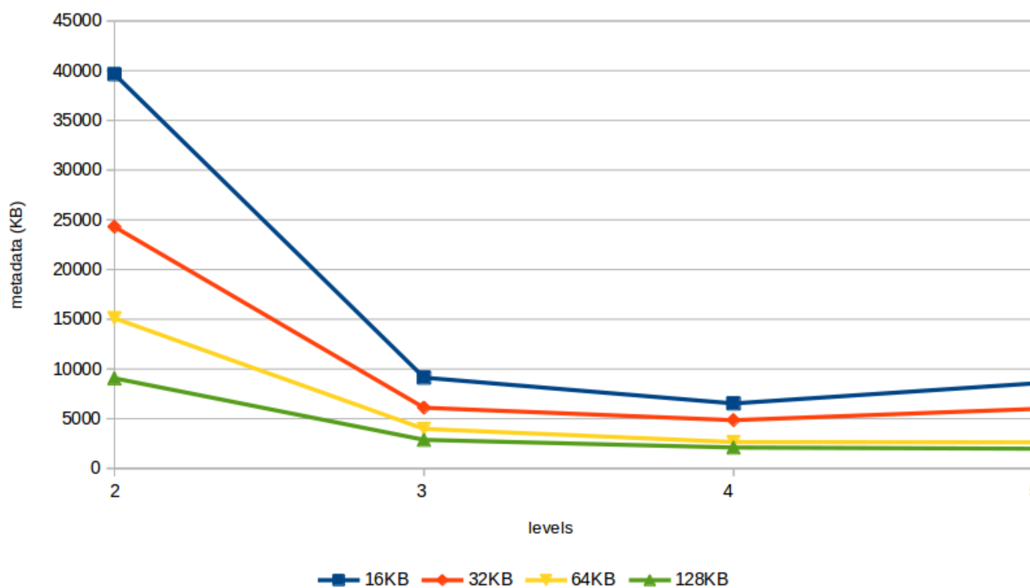
Σχήμα 5.5: *Μεταβολή μεταδεδομένων για διαφορετικές περιπτώσεις δίσκων*

Γενικώς παρατηρήσαμε ότι όσο αυξάνεται το μέγεθος τεμαχίου, τόσο μειώνονται και τα οφέλη που έχουμε σε εξοικονόμηση μεταδεδομένων. Συνεπώς ειδικά για μικρά μεγέθη, όπου τα μεταδεδομένα είναι περισσότερα, καθίστανται ακόμα πιο αναγκαία η χρήση τριών επιπέδων έτσι ώστε να μην έχουμε έκρηξη της ποσότητας των μεταδεδομένων σε περίπτωση συχνών στιγμιοτύπων. Αυτό δυστυχώς έρχεται σε αντίθεση με τις παρατηρήσεις του προηγούμενου κεφαλαίου, όπου η απόδοση του συνδυασμού 16KB και 3 επιπέδων είναι σημαντικά χειρότερη από αυτή των 2.

Επίσης παρατηρούμε ότι στους δίσκους που περιέχουν μόνο δεδομένα και όχι λειτουργικά συστήματα, τότε η επιλογή υψηλότερου επιπέδου είναι πιο σημαντική από την επιλογή υψηλότερου μεγέθους τεμαχίου για την εξοικονόμηση μεταδεδομένων σε απόλυτους αριθμούς. Αυτό είναι λογικό, δεδομένου ότι ανάμεσα σε στιγμιότυπα δεδομένων, έχουμε μεγαλύτερη τοπικότητα στις αλλαγές και πολύ συγκεντρωμένα μοτίβα προσπέλασης του δίσκου. Αυτό εκτός από την εξοικονόμηση μεταδεδομένων

ευνοεί και την εξοικονόμηση των τεμαχίων. Ένα τέτοιο διάγραμμα παρουσιάζεται στο σχήμα 5.6, ενώ παρόμοια είναι και τα διαγράμματα άλλων servers.

Στους δίσκους που περιέχουν λειτουργικά συστήματα είναι λογικό οι προσπελάσεις να είναι πιο διασκορπισμένες, οπότε το μέγεθος τεμαχίων παίζει μεγαλύτερο ρόλο στην ποσότητα των μεταδεδομένων που τελικά θα αποθηκευτούν σε απόλυτους αριθμούς.



Σχήμα 5.6: *Μεταβολή μεταδεδομένων για διαφορετικά μεγέθη τεμαχίων και επίπεδα*

Τέλος να σημειώσουμε ότι σε απόλυτους αριθμούς και η ποσότητα των μεταδεδομένων που απαιτείται για χαμηλό μέγεθος τεμαχίου είναι για 2 επίπεδα σημαντικά μεγαλύτερη από αυτήν για μεγαλύτερα τεμάχια.

Συνεπώς καταλήγουμε στο ότι για την απαλοιφή πλεονασμού

Α. Είναι πολύ προτιμότερη η χρήση 3 επιπέδων παρά 2, για όλα τα σενάρια χρήσης δίσκων. Από εκεί και πέρα τα κέρδη είναι μικρά ή ανύπαρκτα.

Β. Τα οφέλη 3 αντί για 2 επιπέδων είναι μεγαλύτερα για μικρά μεγέθη τεμαχίου παρά για μεγάλα. Δυστυχώς αυτό έρχεται σε αντίθεση με την απόδοση των μικρών τεμαχίων για 3 επίπεδα.

Γ. Για μεταβολές σε δίσκους που περιέχουν δεδομένα και όχι OS, η επιλογή 3 αντί για 2 επιπέδων είναι σημαντικότερη από την επιλογή του μεγέθους τεμαχίου για απαλοιφή πλεοναστικών μεταδεδομένων.

Δ. Οι αλλαγές των μεταδεδομένων φαίνεται να προκύπτουν νωρίς, ανάμεσα στη 1

και 4 ώρες, και μετά να παραμένουν σταθερές, χωρίς να ακολουθούν την ποσότητα των δεδομένων που γράφονται. Αυτό οφείλεται στο ότι δεν αλλάζει το χωρικό μοτίβο εγγραφής στον δίσκο.

Όλα τα παραπάνω παρατηρήθηκαν χωρίς τον συνυπολογισμό των φυσικών μεταδεδομένων, ο οποίος στην υλοποίησή μας είναι 304bytes ανά κόμβο ή στην γενική περίπτωση 30bytes ανά κόμβο. Αυτό έγινε ώστε να διαχωρίσουμε την επίδρασή τους, μιας και ναι μεν παίζει ρόλο στο χώρο που καταλαμβάνουν τα μεταδεδομένα κόμβων στο δίσκο, αλλά δεν παίζει ρόλο στην εξοικονόμηση bandwidth, αφού δεν στέλνονται τα φυσικά μεταδεδομένα αλλά μόνο τα λογικά.

Εκτελέσαμε και πάλι τις μετρήσεις του τελευταίου κομματιού, συνυπολογίζοντας τώρα για κάθε κόμβο που αλλάζει μια επιπλέον επιβάρυνση στην αποθήκευσή του. Εκτελέσαμε πειράματα και με τα δύο μοντέλα, τόσο με επιβάρυνση 304bytes της υλοποίησής μας όσο και με επιβάρυνση 30bytes που θα είχαμε με τον σχεδιασμό μας σε μια άλλη υλοποίηση. Περιμένουμε ότι ίσως να εξανεμίζονται τα οφέλη των τριών επιπέδων σε σχέση τα δύο επίπεδα, αλλά αντιθέτως παρατηρούμε ότι κάτι τέτοιο δεν συμβαίνει, αφού μπορεί να αμβλύνεται η διαφορά σε σχέση με πριν, αλλά και πάλι, ειδικά στους δίσκους δεδομένων, τα 3 επίπεδα επιφέρουν σημαντική βελτίωση. Σε δίσκους με λειτουργικά, παρατηρούμε ότι πέραν της μικρότερης διαφοράς ανάμεσα στα 2 και 3, τώρα ότι η απόδοση χειροτερεύει κάποιες φορές για 4 και 5 επίπεδα. Συνεπώς συμπληρώνοντας το ανωτέρω συμπέρασμα A, καταλήγουμε ότι η συνιστώμενη επιλογή για απαλοιφή μεταδεδομένων κόμβων είναι τα 3 επίπεδα.

E. Ακόμα και συνυπολογίζοντας φυσικά μεταδεδομένα των 304bytes, τα 3 επίπεδα παραμένουν με συνέπεια καλύτερα από τα 2 και είναι σε πολλές περιπτώσεις ανώτερα και από τα 4 και 5, οπότε αποτελούν την βέλτιστη τιμή για απαλοιφή μεταδεδομένων κόμβων.

Το μόνο που μένει να διαπιστώσουμε είναι εάν αυτά τα μεταδεδομένα που εισάγονται από τα επιπλέον επίπεδα θα εξοικονομηθούν σε βάθος χρόνου, από τα οφέλη της απαλοιφής πλεοναστικών μεταδεδομένων σε συνεχόμενα στιγμιότυπα. Δηλαδή μετά από πόσα στιγμιότυπα ένα δέντρο ύψους x θα καταφέρει να αποσβέσει τα επιπλέον μεταδεδομένα κόμβων, σε σχέση με ένα ύψους $x - 1$. Να υπενθυμίσουμε ότι στην αρχή το δέντρο ύψους x θα καταλαμβάνει περισσότερο χώρο από το δέντρου ύψους $x - 1$, γιατί για ίδιο μέγεθος γεμάτου δίσκου, θα χρειάζονται περισσότερα με-

ταδεδομένα. Όσο όμως λαμβάνονται στιγμιότυπα, όπως είδαμε, το δέντρο με ύψος x θα αποθηκεύει τα στιγμιότυπα πιο αποδοτικά και άρα σιγά σιγά θα αποσβένει αυτήν την αρχική απόκλιση.

Γι' αυτό, θα υπολογίσουμε τον αριθμό των στιγμιοτύπων από τον οποίο και μετά ένα ψηλότερο δέντρο θα δαπανά συνολικά λιγότερο χώρο για αποθήκευση μεταδεδομένων. Στον υπολογισμό των μεταδεδομένων κόμβων συνυπολογίζουμε και το λογικό και το φυσικό τους κομμάτι (με 304bytes). Πραγματοποιούμε υπολογισμούς για 7 δίσκους και δεν διαχωρίζουμε τα αποτελέσματα ανάλογα με το μέγεθος τεμαχίου, παρόλο που για μικρά μεγέθη τεμαχίου, η λειτουργία της απόσβεσης θα χρειαστεί περισσότερα στιγμιότυπα λόγω των περισσότερων αρχικών μεταδεδομένων. Έτσι μελετάμε συνολικά 28 συνδυασμούς δίσκων και μεγεθών τεμαχίου.

Τα αποτελέσματα συνηγορούν συντριπτικά υπέρ του επιπέδου 3, αφού για όλους τους συνδυασμούς που δοκιμάστηκαν, η απόσβεση πραγματοποιούνταν πάντα και ήθελε από 1 έως και 9 το πολύ στιγμιότυπα, με μέση τιμή 2.5. Αντίθετα για να αποσβεστούν τα επιπλέον μεταδεδομένα από το 4ο στο 3ο χρειάζονται κατά μέσο όρο 31 στιγμιότυπα, ενώ σε 7 από τους 28 συνδυασμούς (25%) δεν κατέστη δυνατή αυτή η απόσβεση, αφού τα 3 επίπεδα είχαν καλύτερη απόδοση σε βάθος χρόνου. Για απόσβεση από το 5ο στο 4ο επίπεδο δε, θέλουμε κατά μέσο όρο 93 στιγμιότυπα, και πάλι στο 25% των συνδυασμών η απόσβεση δεν είναι εφικτή. Τέλος, για απόσβεση από τα 4 στα 2 επίπεδα χρειάζονται κατά μέσο όρο 9.7 στιγμιότυπα και η απόσβεση είναι πάντα εφικτή.

Τα παραπάνω νούμερα μειώνονται ακόμη περισσότερο εάν δεν συνυπολογίσουμε την επιβάρυνση στα φυσικά μεταδεδομένα, με την απόσβεση να πραγματοποιείται κατά μέσο όρο από τα 3 στα 2 επίπεδα για 1 στιγμιότυπο και από τα 4 στα 3 για 5.4 στιγμιότυπα. Έχουμε λοιπόν τα προαναφερθέντα άνω και κάτω φράγματα για τον αριθμό στιγμιοτύπων που κάνουν εφικτή την απόσβεση, δημιουργώντας μια περιοχή κερδών που εξαρτάται από την επιβάρυνση σε φυσικά μεταδεδομένα.

Καταλήγουμε λοιπόν στο ότι η χρήση τριών ή και τεσσάρων επιπέδων μπορεί εύκολα να εξοικονομήσει σε βάθος χρόνου τα επιπλέον μεταδεδομένα κόμβων που αυτά τα επίπεδα αρχικά εισάγουν.

Να τονίσουμε ότι τα ίχνη των λειτουργιών Ε/Ε αναφέρονται σε μια ποικιλία μεγεθών δίσκου από 20GB έως και 500GB, οπότε τα αποτελέσματά μας είναι αντιπροσωπευτικά κάθε μεγέθους.

Ανακεφαλαίωση

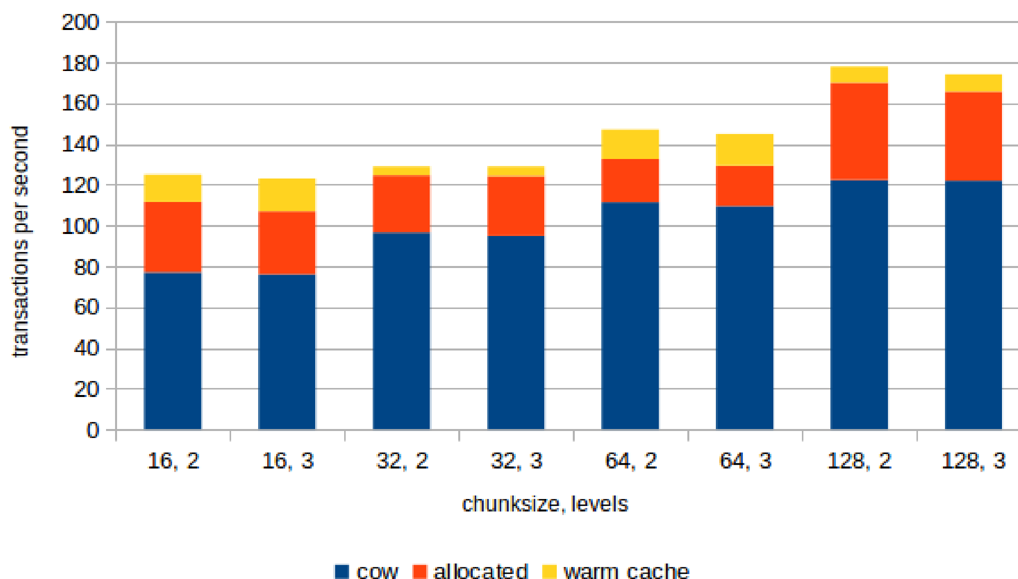
Ανακεφαλαιώνοντας τις παρατηρήσεις μας σε σχέση με τα μεταδεδομένα, καταλήγουμε ότι η ποσότητα των μεταδεδομένων που εισάγουν τα επιπλέον δεντρικά επίπεδα είναι αμελητέα, δεδομένου μάλιστα ότι σε πολύ μικρό βάθος χρόνου μπορεί να εξοικονομηθεί. Η επιλογή 3 αντί για 2 επιπέδων βοηθάει σημαντικά στην εξοικονόμησή τους κατά την λήψη στιγμιοτύπων, ενώ εξοικονομείται και bandwidth δικτύου όταν έχουμε ένα πρόγραμμα συγχρονισμού απομακρυσμένων αντιγράφων. Το χαμηλό μέγεθος τεμαχίου, είναι σημαντική παράμετρος για την απαλοιφή διπλοτύπων, ενώ επηρεάζει με εκθετικό τρόπο τα καθαρά μεταδεδομένα. Δεδομένου ότι έχουμε αποφύγει πολύ μικρά μεγέθη (<16KB) και ότι θα χρησιμοποιηθούν 2 ή 3 επίπεδα για να αποφευχθεί η έκρηξη μεταδεδομένων που ίσως να επέφερε μια σειρά από στιγμιότυπα σε μη δενρική δομή, καταλήγουμε ότι πιθανότατα θα πάρουμε πίσω πολλαπλώς τα οφέλη στα έξτρα μεταδεδομένα από την απαλοιφή διπλοτύπων.

Στην επόμενη ενότητα θα εστιάσουμε στους συσχετισμούς απόδοσης για τα 4 διαφορετικά μεγέθη τεμαχίων και στο κατά πόσο η αυτή μειώνεται αισθητά ή όχι από τα 2 στα 3 επίπεδα, μελετώντας αυτή τη φορά σενάρια πραγματικής χρήσης.

5.4 Macro-Benchmarks

Θα δοκιμάσουμε τα παραπάνω σύνολα παραμέτρων σε μακροσκοπικά φορτία που προσομοιώνουν μοντέλα πραγματικής χρήσης του εικονικού δίσκου μας.

Αρχικά χρησιμοποιούμε το Postmark benchmark (έκδοση 1.5.0) [Kat97] το οποίο προσομοιώνει την λειτουργία ενός web ή mail server, εκτελώντας συναλλαγές πάνω σε πολλά μικρά αρχεία. Για τις ανάγκες του τεστ μας εκτελέσαμε 50.000 συναλλαγές σε 1000 αρχεία μεγέθους από 5KB-512KB, λαμβάνοντας μέσο όρο των μετρήσεών μας. Το τεστ αυτό, δημιουργούσε αρχεία, τα διάβαζε, πρόσθετε δεδομένα στο τέλος τους και τα διέγραφε κατ’ αυτήν την σειρά. Εκτελέσαμε τα τεστ έχοντας δεσμεύσει τα τεμάχια και τους κόμβους που προσπελάζονται, δηλαδή έχουμε την allocated περίπτωση. Επίσης τρέξαμε τα τεστ και μετά τη λήψη ενός στιγμιοτύπου (cow περίπτωση), αλλά και χωρίς να αδειάσουμε τις page cache, inode cache, dentry cache του guest και του host (warm cache περίπτωση). Χρησιμοποιήσαμε μια single threaded προσέγγιση.



Σχήμα 5.7: *Αποτελέσματα postmark benchmark (web, mail server). Κάθε μπάρα αντιπροσωπεύει ένα ζευγάρι {chunksize, ύψος δέντρου} και το σημείο που τελειώνει η μπλε περιοχή σηματοδοτεί την απόδοση της cow περίπτωσης, το σημείο που τελειώνει η πορτοκαλί περιοχή σηματοδοτεί την απόδοση της allocated περίπτωσης και το σημείο που τελειώνει η κίτρινη περιοχή σηματοδοτεί την απόδοση της warm cache περίπτωσης.*

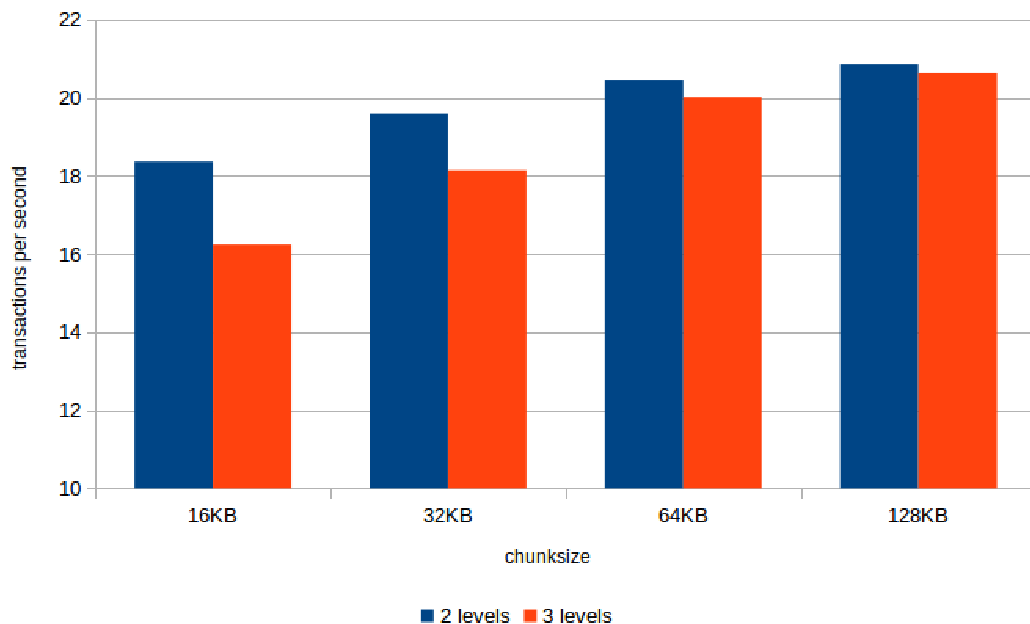
Σύμφωνα με το διάγραμμα 5.7, παρατηρούμε ότι ο κύριος παράγοντας που επηρεάζει την απόδοση δεν είναι το επίπεδο αλλά το μέγεθος των τεμαχίων. Παρατηρούμε ότι και στις 3 περιπτώσεις, η διαφορά στις συναλλαγές ανά δευτερόλεπτο είναι πολύ μικρή από τα 2 στα 3 επίπεδα, ενώ αντιθέτως η ψαλίδα ανοίγει με την αύξηση του μεγέθους τεμαχίου.

Στην COW περίπτωση, οι διαφορές στην απόδοση αμβλύνονται όσο αυξάνεται το μέγεθος τεμαχίου, καθώς ενώ από 16KB σε 32KB υπάρχει διαφορά περίπου 20 συναλλαγών το δευτερόλεπτο, από 64KB σε 128KB υπάρχει διαφορά μόνο περίπου 10 συναλλαγών το δευτερόλεπτο. Συνεπώς, εδώ η COW περίπτωση επωφελείται από μεγάλα μεγέθη τεμαχίων.

Αντιθέτως, στην allocated περίπτωση παρατηρούμε ότι η απόδοση εκτοξεύεται από τα 64KB στα 128KB, καθώς ενώ από τα 16KB στα 32KB έχουμε αύξηση 11% και από τα 32KB στα 64KB έχουμε αύξηση 7%, από τα 64KB στα 128KB έχουμε αύξηση 30%.

Στη συνέχεια χρησιμοποιούμε το Pgbench benchmark (έκδοση 9.5) [pgb], το οποίο ελέγχει την απόδοση επί διαφόρων λειτουργιών των βάσεων δεδομένων και συγκεκριμένα λειτουργιών επί του PostgreSQL συστήματος βάσης δεδομένων. Χρησιμο-

ποιήσαμε την scale επιλογή με παράμετρο 300 για τους δοκιμαστικούς πίνακες της βάσης, έτσι ώστε τα δεδομένα να είναι αρκούντως πολλά (5.4GB) για να πυροδοτήσουν αρκετές προσπελάσεις στο δίσκο και άρα να επιτρέψουν την αποτύπωση των διαφορών ανάλογα με την μορφή του εικονικού δίσκου. Εκτελέσαμε για κάθε παραμετροποίηση 3 επαναλήψεις των 20 λεπτών, ώστε να εξάγουμε σταθερά και αξιόπιστα αποτελέσματα. Επίσης δεσμεύσαμε εκ των προτέρων τα τεμάχια και τους κόμβους που προσπελάζονται, δηλαδή έχουμε την allocated περίπτωση και χρησιμοποιήσαμε μια single threaded προσέγγιση.



Σχήμα 5.8: *Αποτελέσματα pgbench benchmark (βάση δεδεομένων)*

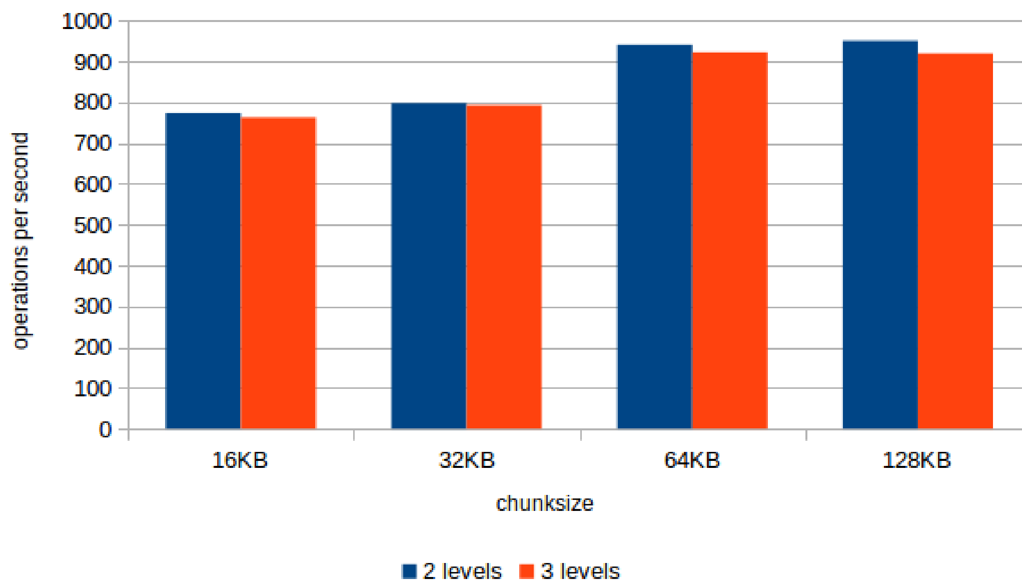
Από το διάγραμμα 5.8 παρατηρούμε ότι για μεγάλα μεγέθη τεμαχίων (64KB, 128KB) η απόδοση είναι παραπλήσια και ελάχιστα διαφοροποιείται από τα 2 στα 3 επίπεδα. Αντιθέτως για μικρά μεγέθη τεμαχίων (16KB, 32KB), υπάρχει μεγάλο χάσμα απόδοσης ανάμεσα στα διαφορετικά ύψη των δέντρων, κάτι που και στην ενότητα των micro-benchmarks παρατηρήσαμε ότι ισχύει για τυχαία μοτίβα προσπέλασης. Δυστυχώς λοιπόν, παρόλο που τα τρία επίπεδα προσφέρουν μεγαλύτερες ευκαιρίες εξοικονόμησης μεταδεδομένων για χαμηλά μεγέθη τεμαχίων, υπάρχει μη αμελητέα πτώση απόδοσης. Να προσθέσουμε ακόμη ότι για δίσκους 2 επιπέδων, η απόδοση των 32KB, 64KB και 128KB είναι αρκετά κοντά, με μείωση να παρουσιάζεται κυρίως στα 16KB.

Παράλληλα, ενδιαφέροντα είναι και τα αποτελέσματα που συλλέξαμε από την αρχικοποίηση των πινάκων της βάσης δεδομένων. Από τον παρακάτω πίνακα μπορούμε

να διαπιστώσουμε την θεαματική υπεροχή των 64KB και 128KB που διαφέρουν μόνο λίγο μεταξύ τους. Και εδώ κυριαρχεί το μέγεθος τεμαχίου και όχι το επίπεδο, εξαιρουμένων των τεμαχίων 16KB, τα οποία ούτως ή άλλως είναι αρκετά υποδεέστερα.

	2 επίπεδα	3 επίπεδα
16KB	7m 52s	10m 2s
32KB	5m 45s	5m 59s
64KB	4m 37s	4m 36s
128KB	4m 23s	4m 33s

Χρησιμοποιήσαμε επίσης το filebench benchmark (έκδοση 1.4.9.1) [TZS16], ώστε να προσομοιωθεί η συμπεριφορά ενός fileserver με δημιουργία, διαγραφή και επεξεργασία αρχείων από 50 νήματα. Διατηρήθηκαν οι προεπιλεγμένες τιμές των παραμέτρων για την προκαθορισμένη «fileserver προσωπικότητα». Εδώ λοιπόν μπορούμε να παρατηρήσουμε και την πολυνηματική συμπεριφορά του σχεδιασμού μας.



Σχήμα 5.9: *Αποτελέσματα filebench benchmark (fileserver)*

Τα αποτελέσματα που προκύπτουν (διάγραμμα 5.9) φανερώνουν ότι τα μεγάλα μεγέθη τεμαχίων (64KB, 128KB) έχουν σημαντικά καλύτερη απόδοση από τα μικρά μεγέθη (16KB, 32KB), αλλά οι διαφορές σε κάθε ομάδα είναι μικρές. Παράλληλα, παρατηρούμε ότι η απόδοση των τριών επιπέδων είναι ελάχιστα μόνο χαμηλότερη από αυτή των δύο επιπέδων, αποτέλεσμα συνηγορεί υπέρ της χρήσης των τριών επιπέδων για λόγους εξοικονόμησης χώρου.

Γενικότερα, μπορούμε να αποδώσουμε την μικρή διαφοροποίηση ανάμεσα στα 2 και στα 3 επίπεδα -όπου αυτή υπάρχει-, στο γεγονός ότι πολλοί από τους επιπλέον ενδιάμεσους κόμβους των τριών επιπέδων τοποθετούνται στην host page cache μετά την πρώτη προσπέλαση, και με αυτόν τον τρόπο μετριάζεται η επιβάρυνση που εισάγουν αφού μειώνεται ο αριθμός των απαιτούμενων λειτουργιών Ε/Ε. Παρόλο που γι' αυτό το λόγο δεν παρατηρούμε τις διαφορές που παρατηρήσαμε στα micro-benchmarks, τα macro-benchmakrs προσομοιώνουν μακροσκοπικά φορτία πραγματικών συστημάτων, οπότε μπορούμε να εμπιστευτούμε ότι η διαφορά απόδοσης ανάμεσα στα 2 και στα 3 επίπεδα θα είναι όντως μικρή σε εικονικούς δίσκους πραγματικής χρήσης.

Τέλος, για να ολοκληρώσουμε την μακροσκοπική ανάλυσή μας αναφέρουμε το χρόνο περάτωσης τριών θεμελιωδών λειτουργιών. Τα δεδομένα αυτά δεν είναι χρήσιμα μόνο για την σύγκριση των διαφόρων παραμετροποιήσεων, αλλά όντας αναπόσπαστες λειτουργίες στην πραγματική χρήση ενός εικονικού δίσκου, βοηθούν στην διαισθητική εκτίμηση σε απόλυτα μεγέθη, των διαφορών καθυστέρησης, για διαφορετικά ύψη και τεμάχια.

Η δημιουργία ενός συστήματος αρχείων ext4 μεγέθους 10GB στον guest (mkfs.ext4), εξαρτάται γραμμικά από το μέγεθος του τεμαχίου (για ύψος 2: 44s, 22s, 10s, 6s), με τον επιπλέον χρόνο στα 3 επίπεδα να μην ξεπερνάει το 10% του χρόνου των 2 επιπέδων. Αυτά τα αποτελέσματα είναι αναμενόμενα, αφού εμπλέκονται σειριακές και όχι τυχαίες αιτήσεις, με μέγεθος μεγαλύτερο των 128KB, που σκοπό έχουν την δέσμευση μπλοκ για superblock, block descriptors και journal. Έτσι έχουμε άμεση εξάρτηση από τον αριθμό των τεμαχίων που προσπελάσονται, με τις επιπλέον λειτουργίες λόγω προσπέλασης κόμβων να είναι αρκετά λιγότερες, όπως περιγράψαμε και στην 3.8.

Στην συνέχεια παραθέτουμε και άλλους δυο πίνακες: έναν για την πρωθύστερη δέσμευση όλων των μεταδεδομένων ενός 10GB δίσκου (preallocation), που αναμένεται να είναι ανάλογος με το διάγραμμα 3.11, και έναν για την διαγραφή όλων των μεταδεδομένων ενός 10GB δίσκου, που όπως φαίνεται εξαρτάται σε μεγαλύτερο βαθμό από τον αριθμό των αρχείων. Και πάλι παρατηρούμε την έντονη υστέρηση των 16KB, ειδικά στην διαγραφή, ενώ τώρα για 16KB και 32KB, ο χρόνος διαγραφής παρουσιάζει αξιοσημείωτη απόκλιση για τα διαφορετικά ύψη. Σε αυτές τις δύο λειτουργίες, τα 128KB οδηγούν σε σημαντικά ταχύτερη διεκπεραίωση.

preallocation	2 επίπεδα	3 επίπεδα
16KB	30.27s	36.95s
32KB	17.48s	19.98s
64KB	12.48s	14.64s
128KB	1.31s	1.5s

διαγραφή	2 επίπεδα	3 επίπεδα
16KB	148.00s	198.00s
32KB	36.83s	45.48s
64KB	7.58s	8.10s
128KB	2.90s	3.20s

6 Συμπεράσματα και μελλοντικές δυνατότητες

Η δενδρική συνταγή που περιγράψαμε μας επιτρέπει να διατηρήσουμε μικρό το μέγεθος τεμαχίου, χωρίς να έχουμε έκρηξη μεταδεδομένων και καθυστέρηση στην λήψη στιγμιοτύπου. Συνδυάζοντας τις παρατηρήσεις που έχουμε από τα micro-benchmarks, η δύναμη της δεντρικής δομής είναι προφανής, αφού τα 2 επίπεδα, τόσο από πλευράς απόδοσης, όσο και από πλευράς επιπλέον μεταδεδομένων εισάγουν μια απειροελάχιστη επιβάρυνση, παρέχοντας παράλληλα δυνατότητα για ακαριαία λήψη στιγμιοτύπου και για σημαντική εξοικονόμηση χώρου για μια σειρά από στιγμιότυπα. Αυτή η εξοικονόμηση μπορεί να είναι ακόμα σημαντικότερη, υποβοηθώντας ένα πρόγραμμα συγχρονισμού και υποσκελίζοντας τα αρχικά επιπλέον μεταδεδομένα, εάν αυξήσουμε το επίπεδο λεπτομέρειας και χρησιμοποιήσουμε 3 επίπεδα. Το γεγονός αυτό, σε συνδυασμό με την αμελητέα ή αποδεκτή πτώση της απόδοσης σε σχέση με τα δύο επίπεδα, όπως επιβεβαιώνουν και τα macro-benchmarks, συνηγορεί έντονα στην επιλογή δέντρου με ύψος 3. Μεγαλύτερα ύψη, επιβαρύνουν αισθητά την απόδοση, προσφέροντας χαμηλού ρυθμού ή καθόλου εξοικονόμηση μεταδεδομένων.

Όσο αφορά την επιλογή του μεγέθους τεμαχίου, θα λέγαμε ότι για μεγέθη μεγαλύτερα από 16KB, δεν πρέπει να λάβουμε υπ' όψιν μας την επιβάρυνση σε μεταδεδομένα, αφού η απαλοιφή διπλοτύπων θα μας επιστρέψει τα οφέλη πολλαπλάσια. Το κυριότερο κριτήριο είναι το tradeoff ανάμεσα στην απόδοση και στην εξοικονόμηση χώρου που επιθυμούμε να πετύχουμε, αφού για μικρά μεγέθη τεμαχίου έχουμε σταθερά χαμηλότερη απόδοση, η οποία κυμαίνεται ανάλογα με το είδος της λειτουργίας Ε/Ε

και την κατάσταση του δίσκου. Αποκλείουμε μεγέθη πάνω από τα 128KB, αφού δεν προσφέρουν με συνέπεια ταχύτερα αποτελέσματα, ενώ παράλληλα δυσχεραίνουν την αποδοτική απαλοιφή διπλοτύπων. Επίσης αποκλείουμε και μεγέθη κάτω από τα 16KB γιατί πρώτον, αν και πιθανότητατα θα αυξήσουν το deduplication ratio, εισάγουν μια μη αμελητέα ποσότητα μεταδεδομένων που ίσως και να υπερκαλύψει τα κέρδη σε πλεονασμό, και δεύτερον και κυριότερο, γιατί μειώνουν δραματικά την απόδοση, τόσο στις λειτουργίες επί του εικονικού δίσκου, όσο και στις λειτουργίες αναζήτησης στο ευρετήριο αποτυπωμάτων.

Εξειδικεύοντας τώρα στην περίπτωση των επιπέδων 2-3, η απόδοση των 16KB είναι στις περισσότερες περιπτώσεις αρκετά χειρότερη από αυτήν για 32KB-128KB και εντυπωσιακά χειρότερη στην περίπτωση πρώτης εγγραφής σε κενό δίσκο. Επίσης η απόκλιση μεταξύ των 64KB και 128KB είναι στις περισσότερες λειτουργίες Ε/Ε σχετικά μικρή, ιδίως σε ότι αφορά το latency.

Συνεπώς, καταλήγουμε στο ότι ένα δέντρο με ύψος 3 και μέγεθος τεμαχίου 64KB, αποτελεί μια επιλογή που φαίνεται να επιτυγχάνει μια επιθυμητή ισορροπία ανάμεσα στην υψηλή απόδοση και στην εξοικονόμηση αποθηκευτικού χώρου. Για εξασφάλιση υψηλότερης απόδοσης σε όλα τα σενάρια, μπορούν να χρησιμοποιηθούν τεμάχια των 128KB σε δέντρο ύψους 3, μιας που πέρα από την μικρή αύξηση, παρακάμπτονται τα εντοπισμένα σενάρια που η απόδοση των 64KB χωλαίνει. Εάν το βάρος πρέπει να στραφεί στην εξοικονόμηση χώρου, προτείνουμε τα 32KB, μιας που τόσο στην απαλοιφή διπλοτύπων όσο και στην εξοικονόμηση μεταδεδομένων μπορούν να παράγουν αξιόλογα αποτελέσματα, χωρίς να παρουσιάζουν, όπως τα 16KB, δραματικές βουτιές στην απόδοση για περιπτώσεις καθαρού δίσκου και COW.

Σημαντικό είναι να τονίσουμε ότι ο σχεδιασμός μας επιτρέπει την παραμετροποίηση κάθε εικονικού δίσκου ξεχωριστά, ανάλογα με τις ανάγκες και τα σενάρια χρήσης του. Τα αποτελέσματα των macro-benchmarks μπορούν να δώσουν μια κατευθυντήριο γραμμή, εάν γνωρίζουμε το σενάριο χρήσης του εικονικού δίσκου μας και αυτά των micro-benchmarks είναι πολύ διαφωτιστικά σε περίπτωση που γνωρίζουμε το κυρίαρχο μοτίβο λειτουργιών. Επίσης δεν πρέπει να ξεχνάμε ότι σε αντίθεση με υπάρχουσες λύσεις ο σχεδιασμός μας αναιρεί τις εξαρτήσεις μεταξύ των εικονικών δίσκων, στιγμιοτύπων και κλώνων, αποδίδοντάς τους αυτονομία, κινητικότητα και κλιμακωσιμότητα.

Να σημειώσουμε ότι οι παρατηρήσεις που κάναμε στο κεφάλαιο 5, μπορούν να γενικευτούν σε αδρές γραμμές και έξω από τα πλαίσια της υλοποίησής μας όπου οι οντότητες είναι αρχεία στο σύστημα αρχείων του host. Όπου αναφέρουμε ότι το εκάστοτε αποτέλεσμα εξαρτάται από τον αριθμό αρχείων (π.χ. clean allocation), τότε η εξάρτηση αυτή μπορεί να μετουσιωθεί σε εξάρτηση από τον αριθμό οντοτήτων.

Μάλιστα μπορούμε να παρατηρήσουμε, ότι σε σχέση με το qcow2 η επιβράδυνση εμφανίζεται εξαιτίας των επιπρόσθετων λειτουργιών που επιφέρει ο μεγάλος αριθμός των αρχείων και οι οποίες δεν υφίστανται στην περίπτωση του ενός αρχείου. Τέτοιες είναι οι αναζητήσεις στους καταλόγους, οι ενημερώσεις των inode, η εγγραφή των παραπάνω στο journal κ.τ.λ, και οι οποίες ευθύνονται για την μεγάλη αύξηση του χρόνου συγχρονισμού με τον δίσκο. Συνεπώς, εάν αλλάξουμε την τεχνολογία υλοποίησης, έτσι ώστε να μειωθεί ο αριθμός των παραπανίσιων αυτών λειτουργιών που δεν συμπεριλαμβάνονται στο καθαρό μοντέλο απόδοσης, θα έχουμε σημαντική επιτάχυνση. Η υλοποίησή μας με την χρήση συστήματος αρχείων στον host ήταν ενδεικτική για την έκθεση του σχεδιασμού μας, αλλά η απόδοση θα παρουσιάσει αξιοσημείωτη βελτίωση εάν τοποθετήσουμε τις οντότητές μας κατευθείαν σε σε raw storage ή εάν τις μετατρέψουμε σε αντικείμενα ενός object storage. Είναι λογικό άλλωστε, ένα γενικού σκοπού σύστημα αρχείων να μην μπορεί να υποστηρίξει την βέλτιστη απόδοση για τις ανάγκες του σχεδιασμού μας.

Εδώ θα παραθέσουμε μερικά μειονεκτήματα της υλοποίησής μας, για να υποστηρίξουμε την πεποίθησή μας ότι είναι δυνατή η περαιτέρω εκμετάλλευση των χαρακτηριστικών του σχεδιασμού μας.

Το ext4 μπορεί να υποστηρίξει μέχρι 2^{32} inodes (inode number   32 bit). Αυτό θα μας δημιουργήσει πρόβλημα για μικρά μεγέθη τεμαχίου αφού θα μπορούμε να δεικτοδοτήσουμε μέχρι και $2^{32} * chunksize$ δεδομένα οπότε για 4KB μπορούμε να έχουμε μέχρι 16TB. Προφανώς σε ένα περιβάλλον υπολογιστικού νέφους που μια τέτοια διαμέριση (partition) θα καλείται να αποθηκεύει πολλούς δίσκους ένα τόσο μικρό νούμερο είναι απαγορευτικό για την πρακτική εφαρμογή του συστήματός μας.

Ο εικονικός δίσκος μας δεν έχει ένα πολύ βασικό χαρακτηριστικό στο οποίο βασίζονται και πολλές βελτιστοποιήσεις λειτουργιών Ε/Ε του guest kernel, την τοπικότητα. Τα αρχεία των τεμαχίων δεσμεύονται σε συνεχόμενα μπλοκ του πραγματικού δίσκου με βάση την σειρά δημιουργίας τους. Έτσι αν έχουμε σειριακές εγγραφές συνεχόμενων

τομέων, τότε τα τεμάχια που θα τους περιέχον θα είναι τοπικά κοντά στον πραγματικό δίσκο. Όμως, τις περισσότερες φορές ο δίσκος μας θα δημιουργείται από τυχαίες εγγραφές οι οποίες θα δημιουργούν τεμάχια-αρχεία που είναι κοντά στον πραγματικό δίσκο αλλά έχουν απομακρυσμένους εικονικούς τομείς. Αυτό μπορεί να αποφευχθεί είτε με πρωθύστερη δέσμευση των τεμαχίων (χρονοβόρα διαδικασία που μπορεί να σπαταλήσει έξτρα χώρο, αφού ένας δίσκος των 100GB στον περισσότερο χρόνο ζωής του δεν θα είναι γεμάτος) είτε με συστήματα κατανεμημένης αποθήκευσης (που δεν βασίζονται στην χωρική τοπικότητα για να προσφέρουν ταχύτητα αλλά στην παραλληλία) είτε με chunk preallocation, όπου κατά την πρώτη εγγραφή ενός τεμαχίου-αρχείου δεν θα δεσμεύεται μόνο αυτό αλλά και ένας αριθμός γειτονικών του, ώστε να δημιουργήσουμε μεγάλες περιοχές που θα έχουν τοπικότητα.

Στον οδηγό μας χρησιμοποιούμε την κλήση `sync()` για να συγχρονίσυμε τα δεδομένα στον δίσκο. Όπως εξηγήθηκε δεν χρησιμοποιούμε `fdatasync()/fsync()/aio_fsync()` για κάθε αρχείο γιατί αυτό θα ήταν απαγορευτικά αργό ή απαιτητικό σε μνήμη, για μικρά μεγέθη αρχείων. Αντίθετα, το `sync()` που γράφει στο δίσκο (flush) τα δεδομένα ολόκληρου του συστήματος αρχείων, μπορεί να συνοψίζει όλες τις αλλαγές στα αρχεία και να είναι πολύ πιο γρήγορο, καθώς τελικά εκτελεί πολύ λιγότερες λειτουργίες Ε/Ε. Αυτό και γιατί τα μεταδεδομένα των τεμαχίων γράφονται με μαζικό τρόπο, και γιατί μπορούμε να επιτύχουμε καλύτερη χρονοδρομολόγηση όταν δεν αναγκάζουμε τις αιτήσεις να εξυπηρετούνται κατευθείαν από τον δίσκο, και γιατί δεν χρειάζεται να αποθηκεύουμε μια λίστα με file descriptors ή ονόματα αρχείων. Αυτό όμως έχει το τίμημά του αφού σε ένα περιβάλλον υπολογιστικού νέφους όπου το σύστημα αρχείων μας θα κρατάει τεμάχια-αρχεία για πολλούς δίσκους, μια λειτουργία συγχρονισμού από τον ένα εικονικό δίσκο θα αναγκάζει και τα δεδομένα του άλλου να συγχρονιστούν, αποδυναμώνοντας το προηγούμενο επιχείρημα υπέρ της χρήσης του ολικού `sync()` στο σύστημα αρχείων αντί για την κλήση των `fsync()`. Επίσης πρόβλημα ανακύπτει και όταν έχουμε εφαρμογές στον guest οι οποίες εκτελούν συχνά `sync()`. Αυτό συμβαίνει γιατί δεν εκμεταλλευόμαστε την host page cache (disk write cache για τον guest), με την απόδοση να πέφτει κατακόρυφα.

Μελλοντικές δυνατότητες

Η παρούσα εργασία προσφέρει δυνατότητες και για μελλοντική έρευνα που θα φωτίσει πτυχές που δεν εξετάστηκαν και θα βελτιώσει τις επιδόσεις του σχεδιασμού μας.

Αρχικά, αξίζει να διερευνηθούν τα πραγματικά οφέλη σε επίπεδο δεδομένων από την απαλοιφή διπλοτύπων, ενώ ιδιαίτερο ενδιαφέρον παρουσιάζει και η διερεύνηση της αποδοτικότητας της εφαρμογής του deduplicataion στα μεταδεδομένα, ώστε να διαπιστώσουμε εάν η υπόθεσή μας για απουσία σημαντικού χωρικού ή εξωτερικού πλεονασμού σε μεταδεδομένα, είναι όντως έγκυρη.

Ενδιαφέρον θα ήταν να μελετηθεί η χρήση της ιδιωτικής cache που προτείναμε στην ενότητα 4.4, και η οποία όχι μόνο θα έλυνε πιο αποδοτικά το πρόβλημα της συνεργασίας ανάμεσα στις πεπλεγμένες αιτήσεις εγγραφής, ώστε να αποφευχθεί η πιθανότητα διπλής δημιουργίας κόμβων/τεμαχίων κατά το COW, αλλά θα βελτίωνε και την απόδοση. Συγκεκριμένα, στην παρούσα υλοποίηση για κάθε αίτηση ανάγνωσης, ο οδηγός μας εκδίδει μια λειτουργία Ε/Ε προς το λειτουργικό για κάθε κόμβο που προσπελάζεται. Αντ' αυτού, σε μια υλοποίηση που θα περιείχε μια ιδιωτική userspace cache, δεν θα χρειαζόταν η έκδοση παρά μόνο της πρώτης λειτουργίας Ε/Ε για κάθε κόμβο, αφού στη συνέχεια οι επόμενες προσπελάσεις θα έβρισκαν τον κόμβο στην ιδιωτική cache. Οι αιτήσεις εγγραφής βέβαια, θα συνέχιζαν να πρέπει να εκδίδουν λειτουργίες Ε/Ε προς το λειτουργικό, αφού όμως πρώτα ενημέρωναν τις δομές στην μνήμη. Παρόλο που και στην παρούσα υλοποίηση οι προσπελάσεις κόμβων δεν προκαλούν προσβάσεις στο δίσκο, μιας και οι δομές του δίσκου αποθηκεύονται στην host page cache, η ιδιωτική userspace cache θα γλιτώναμε την επιβάρυνση έκδοσης λειτουργιών Ε/Ε. Παράλληλα, με αυτόν τον τρόπο θα γινόταν δυνατή η υποστήριξη της σημασιολογίας του «none» cache mode, δηλαδή παράκαμψης της host page cache, χωρίς όμως να πληγεί η απόδοση των αναγνώσεων, αφού θα διατηρούσαμε τις δικιές μας δομές αποθήκευσης.

Επίσης, περιορίσαμε την πειραματική αξιολόγησή μας, ως επί το πλείστον σε μονονηματικές προσεγγίσεις. Η εξαίρεση του πολυνηματικού filebench benchmark, μπορεί να μην δίνει κάποια αφορμή για διεξαγωγή πολυνηματικών μετρήσεων, αφού τα αποτελέσματα είναι σε γενικές γραμμές συμβατά με τα μονονηματικά, αλλά παρόλ' αυτά θεμιτή θα ήταν μια πιο ενδελεχής μελέτη των επιπτώσεων του παραλληλισμού στον σχεδιασμό μας, ώστε να αποφανθούμε και για την κλιμακωσιμότητα του σχήματος κλειδωμάτων μας.

Παραπάνω αναλύσαμε την αναντιστοιχία μεταξύ τοπικότητας στον εικονικό και στον πραγματικό δίσκο. Στις πειραματικές μας μετρήσεις, η δέσμευση των τεμαχίων έγινε

με την διατήρηση της τοπικότητας του εικονικού δίσκου, δηλαδή τα λογικά τεμάχια που είναι δίπλα στον εικονικό δίσκο, δεσμεύονται σε τεμάχια των οποίων τα μπλοκ δεδομένων είναι δίπλα και στον φυσικό δίσκο. Αυτό ήρε τις αλληλοσυγκρουόμενες δυνάμεις του μεγάλου μεγέθους και της μεγάλης έκτασης του δίσκου, ώστε να εξάγουμε κάποια πιο αντιπροσωπευτικά αποτελέσματα, αλλά δεν πρέπει να παραβλέπουμε το γεγονός ότι σε πραγματικά συστήματα, αυτό το φαινόμενο είναι αρκετά συχνό. Έτσι, αξίζει να εξεταστεί εάν οι καμπανοειδής καμπύλες απόδοσης σε σχέση με το μέγεθος τεμαχίου που παρατηρήσαμε είναι γενικότερα συνεπείς, και εάν ναι, τότε μπορεί να διερευνηθεί μια νέα συμφέρουσα παραμετροποίηση ή να υπογραμμιστεί η ανάγκη για υιοθέτηση κάποια εκ των λύσεων που σκιαγραφήθηκαν παραπάνω (μερική ή ολική πρωθύστερη δέσμευση τεμαχίων, χρήση συστημάτων κατανεμημένης αποθήκευσης).

Επιπροσθέτως, ενδιαφέρον θα ήταν να εξετάσουμε πώς επηρεάζει ο αριθμός των εικονικών δίσκων και των εμπλεκόμενων οντοτήτων τους, την απόδοση του συστήματος. Εμείς εκτελέσαμε τα πειράματά μας με την υπόθεση ότι κάθε κατάλογος (για τεμάχια, κόμβους και επικεφαλίδες) περιέχει μόνο τα αρχεία συναφή προς τον υπό εξέταση εικονικό δίσκο. Όταν όμως το σύστημά μας έχει αποκτήσει περισσότερους εικονικούς δίσκους, στιγμιότυπα και κλώνους, είναι αναπόφευκτο ο αριθμός των αρχείων σε κάθε κατάλογο να αυξάνεται. Αυτό σημαίνει ότι αυξάνεται ο χρόνος αναζήτησης στους καταλόγους, ενδέχεται να αυξάνεται ο κατακερματισμός για τους COW κόμβους/τεμάχια, ενώ στην περίπτωση πολλών ταυτόχρονα ενεργών εικονικών δίσκων να εισάγεται καθυστέρηση από το σύστημα αρχείων του host. Αυτές οι καθυστερήσεις είναι ίδιον τόσο της γενικότερης υλοποίησης μας που χρησιμοποιεί σύστημα αρχείων, όσο και των συγκεκριμένων λεπτομερειών της, όπως η οργάνωση των αρχείων και των καταλόγων. Εάν τα αποτελέσματα δείξουν ότι η αύξηση των εικονικών δίσκων επιδρά αισθητά στην απόδοση κάθε εικονικού δίσκου, τότε για να υποστηριχθεί η κλιμακωσιμότητα, μπορούμε είτε να συστήσουμε την εξ'ολοκλήρου απαλλαγή από το σύστημα αρχείων του host είτε να βελτιώσουμε την διάρθρωση των καταλόγων και των αρχείων, ώστε να υπάρχει μεγαλύτερη ισορροπία στον αριθμό αρχείων κάθε καταλόγου και να μην έχουμε αύξηση της απόδοσης εξαιτίας κοστοβόρων αναζητήσεων στους καταλόγους.

Μια ιδέα για βελτιστοποίηση είναι η τοποθέτηση των κόμβων σε SSD ή εναλλακτικά η υλοποίηση μιας SSD cache για την προφόρτωση των entries. Με αυτόν τον τρόπο θα μειωθεί δραματικά η καθυστέρηση, λόγω της απουσίας χρόνου αναζήτησης στους

SSD δίσκους. Εάν το κόστος το επιτρέπει, μπορούμε για ακόμη μεγαλύτερη βελτίωση να τοποθετήσουμε και τα τεμάχια του πρωτεύοντος εικονικού δίσκου σε έναν SSD δίσκο και να αφήσουμε σε HDD δίσκο μόνο τα τεμάχια που έχουν υποστεί deduplication και είναι πλέον immutable. Έτσι θα έχουμε γρήγορη προσπέλαση των hot δεδομένων του πρωτεύοντος δίσκου και θα εξοικονομούμε κόστος κατά την αποθήκευση σπάνια χρησιμοποιούμενων τεμαχίων. Μειονέκτημα βέβαια αποτελεί ο πεπερασμένος αριθμός εγγραφών, ο οποίος θα καταναλωθεί γρήγορα σε περιπτώσεις λήψης συχνών στιγμιοτύπων ή κλώνων.

Τέλος, όπως προείπαμε, αξίζει να εξεταστεί μια εναλλακτική υλοποίηση που θα αποτινάσσεται από τις επιβραδυντικές επιπρόσθετες λειτουργίες Ε/Ε που επιφέρει το σύστημα αρχείων του host και οι οποίες μεγεθύνονται σε περιπτώσεις χαμηλών μεγεθών τεμαχίου.

Βιβλιογραφία

[Agc] Jens Axboe and git contributors, *fio benchmark*, `https://github.com/axboe/fio`.

[BC05] Daniel P. Bovet and Marco Cesati, *Understanding the linux kernel, 3rd edition*, 3rd ed., O'Reilly Media, 2005.

[Blo70] Burton H. Bloom, *Space/time trade-offs in hash coding with allowable errors*, Communications of the ACM (1970).

[btra] *Btrfsf design wiki pages*, `https://btrfs.wiki.kernel.org/index.php/Btrfs_design`.

[btrb] *dedupremove*, `https://github.com/markfasheh/duperemove/blob/master/README.md`.

[Can] Benoît Canet, *qcow2 deduplication*, `http://www.linux-kvm.org/images/d/d6/Kvm-forum-2013-toward_qcow2_deduplication.pdf`.

[CLX+16] Qingshu Chen, Liang Liang, Yubin Xia, Haibo Chen, and Hyunsoo Kim, *Mitigating sync amplification for copy-on-write virtual disk*, FAST'16 Proceedings of the 14th Usenix Conference on File and Storage Technologies (2016).

[CNF+09] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee, *Better i/o through*

byte-addressable, persistent memory, SOSP '09 Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles (2009).

[CWTX14] Jie Chen, Jun Wang, Zhihu Tan, and Changsheng Xie, *Recursive updates in copy-on-write file systems - modeling and analysis*, Journal of Computers (2014).

[DGH+09] Cesar Dubniki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowsk, Cristian Ungureanu, and Michal Welnicki, *Hydrastor: A scalable secondary storage*, Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST 2009) (2009).

[ed] ext4 developers, *ext4 wiki pages*, `https://ext4.wiki.kernel.org/index.php/Main_Page`.

[ESKK+12] Ahmed El-Shimi, Ran Kalach, Ankit Kuar, Adi Oltean, Jin Li, and Sudipta Sengupta, *Primary data deduplication-large scale study and system design*, USENIX ATC'12 Proceedings of the 2012 USENIX conference on Annual Technical Conference (2012).

[FFH+15] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, and Yujuan Tan, *Design tradeoffs for data deduplication performance in backup workloads*, FAST'15 Proceedings of the 13th USENIX Conference on File and Storage Technologies (2015).

[gd16a] QEMU git developers, *The qcow2 image format documentation - new*, qemu.git, February 2016, `http://git.qemu.org/?p=qemu.git;a=blob;f=docs/specs/qcow2.txt`.

[gd16b] QEMU git developers, *Qemu source code*, February 2016, `https://github.com/qemu/qemu`.

[GK12] Kazuo Goda and Masaru Kitsuregawa, *The history of storage systems*, Proceedings of the IEEE (2012).

[JM09] Keren Jin and Ethan L. Miller, *The effectiveness of deduplication on virtual machine disk images*, SYSTOR '09 Proceedings (2009).

[JPZ+11] K. R. Jayaram, Chunyi Peng, Zhe Zhang, Minkyong Kim, Han Chen, and Hui Lei, *An empirical analysis of similarity in virtual machine images*, Proceeding Middleware '11 Proceedings (2011).

[Kat97] J. Katcher, *Postmark: A new file system benchmark*, Technical Report TR3022, Network Appliance (1997).

[Ker10] Michael Kerrisk, *The linux programming interface*, No Starch Press, 2010.

[Knu97] Donald E. Knuth, *The art of computer programming, vol 1: Fundamental aglorithms, 3rd edition*, 3rd ed., vol. 1, Addison Wesley Longman Publishing Co, 1997.

[MB12] Dutch T. Meyer and William J. Bolosky, *A study of practical deduplication*, ACM Transactions on Storage (2012).

[McL08] Mark McLoughlin, *The qcow2 image format documentation*, gnome, September 2008, `https://people.gnome.org/~markmc/qcow-image-format.html`.

[MCM01] Athicha Muthitacharoen, Benjie Chen, and David Mazières, *A low-bandwidth network file system*, SOSP '01 Proceedings of the eighteenth ACM symposium on Operating systems principles (2001).

[Mer87] Ralph C. Merkle, *A digital signature based on a conventional encryption function*, CRYPTO '87 A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (1987).

[NDR08] Dushyanth Narayanan, Austin Donnell, and Antony Rowstron, *Write off-loading: Practical power management for enterprise storage*, ACM Transactions on Storage (TOS) (2008).

[NKO+06] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jon Harkles, and Mahadev Satyanarayanan, *Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines*, Proceedings USENIX ATC 2006 (2006).

[NP02] Zachary Nathaniel and Joseph Peterson, *Data placement for copy-on-write using virtual contiguity (2002)*.

[pgb] *pgbench*, `https://www.postgresql.org/docs/devel/static/pgbench.html`.

[PP12] João Paulo and José Pereira, *A survey and classification of storage deduplication systems*, ACM Computing Surveys (2012).

[QD02] Sean Quinlan and Sean Dorward, *Venti: a new approach to archival storage*, First USENIX conference on File and Storage Technologies (2002).

[Rab81] Michael O. Rabin, *Fingerprinting by random polynomials*, Center for Research in Computing Technology, Harvard University (1981).

[Rod08] Ohad Rodeh, *B-trees, shadowing and clones*, ACM Transactions on Storage (2008).

[Rus08] Rusty Russell, *virtio: towards a de-facto standard for virtual i/o devices*, ACM SIGOPS Operating Systems Review - Research and developments in the Linux kernel (2008).

[SKM+16] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xia, and Erez Zadok, *A long-term user-centric analysis of deduplication patterns*, MSST '16 Proceedings of the 32nd IEEE Conference on Mass Storage Systems and Technologies (2016).

[TZS16] Vasily Tarasov, Erez Zadok, and Spencer Shepler, *Filebench: A flexible framework for file system benchmarking*, `https://github.com/filebench/filebench/wiki`.

[WDQ+12] Grant Wallace, Fred Douglis, Hangwei Qian, Philip Shilane, Stephen Smaldone, Mark Chamness, and Windsor Hsu, *Characteristics of backup workloads in production systems*, FAST'12 Proceedings of the 10th USENIX conference on File and Storage Technologies (2012).

[WKC07] Chin-Hsien Wu, Tei-Wei Ko, and Li Ping Chang, *An efficient b-tree layer implementation for flash-memory storage systems*, ACM Transactions on Embedded Computing Systems (TECS) (2007).

[XL+06] Weijun Xiao, Yinan Liu, , Qing (Ken) Yang, Jin Ren, and Changsheng Xie, *Implementation and performance evaluation of two snapshot methods on iscsi target storages*, NASA/IEEE Conference on Mass Storage Systems and Technologies (2006).

[ZLP08] Benjamin Zhu, Kai Li, and Hugo Patterson, *Avoiding the disk bottleneck in the data domain deduplication file system*, FAST'08 Proceedings of the 6th USENIX Conference on File and Storage Technologies (2008).