Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
Και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης Και Υπολογιστων

Επέκταση Συστήματος Μοιραζόμενης Κρυφής Μνήμης Επιπέδου Μπλοκ για την Αποδοτική Μετακίνηση Εικονικών Μηχανών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βασίλειος Π. Σουλελές

Αθήνα, Νοέμβριος 2015

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Επέκταση Συστήματος Μοιραζόμενης Κρυφής Μνήμης Επιπέδου Μπλοκ για την Αποδοτική Μετακίνηση Εικονικών Μηχανών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Βασίλειος Π. Σουλελές

Επιβλέπων Καθηγητής: Νεκτάριος Κοζύρης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 25η Νοεμβρίου 2015.

. . . . .	. . . . .	. . . . .
Νεκτάριος Κοζύρης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Αναπ. Καθηγητής ΕΜΠ	Λέκτορας ΕΜΠ

Αθήνα, Νοέμβριος 2015

. . . . . .

Βασίλειος Π. Σουλελές
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

0.1 Περίληψη

Τα υπολογιστικά νέφη έχουν καθιερωθεί σήμερα ως η καινοτόμα τεχνολογία ικανή να αλλάξει το τοπίο της πληροφορικής και των επικοινωνιών, προσφέροντας κατά παραγγελία κλιμακώσιμες υπηρεσίες υψηλής πιστότητας και διαθεσιμότητας. Η ανάγκη για υψηλές επιδόσεις σε αυτά τα συστήματα είναι το κίνητρο για την σταθερή πρόοδο και βελτίωση των παρεχόμενων υπηρεσιών.

Τα σύγχρονα περιβάλλοντα υπολογιστικού νέφους τείνουν να χρησιμοποιούν ένα κεντρικό μοντέλο αποθηκευτικού χώρου, συνήθως προσβάσιμο μέσω ενός ξεχωριστού δικτύου υπολογιστών. Συνεπώς το μονοπάτι των δεδομένων ανάμεσα στις εικονικές μηχανές και στους δίσκους τους εμφανίζει σημαντικές καθυστερήσεις, οι οποίες εισάγουν περιορισμούς στην συνολική απόδοση του συστήματος.

Για να αντιμετωπίσουν αυτό το πρόβλημα, οι διαχειριστές ενός υπολογιστικού νέφους συχνά επιλέγουν να χρησιμοποιήσουν σκληρούς δίσκους μνήμης Flash στους κόμβους που φιλοξενουν εικονικές μηχανές και να τους χρησιμοποιήσουν ως κρυφές-μνήμες επιπέδου μπλοκ για τα δεδομένα που έρχονται από και πηγαίνουν προς τους δίκτυο αποθηκευτικού χώρου.

Το κέρδος σε απόδοση είναι σπουδαίο για τις τυπικές λειτουργίες μιας εικονικής μηχανής επιπέδου μπλοκ, αλλά το σενάριο της "ζωντανής" μετακίνησης μιας εικονικής μηχανής αποδεικνύεται προβληματικό όταν χρησιμοποιείται η κρυφή μνήμη επιπέδου μπλοκ.

Για να αντιμετωπίσουμε αυτό το ζήτημα, σχεδιάσαμε, υλοποιήσαμε, και επεκτείναμε ένα υπάρχον σύστημα κρυφής μνήμης επιπέδου μπλοκ ώστε να του επιτρέψουμε να λειτουργήσει πάνω απο μοιραζόμενους δίσκους. Με αυτόν τον τρόπο οι εικονικές μηχανές μπορούν να μετακινούνται μεταξύ φυσικών κόμβων, ενώ ταυτόχρονα κάνουν αιτήσεις για λειτουργίες εισόδου-εξόδου επιπέδου μπλοκ προς τους δίσκους τους.

Για να πετύχουμε αυτό το αποτέλεσμα, εισάγαμε ένα νέο τρόπο λειτουργίας που ονομάζεται "Frozen Metadata". Η έρευνα μας υπαγορεύει πως αυτός ο τρόπος λειτουργίας πρέπει να χρησιμοποιείται κατά την διάρκεια μιας ζωντανής μετακίνησης μιας εικονικής μηχανής, προκειμένου να αποφευθχεί η καταστροφή και απώλεια δεδομένων.

Χρησιμοποιώντας συμβατικά εργαλεία δείξαμε πως είναι δυνατό μια εικονική μηχανή να μετακινηθεί από έναν φυσικό κόμβο σε έναν άλλον, ενώ συνεχίζει να λειτουργεί και κάποια απο τα αιτήματα της εξυπηρετούνται απο την κρυφή μνήμη επιπέδου μπλοκ.

Συνεπώς, τα αποτελέσματα της δουλειάς μας είναι πως πλέον τα οφέλη ύπαρξης μιας κρυφής μνήμης συνοδεύουν μια εικονική μηχανή από την στιγμή δημιουργίας της, μέχρι και την καταστροφή της.

Λέξεις κλειδιά: Εικονικοποίηση, Κρυφή Μνήμη Επιπέδου Μπλοκ, Μοιραζόμενοι Δίσκοι, Υπολογιστικά Νέφη, Μετακινήσεις Εικονικών Μηχανών

Abstract

Cloud environments have been established today as the new technology that can offer scalable and higly-available computational resources. Such systems are driven by a constant demand for performance improvements. Modern cloud clusters tend to use a centralized model of storage, offered through a dedicated network. Therefore, the path between instances and their disks imposes substantial latency, having significant impact on the overall system performance. Clouds often deploy a host-side Flash memory disk to act as a block-level write-back cache for data coming from and going to networked storage. The performance gain is tremendous for normal VM block I/O operations, but the scenario of instance live migration turns out to be problematic. To address this issue, we have designed and extended a block-level cache framework, making it capable of working over shared storage, allowing VM live migrations to be completed while the cache is still serving VM requests. We have achieved this by introducing a new mode of operation for the cache, analogous to write-back or write-through, called "Frozen Metadata" mode. Our new mode is meant to be used by the cache framework while live migration is taking place, to avoid data corruption. With common benchmarks we can show that it is now possible to migrate a VM while it is issuing block I/O towards the cache. Consequently, caches can now provide performance benefits that accompany instances from their creation time to their destruction.

Keywords: Virtualization, Block-level Cache, Shared Storage, Cloud Computing, Frozen Metadata, Virtual Machine Migrations

στους γονείς μου

Παναγιώτη και Σοφία

στα αδέρφια μου

Κωνσταντίνο και Ειρηνάντζελα

Contents

Αντί Προλόγου

Η παρούσα διπλωματική εργασία σημαίνει την ολοκλήρωση ενός σημαντικού κεφαλαίου της ακαδημαϊκής μου πορείας. Θα ήθελα στο σημείο αυτό να ευχαριστήσω ορισμένους ανθρώπους που με βοήθησαν στη διαδρομή αυτή.

Αρχικά θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Νεκτάριο Κοζύρη, ο οποίος μου έδωσε την δυνατότητα να ασχοληθώ με ένα τόσο σύγχρονο και ενδιαφέρον θέμα. Οφείλω επίσης ένα μεγάλο ευχαριστώ στο Δρα Βαγγέλη Κούκη, για την πολύτιμη βοήθεια του, για όλες τις εποικοδομητικές συζητήσεις μας και την διαρκή καθοδήγηση του, κατα την διάρκεια της εκπόνησης της διπλωματικής εργασίας μου, καθώς και για την όρεξη που μου ενέπνευσε για το αντικείμενο αυτό μέσα από τις διαλέξεις και τα μαθήματά του. Θα ήθελα επίσης να ευχαριστήσω τους συμφοιτητές, συνεργάτες και φίλους οι οποίοι με συντρόφευσαν σε αυτά τα σημαντικά χρόνια της φοίτησής μου, Οδυσσέα, Αλέξανδρο, Γιώργο, Ibrahim, Νίκο, Δημήτρη, Μιχάλη, Γιάννη, Πάνο Μ., Γιάννη Π., Νίκο Δ., Δήμητρα και Ντόρα και αρκετούς ακόμη που ίσως αυτή τη στιγμή να μου διαφεύγουν.

Τέλος, θα ήθελα να ευχαριστήσω τους γονείς μου, Παναγιώτη και Σοφία, καθώς και τα αδέρφια μου, Κωνσταντίνο και Ειρηνάντζελα, για την αγάπη, τη συνεχή υποστήριξη και συμπαράσταση που μου προσέφεραν έως τώρα.

Βασίλης Σουλελές
Νοέμβριος 2015

1 Introduction

In this chapter we outline the scope of our work. We first provide a quick overview of the problem we are trying to solve and argue about its importance. Next we shortly describe how naive approaches do not offer acceptable results and we highlight potential problems in achieving our goal. We move on to illustrate our proposed design and how it fits in as an solution meeting the optimal requirements. Finally we conclude with an early preview of some promising results.

1.1 Problem Statement

The primary objective of this work is to enable live migration of Virtual Machines that benefit from the use of a block-level cache. Block-level caching has already been deployed in cloud environments, by using host-side Flash memory disks such as SSDs. However, live migrations while a host-side cache is active have a number of complications. The main factor of this impediment lies in the architectural model of VM disks and block caches. Instance block devices are considered a part of the Virtual Machine entity and as such, they are managed by the hypervisor. The latter is not only responsible for creating and destroying the disks, but also carries out the preparation and the actual migration of the disks. This is not the case for the cache, though. The hypervisor and the instance itself, are not aware of the existence of the cache, because the cache is managed by the host OS. As a result, the cache cannot be managed by the hypervisor, and the host OS kernel is pledged to perform the cache migration. We assume that the cache like the instance hard disks, is stored in shared storage and consequently, a need

arises for coordination between the source and destination nodes during migration, in order to preserve data integrity while offering a significant performance boost. By taking KVM hypervisor as an example, we see that live migration is not a well-defined procedure when it comes to block I/O operations. The KVM process, does not clarify when the actual VM transfer is happening, and this phenomenon can be observed as concurrent block I/O operations issued by both source and destination host nodes at the same time. This case, combined with the fact that caches are externally managed by the host, clearly illustrate the need for coordination, or else data corruption is possible.

1.2 Motivation

The goal we set in the previous section is very important, because we aim to integrate two components of different nature but of complementary aspect, the Virtual Machine and its disk caches, into a unified system. This will enable truly atomic creation, processing, management and destruction of cloud instances, making system administration easier, while preserving the performance gain of using a cache.

1.3 Shortcomings

One straightforward way to handle this issue would be to destroy the cache before migration and re-create it after the migration has completed. However, this would kill performance and so, it cannot be considered in production environments for real-world use. Moreover, using an existing cache framework will not work either, because they are not built for shared storage and they do not contain concurrency logic. So we could either create a new distributed cache framework or we could modify an existing one and add a full-blown coherence protocol and distributed lock manager. Using a different approach we could even adapt the KVM hypervisor to provide us with clear information about where the instance is hosted in any time and whether it can or cannot issue block I/O.

1.4 Design

We target to incorporate block-level caching for a VM live migration. To achieve this, we haven't chosen any of the aforementioned techniques. Rather we have designed, implemented and expanded an existing cache framework by adding logic that enables a shared-disk cache to be used concurrently by more than one hosts. We have resolved this issue without adding complex distributed logic. Instead we have added a new operating mode called "Frozen Metadata" mode or "FM" mode. Before initiating live migration, the source node switches from write-back to our new mode. At the same time, the destination node must also set the cache mode to Frozen Metadata, and keep it this way, until the live migration has completed. The main idea it that while in FM mode, the set of cached blocks remain the same; that is, no new allocations occur, neither do evictions take place. By keeping the cached block set unchanged, we minimized the metadata updates that need to be tracked to a bare minimum. This enables source and destination host kernels to preserve data and metadata integrity, without explicit communication between them.

1.5 Results

To meet our goal, we have performed a number of development iterations, each one verging closer on an optimal solution. We have integrated the `EnhanceIO` cache framework with the Ganeti cluster management tool, and we have introduced the `eio_rbd` external storage provider. At the same time, we have developed a pair of usefull monitoring tools; one that can dump the `EnhanceIO` metadata from kernel memory to userspace, and one that can parse this information and display it in human-readable form. By using this setup as a testbed, we have been able to verify the correctness of our approach and actually prove that data corruption can be eluded when live migration is taking place.

2 Background

In this chapter, we provide the necessary background information required to understand the hardware and software environment targeted by this work. Initially, we review the basic principles of Virtualization as an ancestral technology of cloud computing. Next we proceed to explain what exactly is cloud computing and discuss how cloud systems can be deployed on computer clusters. The next section presents a short overview of modern storage solutions, outlines an typical OS storage stack, denotes the differences between block and file storage and examines how storage can be offered by cloud providers in the form of object stores.

2.1 Virtualization

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. Computer Virtualization has a long history, spanning nearly half a century. Over the last decade, Virtualization has transformed the IT landscape and radically changed the way people utilize technology, with the establishment of Cloud Computing.

2.1.1 A bit of history

We can trace the roots of Computer Virtualization back in 1960s [22][29][11][27], a time when computers were really big in size. Back then, computers ran programs in

batches which were read from individual punched cards. The punch cards would be directly loaded or have their contents uploaded onto large magnetic tapes. The mainframe would then be prepared to load a job, read the data from tape or punch cards, execute the program and record the output in another tape. This output would be offloaded and generally either sent to be printed or sent for punch card generation. There was no interactive ability between a running program and a user. It was a case of starting a program and waiting with crossed fingers to see if it produced the desired results hours or sometimes days later.

In 1959, Christopher Strachey, First Professor of Computation at Oxford University published his ideas to a paper entitled 'Time sharing in large fast computers'. Strachey described what he referred to as multi-programming which included memory protection, shared interrupts and some kind of interactiveness to allow a user to debug code. Later on, the traditional approach for a time sharing computer evolved into dividing up the memory and other system resources between users.

In 1962, the Massachusetts Institute of Technology decided to instigate a new project called Project MAC (Multiple Access Computer) with a goal to design and build an advanced time sharing OS (operating system). An interesting fact is that MultiCS was created as part of Project MAC at MIT and additional research and development was performed on MultiCS at Bell Labs, where it later evolved into Unix.

In 1967, International Business Machines Cambridge Scientific Center released their new operating system named CP-40 which stands for Control Program 40. It was developed to run on System/360 Model 40, a custom IBM Computer, and it was a new type of operating system that provided virtual memory and full hardware Virtualization. While running, CP would create 14 pseudo machines, later called Virtual Machines, and each VM would run CMS (Cambridge/Console/Conversational Monitoring System) with a fixed amount of 256k virtual memory. CMS was a single-user OS designed to be interactive with the user. It provided an interface with CP using privileged instructions (arguably an ancestor of today's system calls) and was also capable of presenting a file system to users. The main advantages of using virtual machines versus a time sharing operating system was more efficient use of the system because virtual machines were able to share the overall resources of the mainframe, instead of having the resources split equally between all users. Security was achieved by letting each user run in a com-

pletely separate Operating System. The system was also reliable since no user could crash the entire system; only their own OS.

A fun note is that in 1972, IBM VM/370 was the first system able to run a virtual machine from within another virtual machine, a procedure called virtual machine nesting.

During the 1980s, desktop computing and x86 platform became mainstream with the introduction of the IBM PC and its clones followed by the Apple Macintosh. In 1985 Intel released the 80386, a 32-bit microprocessor which introduced virtual 8086 mode. This mode could offer virtualized 8086 processors on the 386 and later chips. In 1987, Locus Computing Corporation developed Merge/386, a virtual machine monitor that enabled the direct execution of an Intel 8086 guest operating system under a host Unix System V Release 2 OS. The virtual machines supported unmodified guest operating systems and standalone programs such as Microsoft Flight Simulator. Merge/386 could run multiple simultaneous virtual 8086 machines as long as guest OS and programs were using valid 8086 instructions.

In 1998, a company called VMWare was established, and in 1999 began selling a product called VMWare Virtual Platform for the Intel IA-32 architecture. It was the first x86 Virtualization product and later VMWare would emerge as the biggest player in Enterprise Virtualization.

In 2001, VMWare released two products, called ESX Server and GSX Server. GSX Server allowed users to run virtual machines on top of an existing operating system such as Microsoft Windows. This is known as a Type-2 Hypervisor. ESX Server was a Type-1 Hypervisor, and did not require a host operating system to run Virtual Machines. A Type-1 Hypervisor is more efficient than a Type-2 hypervisor since it can be better optimized for Virtualization, and does not require all the resources it takes to run a traditional operating system.

In 2003, Xen was released and it was the first open-source x86 hypervisor.

The x86 platform wasn't originally designed to handle Virtualization but this changed in 2006 when both Intel (VT-x) and AMD (AMD-V) introduced limited hardware Virtualization support for the x86 architecture that allowed for simpler Virtualization software but offered very little speed benefits. Greater hardware support, which allowed for substantial speed improvements, came with later processor models.

In 2007 KVM was released. It is an open source project that has been integrated with the Linux kernel and provides Virtualization on Linux-only systems, utilizing hardware Virtualization support such as Intel VT-x and AMD-V.

2.1.2 Virtualization Today

Virtualization technology today has evolved enough to create usable Virtual Machines that can be indistinguishable from physical ones. As an example, we note that real world Operating Systems such as Microsoft Windows and GNU/Linux flavors can run unmodified on VMs. On such scenarios, the physical machine that runs the Virtualization software is called host, while the Virtual Machine is called guest.

Although other Virtualization platforms exist such as PowerPC, Sparc and ARM, the extreme majority of Virtualization software today runs on the x86 architecture.

Prior to 2006, Virtualization of the x86 architecture was made possible using techniques such as binary translation, page table shadowing and I/O device emulation. Binary translation was mandatory to take care of the privileged instructions issued by the guest VM. These instructions needed to be replaced by other instructions of lower privileges since guests are considered userspace applications and have limited access to actual hardware. Page table shadowing in software was needed to prevent important data structures from being modified by the guest OS. For example, the guest OS should not be allowed to change the actual page table entries. Such actions were emulated in software. I/O device emulation was required so that physically absent devices can be emulated on the guest OS by a device emulator that runs in the host OS.

A different approach is known as Paravirtualization. Paravirtualization is extending the guest OS with explicit modification so it knows it's running inside a Virtual Machine. The hypervisor exposes a software interface to the virtual machine that is similar but not identical to that of the underlying hardware. The intent of the modified interface is to reduce the portion of the guest's execution time spent performing operations which are substantially more difficult to run in a virtual environment compared to a non-virtualized environment. Paravirtualization provides specially defined 'hooks' to allow the guests and host to request and acknowledge the tasks which would otherwise be executed in the virtual domain where execution performance is worse. A successful par-

avirtualized platform may allow the virtual machine monitor or VMM to be simpler, by relocating execution of critical tasks from the virtual domain to the host domain, and/or reduce the overall performance degradation of machine-execution inside the virtual-guest. Paravirtualization requires the guest operating system to be explicitly ported for the para-API. A conventional OS distribution that is not Paravirtualization-aware cannot be run on top of a paravirtualizing VMM. However, even in cases where the operating system cannot be modified, components may be available that enable many of the significant performance advantages of Paravirtualization.

In 2005 and 2006, Intel and AMD, working independently, created new processor extensions to the x86 architecture. The first generation of x86 hardware Virtualization addressed the issue of privileged instructions. The issue of low performance of virtualized system memory was addressed with MMU Virtualization that was added to the chipset later. This break through was named hardware-assisted Virtualization and offers better performance while reducing the maintenance overhead of Paravirtualization by ideally eliminating the changes needed in the guest operating system. Rather than needing to do software emulation or binary translation, the hardware extensions do what might be called “hardware emulation”.

After having a fully virtualized system using hardware extensions to accelerate difficult-to-virtualize parts one may notice that the interface for networks and disks is unnecessarily complicated. Since nearly all modern operating system kernels have ways to load third-party device drivers, it’s a fairly obvious step to create disk and network drivers that can use the paravirtualized interfaces. This small step, from full Virtualization towards Paravirtualization, begins to hint at the idea of a spectrum of Virtualization. Today, the parts that can be fully virtualized or paravirtualized are: disk and network devices, interrupts and timers, the platform, such as the motherboard, device buses, BIOS, legacy boot, etc and finally the privileged instructions and pagetables. Depending on which of those parts one chooses to fully virtualize or paravirtualize, a different point on the Virtualization spectrum is produced.

Nowadays, it is safe to conclude that the term “Virtualization” refers to hardware-assisted Virtualization enforcing the concept of Paravirtualization to some parts of the guest OS.

2.1.3 Hypervisor

A hypervisor or virtual machine monitor (VMM) is a piece of computer software, firmware or hardware that creates and runs virtual machines. A computer on which a hypervisor is running one or more virtual machines is defined as a host machine. Each virtual machine is called a guest machine. The hypervisor manages and oversees the execution of the guest operating systems and is responsible for creating the illusion of actual hardware, as seen by the guest OS. When the Virtual Machine is running non-privileged code such as computations, the hypervisor remains idle and the VM code is allowed to execute on the physical host CPU achieving native performance. However, when the Virtual Machine issues a privileged instruction such as IO or software interrupts, the VM is paused, and the hypervisor is notified to handle this event. In Paravirtualization, anything that is slow or difficult to virtualize, is replaced by calls to the hypervisor, which are called hypercalls.

2.1.4 QEMU

QEMU is a generic and open source machine emulator and virtualizer[18]. QEMU was written by Fabrice Bellard, it is free software and is mainly licensed under GNU General Public License (GPL). When used as a complete machine emulator, QEMU can run OS and programs made for one architecture (such as an ARM board) on a different platform. By using dynamic translation, it can achieve very good performance. When used as a virtualizer, QEMU achieves near native performance by executing the guest code directly on the host CPU. QEMU supports Virtualization when executing under the Xen hypervisor or using the KVM kernel module in Linux. Guest operating systems do not need patching in order to run inside QEMU. When used with KVM, QEMU can virtualize x86 in an operating mode called 'KVM Hosting' in which QEMU deals with setting up, running, administering and migrating KVM instances. The execution of the guest is done by KVM on host CPU but QEMU is still involved in the emulation of hardware devices and peripherals, such as disks, buses, display cards, network cards and modes of connectivity.

2.1.5 KVM

KVM stands for Kernel-based Virtual Machine and it is a full Virtualization solution for Linux on x86 hardware that supports the Intel VT or AMD-V Virtualization extensions[12]. It consists of a loadable kernel module, `kvm.ko`, which provides the core Virtualization infrastructure, as well as a processor specific module, `kvm-intel.ko` or `kvm-amd.ko`. Using KVM, one can run multiple virtual machines running unmodified Linux or Windows images. Each virtual machine has private virtualized hardware: network cards, disk drives, serial devices, graphics adapter, etc. KVM is open source software. The kernel component of KVM is included in mainline Linux, as of 2.6.20. The userspace component of KVM is included in mainline QEMU, as of 1.3. They enable running fully isolated virtual machines at native hardware speeds, for some workloads. KVM uses a slightly modified QEMU program to instantiate the virtual machine. Once running, a virtual machine is just a regular process. KVM is part of Linux and uses the regular Linux scheduler and memory management. This means that KVM is very simple to use. It is also more featureful; for example KVM can swap guests to disk in order to free RAM. KVM does not support Paravirtualization for the CPU but may support Paravirtualization for device drivers to improve I/O performance. Simply put, KVM is a way to turn the Linux kernel into a hypervisor by adding a kernel module.

2.1.6 Xen

The Xen Project hypervisor is an open source Type-1 or baremetal hypervisor, which makes it possible to run many instances of an operating system or indeed different operating systems in parallel on a single machine or host[23]. The Xen Project hypervisor is the only Type-1 hypervisor that is available as open source. It is used as the basis for a number of different commercial and open source applications, such as server Virtualization, Infrastructure as a Service, desktop Virtualization, security applications, embedded and hardware appliances. The Xen Project hypervisor runs directly on the hardware and is responsible for handling CPU, Memory, and interrupts. It is the first program running after exiting the bootloader. On top of the hypervisor run a number of virtual machines. A running instance of a virtual machine is called a “domain” or “guest”. A special domain, called domain 0 contains the drivers for all the devices in the system. Domain 0 also contains a control stack to manage virtual machine creation,

destruction, and configuration. The hypervisor supports running two different types of guests: Paravirtualization (PV) and Full or Hardware assisted Virtualization (HVM). Both guest types can be used at the same time on a single hypervisor. It is also possible to use techniques used for Paravirtualization in an HVM guest and vice versa, essentially creating a continuum between the capabilities of pure PV and HVM. We use different abbreviations to refer to these configurations, called HVM with PV drivers, PVHVM and PVH.

2.1.7 VMWare ESXi

VMWare ESXi, formerly ESX, is an enterprise class, Type-1 hypervisor developed by VMWare for deploying and serving virtual computers[6]. As a Type-1 hypervisor, ESXi is not a software application that one installs in an operating system; instead, it includes and integrates vital OS components, such as a kernel. After version 4.1, VMWare renamed ESX to ESXi. ESXi replaces Service Console (a rudimentary operating system) with a more closely integrated OS. ESX/ESXi is the primary component in the VMWare Infrastructure software suite. ESX runs on bare metal, without running an operating system unlike other VMWare products. It includes its own kernel: A Linux kernel is started first, and is then used to load a variety of specialized Virtualization components, including ESX, which is otherwise known as the `vmkernel` component. The Linux kernel is the primary virtual machine; it is invoked by the service console and acts as the management domain, analogous to Dom0 in Xen. At normal run-time, the `vmkernel` is running on the bare computer, and the Linux-based service console runs as the first virtual machine.

2.1.8 VirtIO

So called “full Virtualization” is a nice feature because it allows you to run any operating system virtualized. However, it’s slow because the hypervisor has to emulate actual physical devices such as network cards and hard disks. This emulation is both complicated and inefficient. 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[28]. This enables guests to get high performance

network and disk operations, and gives most of the performance benefits of Paravirtualization. VirtIO was written by Rusty Russel and is chosen to be the main platform for IO Virtualization in KVM. The host implementation is in userspace by QEMU so no driver is needed in the host. Under the hood, VirtIO works by sharing memory between host and guest. The latter uses the VirtIO device drivers and instead of usual IO it writes data to a circular buffer and notifies the former. This simplification grants a significant performance boost.

2.1.9 Containers

Virtualization as described above is more or less operating system agnostic when it comes to guest operating systems. The primary advantage of hypervisor-based solutions is that they allow to run a fuller range of operating systems, just about any x86 operating system as a guest on a wide variety of host OS and Virtualization platforms such as Linux KVM, Xen or VMWare. Performance, however, is likely to take at least a slight hit when running a VM on a hypervisor. By introducing an extra layer of abstraction between the operating system and hardware with hypervisor Virtualization, definitely there will be a performance impact as a result. Also a complete OS stack must exist for each guest when using hypervisor Virtualization, from the kernel to libraries, applications, and so on. This will lead to additional storage overhead and memory use from running OS entirely separate.

An alternative to hypervisor Virtualization is container based Virtualization, also called operating system Virtualization [26][13][4]. One of the first container technologies on x86 was actually on FreeBSD, in the form of FreeBSD Jails. Instead of trying to run an entire guest OS, container Virtualization isolates the guests, but doesn't try to virtualize the hardware. Instead, there are containers for each virtual environment.

With container-based technologies, a patched kernel is needed and also user tools to run the virtual environments. The kernel provides process isolation and performs resource management. This means that even though all the virtual machines are running under the same kernel, they effectively have their own filesystem, processes, memory, devices, etc.

The net effect is very similar to hypervisor Virtualization, and there's a good chance

that users of the guest systems will never know the difference between using a system that's running on bare metal, under a hypervisor, or in a container. Usually containers are limited to a single operating system, the same one the host is running. Although hypervisor Virtualization usually has limits in terms of how many CPUs and how much memory a guest can address, the container-based solutions should be able to address as many CPUs and as much RAM as the host kernel.

2.2 Cloud computing and cluster management

2.2.1 What is Cloud Computing

Cloud computing, also known as on-demand computing, is a kind of Internet-based computing, where shared resources and information are provided to computers and other devices on-demand. It is a model for enabling ubiquitous, on-demand access to a shared pool of configurable computing resources.

Cloud computing and storage solutions provide users and enterprises with various capabilities to store and process their data in third-party data centers. It relies on the sharing of resources to achieve economies of scale, similar to a utility like the electricity grid over a network. At the foundation of cloud computing is the broader concept of converged infrastructure and shared services. Cloud computing, or in simpler shorthand just "the cloud", also focuses on maximizing the efficiency of the shared resources. Cloud resources are usually not only shared by multiple users but are also dynamically reallocated per demand. This can work for allocating resources to users. This approach helps maximize the use of computing power while reducing the overall cost of resources by using less power, air conditioning, rack space, etc to maintain the system. With cloud computing, multiple users can access a single server to retrieve and update their data without purchasing licenses for different applications.

The origin of the term cloud computing is unclear. The expression cloud is commonly used in science to describe a large agglomeration of objects that visually appear from a distance as a cloud and describes any set of things whose details are not inspected further in a given context.

The concept of resource sharing for better manageability and efficiency that stands in

the core of cloud computing is not new, as can be seen from the previous section where we talked about centralized compute resources shared among users during the 1970s.

Virtualization is the main enabling technology for cloud computing. Virtualization software separates a physical computing device into one or more "virtual" devices, each of which can be easily used and managed to perform computing tasks. With operating system-level Virtualization essentially creating a scalable system of multiple independent computing devices, idle computing resources can be allocated and used more efficiently. Virtualization provides the agility required to speed up IT operations, and reduces cost by increasing infrastructure utilization. Autonomic computing automates the process through which the user can provision resources on-demand. By minimizing user involvement, automation speeds up the process, reduces labor costs and reduces the possibility of human errors. Though service-oriented architecture advocates 'everything as a service', cloud computing providers offer their 'services' according to different models, which happen to form a stack: Infrastructure-, Platform- and Software-as-a-service[25]. In the following paragraphs, we give a brief overview of these core cloud technologies and we emphasize on IaaS, since our work targets the context of IaaS providers.

Software as as service, SaaS

In the Software-as-a-Service (SaaS) model, users gain access to application software and databases. Cloud providers manage the infrastructure and platforms that run the applications. SaaS is sometimes referred to as 'on-demand software' and is usually priced on a pay-per-use basis or using a subscription fee. In the SaaS model, cloud providers install and operate application software in the cloud and cloud users access the software from cloud clients. Cloud users do not manage the cloud infrastructure and platform where the application runs. This eliminates the need to install and run the application on the cloud user's own computers, which simplifies maintenance and support. Cloud applications differ from other applications in their scalability, which can be achieved by cloning tasks onto multiple virtual machines at run-time to meet changing work demand. Load balancers distribute the work over the set of virtual machines. This process is transparent to the cloud user, who sees only a single access-point. To accommodate a large number of cloud users, cloud applications can be multitenant, meaning that any

machine may serve more than one cloud-user organization.

Platform as as service, PaaS

PaaS vendors offers a development environment to application developers. The provider typically develops toolkit and standards for development and channels for distribution and payment. In the PaaS models, cloud providers deliver a computing platform, typically including the Operating System, programming-language execution environment, database, and a Web Server. Application developers can develop and run their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers.

Infrastructure as as service, IaaS

In the most basic cloud-service model providers of IaaS offer computers, physical or more often virtual machines, and other resources. IaaS refers to on-line services that abstract user from the detail of infrastructure like physical computing resources, location, data partitioning, scaling, security, backup etc. A hypervisor, such as Xen, KVM or VMWare ESX/ESXi runs the virtual machines as guests. Pools of hypervisors within the cloud operational system can support large numbers of virtual machines and the ability to scale services up and down according to customers' varying requirements. IaaS clouds often offer additional resources such as a virtual-machine disk-image library, raw block storage, file or object storage, firewalls, load balancers, IP addresses, virtual local area networks (VLANs), and software bundles. IaaS cloud providers supply these resources on-demand from their large pools of equipment installed in data centers. To deploy their applications, cloud users install Operating-System images and their application software on the cloud infrastructure. In this model, the cloud user patches and maintains the operating systems and the application software. Cloud providers typically bill IaaS services on a utility computing basis: cost reflects the amount of resources allocated and consumed.

Our work is strongly related to the IaaS cloud model. We aim to integrate a block-level caching solution with existing IaaS platforms and offer significant performance improvements to the storage subsystem of hosted instances.

2.2.2 Amazon Web Services

A well-known example of IaaS services is Amazon Web Services (AWS), which is a collection of remote computing services, also called web services, that make up a cloud computing platform offered by Amazon.com[2]. These services operate from 11 geographical regions across the world. The most central and well-known of these services arguably include Amazon Elastic Compute Cloud, also known as "EC2", and Amazon Simple Storage Service, also known as "S3". Amazon markets AWS as a service to provide large computing capacity more quickly and more cheaply than a client company building an actual physical server farm.

2.2.3 Google Cloud Platform

Another example of IaaS provider is Google Cloud Platform which is a cloud computing platform by Google that offers hosting on the same supporting infrastructure that Google uses internally for end-user products like Google Search and YouTube[10]. Cloud Platform provides developer products to build a range of programs from simple websites to complex applications. Google Cloud Platform is a part of a suite of enterprise solutions from Google for Work and provides a set of modular cloud-based services with a host of development tools. For example, hosting and computing, cloud storage, data storage, translations APIs and prediction APIs.

2.2.4 OpenStack

OpenStack is a free and open source cloud computing software platform and it is usually deployed as an infrastructure-as-a-service (IaaS)[16]. The technology consists of a group of interrelated projects that control pools of processing, storage, and networking resources throughout a data center which users manage through a web based dashboard, through command-line tools, or through a RESTful API. OpenStack.org releases it under the terms of the Apache License. OpenStack has a modular architecture with various code names for its components, such as Nova for compute which is the main part of an IaaS system and Swift for the Object Storage.

2.2.5 Ganeti

The most important cloud software targeted by this work is Ganeti [9][7][8]. We have used a Ganeti cluster for integration with our cache framework and for performing basic test and benchmarks. Ganeti is a cluster virtual server management software tool built on top of existing Virtualization technologies such as Xen or KVM and other open source software. Ganeti requires pre-installed Virtualization software on servers in order to function. Once installed, the tool assumes management of the virtual instances. Ganeti controls Disk creation management, Operating system installation for instances, and also Instance Management such as startup, shutdown, and failover between physical systems. Ganeti is designed to facilitate cluster management of virtual servers and to provide fast and simple recovery after physical failures using commodity hardware. Ganeti provides support for Xen and KVM Virtualization, support for fully and para- virtualized instances, support for live migration, for disk management and others. Ganeti is developed by Google and the first design included a solution stack that uses either Xen or KVM as the Virtualization platform, LVM for disk management, and optionally DRBD for disk replication across physical hosts. Ganeti is essentially a wrapper around existing hypervisors which makes it convenient for system administrators to set up a cluster.

2.2.6 Synnefo

Synnefo[21] is a complete open source cloud stack written in Python that provides Compute, Network, Image, Volume and Storage services, similar to the ones offered by AWS. Synnefo manages multiple Google Ganeti clusters at the back-end that handle low-level VM operations and uses Archipelago to unify cloud storage. To boost 3rd-party compatibility, Synnefo exposes the OpenStack APIs to users. Synnefo is being developed by GRNET (Greek Research and Technology Network), and is powering two of its public cloud services, the ~okeanos service, which is aimed towards the Greek academic community, and the ~okeanos global service, which is open for all members of the GÉANT network. In June 2010, GRNET decided to create a complete, AWS-like cloud service (Compute/Network/Volume/Image/Storage). This service, called ~okeanos, aims to provide the Greek academic and research community with access to a virtualized infrastructure that various projects can take advantage of experiments, simulations and

labs. Given the non-ephemeral nature of the resources that the service provides, the need arose for persistent cloud servers. In search for a solution, in October 2010 GRNET decided to base the service on Google Ganeti and to design and implement all missing parts in-house. Synnefo has been designed to be deployed in any environment, from a single server to large-scale configurations. All Synnefo components use an intuitive settings mechanism that adds and removes settings dynamically as components are getting added or removed from a physical node. All settings are stored in a single location. Synnefo is modular in nature and consists of the following components: Astakos as an Identity/Account service, Pithos for File and Object Storage services and Cyclades for Compute, Network, Image and Volume services.

2.3 Storage in the Cloud

Cloud storage is a model of data storage where the digital data is stored in logical pools, the physical storage spans multiple servers and locations, and the physical environment is typically owned and managed by a hosting company. These cloud storage providers are responsible for keeping the data available and accessible, and the physical environment protected and running. Cloud storage is based on highly virtualized infrastructure and is like broader cloud computing in terms of accessible interfaces, near-instant elasticity and scalability, multi-tenancy, and metered resources. Cloud storage typically refers to a hosted object storage service, but the term has broadened to include other types of data storage that are now available as a service, like block storage. Cloud storage is made up of many distributed resources, but still acts as one and is highly fault tolerant through redundancy and distribution of data.

2.3.1 Basics of Computer Storage

The core of computer storage is built around Non-Volatile Memory or NVM. Non-volatile memory is computer memory that can retrieve stored information even after having been power cycled, that is turned off and back on. Examples of non-volatile memory include read-only memory, flash memory, and most types of magnetic computer storage devices such as hard disk drives and magnetic tapes. Such type of devices can be found on every device that needs to store information and data, from mobile

phones and digital cameras, to common desktop computers and laptops and to cloud computing nodes, cluster members and supercomputers. All important data that needs to be saved permanently and possibly later retrieved and processed is stored in NVM.

The most common examples of NVM are hard disk drives and Flash memory. A hard disk drive or HDD, is a data storage device used for storing and retrieving digital information using one or more rigid 'hard' rapidly rotating disks called platters coated with magnetic material. The platters are paired with magnetic heads arranged on a moving actuator arm, which read and write data to the platter surfaces. Data is accessed in a random-access manner, meaning that individual blocks of data can be stored or retrieved in any order rather than sequentially. Since being introduced in 1960s, Hard Disk Drives have become dominant as secondary storage devices and have evolved in many ways, providing bigger capacity and faster performance while lowering the cost per unit of storage. Also their reliability and lifespan has been greatly improved and their size has been constantly shrinking.

In the field of Non-Volatile Memory devices, the main competitor of HDDs are SSDs. SSDs are really important for our work because they are faster than HDDs and can be used as block-level caches. A solid-state drive or SSD is a solid-state storage device that uses integrated circuit assemblies as memory to store data persistently. SSD technology primarily uses electronic interfaces compatible with traditional block input/output (I/O) hard disk drives, which permit simple replacements in common applications. SSDs have no moving mechanical components. This distinguishes them from traditional electromechanical magnetic disks such as hard disk drives (HDDs) or floppy disks, which contain spinning disks and movable read/write heads. Compared with electromechanical disks, SSDs are typically more resistant to physical shock, run silently, have lower access time, and less latency. Most SSDs use NAND-based flash memory which retains data without power.

Due to the mechanical nature of HDDs, the time needed to access data is limited by the speed of rotating disks and the movement of read/write heads. Seek time is a measure of how long it takes the head assembly to travel to the track of the disk that contains data. Typical seek time for common HDDs is about 10ms, dropping down to 4ms for high-end server drives. Seek time is an important factor of latency considering a 10ms delay for each block I/O operation of a poorly designed request pattern. SSDs, having no

mechanical parts, do not suffer such performance penalties and thus they respond faster. To mitigate this drawback, higher levels of the storage stack, such as the OS kernel and the filesystem, are often designed with rotational latency in mind. They tend to produce I/O load able to be served with a single movement of the head, rather than moving back and forth. This trick has greatly improved the performance and responsiveness of applications.

In terms of data transfer rate, commodity HDDs can transfer data up to 250MB/s assuming the head is correctly positioned and also assuming sequential I/O. On the other hand, SSDs can transfer data at about 300MB/s or even up to 700MB/s for high-end enterprise drives. Although SSDs seem to be an obvious choice when talking about performance, there is a catch. Except from being more expensive than HDDs per MB, SSDs have a lot shorter life span due to wearing off. Each block can support a finite number of writes before becoming unusable. So, if a particular block was programmed and erased repeatedly without writing to any other blocks, that block would wear out before all the other blocks, thereby prematurely ending the life of the SSD. For this reason, SSD controllers use a technique called wear leveling to distribute writes as evenly as possible across all the flash blocks in the SSD. As a final note we claim that there's not an obvious choice to me made between HDDs and SSDs and hybrid solutions emerge, as it will be seen in the following sections.

2.3.2 The OS storage stack

To better understand the path data follows from application level down to NVM, a full storage stack is examined in this section. Without loss of generality, the Linux storage stack is hereby used as an example of a modern Operating System Storage Stack. A simplified top-down view of the Linux I/O stack is shown in the following figure, and following that we talk about each layer in a bottom-up fashion.

- Applications
- VFS Layer
- FS
- Page Cache

- Generic Block Layer
- Block IO Scheduling Layer
- Block Device Driver Layer
- Actual HDD

Physical Disk and Drive Controller

The on-device disk controller exists at the lowest level of the Stack and it is the hardware part which empowers the communication with the disk drive. Modern disk controllers are integrated into the disk drive, such as built-in SCSI controllers. The controller is the component that enables the drive to be connected to the computer I/O bus, providing an interface for communication. The most common types of interfaces provided nowadays by disk controllers are PATA (IDE) and Serial ATA for home use. High-end disks use SCSI, Fibre Channel or Serial Attached SCSI (SAS).

Apart from access to the medium, the disk controller also embeds extra functionality such as intelligent command reordering, a small cache and a barrier operation. Disk controllers can also control the timing of access to flash memory which is not mechanical in nature, as in SSDs.

With Tagged Command Queuing (TCQ), the drive can make its own decisions about how to order the requests and in turn relieve the operating system from having to do so. For efficiency the sectors are serviced in order of proximity to the current head position, rather than in the order received. The result is that TCQ can improve the overall performance of a hard drive if it is implemented correctly.

Native Command Queuing (NCQ) is an extension of TCQ and also allows hard disk drives to internally optimize the order in which received read and write commands are executed. This can reduce the amount of unnecessary drive head movement, resulting in increased performance. NCQ differs from TCQ in that, with NCQ, each command is of equal importance, but NCQ's host bus adapter also programs its own first party DMA engine with CPU-given DMA parameters during its command sequence whereas TCQ interrupts the CPU during command queries and requires it to modulate the ATA host

bus adapter's third party DMA engine. Both NCQ and TCQ have a maximum queue length of 32 outstanding commands.

Another feature of the device controller is that allows the host to specify whether it wants to be notified when the data reaches the disk's platters, or when it reaches the disk's buffer or on-board cache. Assuming a correct hardware implementation, this feature allows data consistency to be guaranteed when the disk's on-board cache is used in conjunction with system calls like `fsync`. The associated write flag, which is borrowed from SCSI, is called Force Unit Access (FUA).

Block Device Driver Layer

One step above the on-disk drive controller, lies a mixture of hardware support and low level kernel code and device drivers, arbitrarily entitled as Block Device Driver Layer by our schema. This part of the stack is device-specific and is responsible for actually issuing the block I/O operations to the device. This is mostly done by setting up DMA for the physical data transfer between block devices and kernel memory pages.

Direct memory access or DMA is a feature of computer systems that allows certain hardware subsystems such as block devices and network cards to access main system memory independently of the central processing unit. Without DMA, when the CPU is using programmed input/output, it is typically fully occupied for the entire duration of the read or write operation, and is thus unavailable to perform other work. With DMA, the CPU first initiates the transfer, then it does other operations while the transfer is in progress, and it finally receives an interrupt from the DMA controller when the operation is done. If the device supports scatter/gather I/O, then the block device driver's job is to create a single DMA operation, which is much easier than creating a DMA mapping for each block I/O operation. Moreover, since this device driver is aware of the special capabilities of the device, it may issue manufacturer-specific commands to the disk, if higher levels of the stack request it.

This layer typically also includes a device driver for the Host Bus Adapter or HBA. This component allows the computer to talk to a peripheral bus such as PCI and SBus. Common host adapters are integrated into motherboards of modern computers for typical ATA, IDE and USB devices, but also PCI-X or ISA cards exist to allow connecting with SCSI, SAS, Fibre Channel, eSATA and InfiniBand devices.

Block I/O Scheduling Layer

On top of the Block Device Driver layer is the Block I/O Scheduling Layer. This is the part of the Linux kernel where block I/O operations are created as instructed by upper levels of the stack, managed and later dispatched to the device driver part below. The Block I/O layer is device agnostic, meaning that the same piece of kernel code offers functionality for many block devices without knowing any further details about the device itself.

The basic building block of the Linux kernel block subsystem since kernel version 2.6 is the `struct bio`. Each bio structure represents a block I/O operation by holding a block device start sector, the size of data to be transferred and a list of 3-tuples, each defining a `struct page`, a start offset and a size in bytes. For a read operation, a bio describes that the set of device blocks starting with the block device start sector and adjacent sectors until size reaches the size described by the bio should be transferred to the pages pointed by the list of 3-tuples. For a write operation, a bio states that data pointed to by the list of 3-tuples should be written to disk, starting with the bio start sector and writing a total of bio size data. The nature of bio structure implies some limitations. Namely, a bio cannot represent both a read and write operation. The size of I/O to be performed for a single bio is also restricted by the block device driver and it is limited by the capabilities of the actual device. Finally, a bio can refer to many different memory pages and to different start offsets within them, but it may only refer to contiguous sectors on the block device. In other words, a single read bio can read many adjacent device sectors and write them to many different kernel buffers in memory, and respectively a write bio can gather many different data from different points in memory and write them to a set of neighboring device sectors. In a struct bio, as defined in `linux/bio.h`, a lot more information is also stored, such as a pointer to the block device this bio refers to and others used mainly in keeping track of ongoing block I/O operations and extra accounting and housekeeping tasks. With this short description of a bio structure in mind, it is easy to describe a wrapper structure called `struct request`.

Essentially a request struct contains a linked list of struct bios along with some information on the total size of data to be transferred and others. Although a struct bio is the simplest form of an block I/O operation and refers to sets of blocks to be transferred, it is safe to characterize a request structure in kernel space as a high level block I/O request

that could have been created by some higher level entity.

Request structs, after being created, are kept inside queues, called request queues or `struct request_queue`. In the kernel, there is one request queue for each block device. Not only is this struct a queue for block I/O requests, but it also holds many useful information about those requests, many different parameters about the state of the block device and the service-able requests and finally it also implements the I/O scheduler or elevator. Simply put, a request queue is the place of kernel memory where requests are stored until dispatched to the device.

Request queues provide ground for request preprocessing and scheduling. The nature of struct bio, thus referring to contiguous device sectors, allows the request queue to merge two different request structs into one if the bio structs contained in those requests are found to be adjacent on disk. There are two types of merging, one called front merging and one called back merging. When two request structs are merged, they are removed from the request queue and a new request structure is allocated and placed into the queue. It is clear that by merging requests, the kernel is able to create bigger coalesced I/O requests that are more efficiently serviced by rotating hard disk drives with a single movement of the read/write head.

The part of the request queue that handles this types of operations is called the I/O scheduler or elevator. The scheduler is also responsible for forcing service policies to requests, for example by reordering the requests and giving priority to important block I/O operations over other less important ones. Depending on the merging policy, the reordering policy and the various accounting tasks that each scheduler utilizes, there are currently many different schedulers available.

One of them is called 'CFQ' and stands for 'completely fair queue'. It is pretty straightforward to see that this elevator does not include extra logic for a much better issuing order. Using a CFQ elevator should be suitable for most desktops and every day linux boxes, where responsiveness to user actions is a critical parameter. Another elevator is called 'anticipatory'. When using this scheduler, after receiving a block I/O request, the kernel waits for an other possibly adjacent in terms of device sectors block I/O request to come. If this happens, the requests are then merged, before sent to the device. An other one is the 'deadline' elevator. This one keeps track of how long do some requests stay in the request queue and enforces an aging policy by not allowing requests to wait

for a long period of time. For most server loads like databases and cloud computing services, 'anticipatory' and 'deadline' elevators should make a good choice. Finally, one more elevator of great importance is the 'noop' scheduler. This one is meant to be used with flash memory disks such as SSDs which do not depend on the order of block I/O requests to achieve great performance since they do not have moving parts. As a result, when using an SSD, one could use the 'noop' elevator.

The main user of the request queue structure is the request function. This function is called asynchronously by the kernel and it is expected to make some progress with the pending requests in the request queue. It is not required to actually complete any block I/O operations since they are purely asynchronous. The request function is used to make sure that the requests will eventually be serviced. The request function could also call the elevator and try to perform some kind of merging and/or reordering of the requests.

After having covered bios, requests, request queues, elevators and the request function, one final note on the Block I/O Scheduling Layer is about plugging. The kernel can control the request queue in two ways, by either plugging or unplugging it. When the request queue is plugged, a short time delay is introduced to allow some more request processing to happen such as merging and reordering. The request queue is then unplugged, which can be explained as a 'dispatch to disk' operation. Along with some bio flags, the act of unplugging is used to force a barrier between block I/O requests.

Generic Block Layer

Above the Block I/O Scheduling Layer, there is a thin part called 'Generic Block Layer'. Much of the actual block I/O creation and processing is handled by the lower level of the storage stack, so this layer serves block devices at a higher level of abstraction. Here one may find code that creates logical block devices and maps physical disks and partitions to logical ones. For example, the Linux Device Mapper framework is found in this layer, that can create many sophisticated mappings, enabling data replication, raid, and other logic. Moreover, block-level caches such as dmcache and flashcache, as described later are generally implemented in this layer.

Another important component of this level is a mapping tool called 'DRBD'. DRBD stands for Distributed Replicated Block Device and it layers logical block devices over existing local block devices. Then it performs synchronous replication of data between

the logical block device and other logical block devices, found on different hosts on the same cluster. In other words, DRBD is a kernel module that tracks block I/O activity on a given block device and replicates these operations on other block devices across a LAN network using TCP traffic. DRBD introduces the concept of Primary and Secondary nodes. Primary node is the node which actually issues block I/O and utilizes the block device, while secondary node is the one that mimics the block I/O operations. DRBD can also be used for asynchronous replication.

Page Cache

On top of the Generic Block Layer, we see the 'Page Cache' layer. This level is essentially memory pages used by the kernel as a write-back cache to hold data coming from and going to block and other devices. The Linux kernel can keep some blocks from HDDs and save them to unused memory pages resulting in quicker access to the contents of cached pages and overall performance improvement. Memory pages from page cache are later flushed to disk, to achieve consistency and persistently save data to HDD.

Potential OS crash would lead to data loss, if data is stored in the page cache but not in NVM. For that reason, applications can choose either to completely avoid the page cache, or they can enforce data writes to reach the disk. To avoid the cache completely, one can use the `O_DIRECT` flag. This way, instead of using kernel memory from page cache, the kernel uses the userspace address of an application buffer and directly sets up block I/O requests and bios to perform the I/O operation to disk. Alternatively, the userspace application could issue specific system calls, such as `fsync` and its variants to force a persistent write that will reach the medium.

Filesystems

The filesystem layer is responsible for creating the notion of files and directories on top of device blocks. This is achieved by mapping files and directories to physical blocks on disk. Without a file system, information placed in a storage area would be one large body of data with no way to tell where one piece of information stops and the next begins. By separating the data into individual pieces, and giving each piece a name, the information is easily separated and identified. The file system keeps track of which disk blocks hold

the contents of each file, keeps holds information about files stored inside directories and also stores metadata such as name, size, modification time, user access permissions and others. It is responsible for arranging storage space; reliability, efficiency, and tuning with regard to the physical storage medium are important design considerations.

It is common to store filesystem characteristics on a superblock on disk, including filesystem size, block size, empty and filled blocks and their respective counts. During runtime, if a filesystem's superblock cannot be accessed, the filesystem cannot be mounted and thus is unusable by the OS. For that reason, the superblock is replicated across the disk or partition at different block offsets.

Virtual File System Layer

The Virtual File System layer can be seen as a superset of filesystems. It combines many different filesystems that can be either block-based, network- based, or even memory-based into one hierarchical tree. Moreover, there are pseudo files in the VFS created by the kernel that may represent computer devices or running system parameters. By mounting different filesystems into nodes, the user applications cannot tell the difference between files and thus is created a form of homogeneity. After all, 'Everything is a file'.

Applications in Userspace

Finally, on top of the storage stack we find the user space applications and programs. They interact with files found on the VFS and they request to read and write data by issuing system calls such as 'open', 'read', and 'write'.

2.3.3 SAN Appliances and Shared Storage

Having covered the various types of disk drives and their special characteristics, we will now present the type of computer storage commonly used in cloud environments and thus in our work, for example SAN appliances and shared block storage.

Hard Disk Drives and Solid State Disks are usually referred to as Direct Attached Storage or DAS when they can be directly accessed through a Host Bus Adapter by the computer

using them. The protocol used for this communication can be one of ATA, SATA, SCSI, SAS, USB, IEEE 1934, FibreChannel or others.

In contrast to DAS, Network Attached Storage or NAS and Storage Area Network or SAN solutions exist to separate the storage users and the storage providers, inside a Local Area Network.

NAS is a file-level computer data storage server connected to a computer network providing data access to a heterogeneous group of clients. NAS is specialized for serving files either by its hardware, software, or configuration. Besides being networked, NAS is mainly used to provide both storage and a filesystem on top of it. NAS requires clients to use a special network file system protocol such as NFS to access remote files and directories. With NFS, users and programs can access files on remote systems as if they were stored locally. Network File System (NFS) is a distributed file system protocol originally developed by Sun Microsystems and it is used to coordinate and manage access to shared files, avoiding data corruption. However, storing data in form of files has little adoption in cloud environments for scalability reasons.

Instead of NAS, SAN appliances are installed and widely used in cloud configurations and clusters. A storage area network or SAN is a network that provides access to consolidated, block-level data storage. A SAN does not provide file abstraction, only block-level operations. When compared with NAS, one could argue that a SAN appliance is like a NAS but without the filesystem logic. Clients of SAN storage can either communicate at a block-level or they can mount the remote block devices as local and build special filesystems on them. Sharing storage usually simplifies storage administration and adds flexibility. Most storage networks use the SCSI protocol for communication between servers and disk drive devices. A mapping layer to other protocols is used to form a network such as iSCSI (SCSI over TCP/IP), Fibre Channel Protocol (FCP) which is SCSI over Fibre Channel, and others. SANs often use a Fibre Channel fabric topology, an infrastructure specially designed to handle storage communications.

A SAN appliance creates a medium reachable by all of the subscribers in a network, allowing simultaneous access by them. This type of storage device is called shared storage. A shared storage device has multiple ports or a way to identify and track multiple sessions in a single port. Some examples of shared storage include an IEEE 1934 device with two or more physical ports, a SAN appliance, a RADOS cluster, and an iSCSI server

with DAS devices. The semantics of a shared storage device define that the results of an operation issued by one user are visible to other users that have access to the medium.

NAS severs and SAN appliances, can be characterized as shared storage either when accessed at a block-level or when a filesystem is used. Filesystems implemented to work with SANs are known as shared-disk file systems or distributed file systems or even parallel filesystems. These file systems add mechanisms for concurrency control and provide a consistent and serializable view of the file system, avoiding corruption and unintended data loss even when multiple clients try to access the same files at the same time. It is a common practice for shared-disk filesystems to employ some sort of a fencing mechanism to prevent data corruption in case of node failures, because an unfenced device can cause data corruption if it loses communication with its sister nodes, and tries to access the same information other nodes are accessing. Concurrency control becomes an issue when more than one clients are accessing the same file or block and they want to update it. Updates to the file from one client should not interfere with access and updates from other clients. In shared-disk file systems, concurrency issues are resolved using a Distributed Lock Manager or DLM which serializes access to common files and data structures. Examples of shared-disk filesystems include OCFS2[15], GPFS[19], ZFS[24] and CephFS[3].

However, in most cloud environments that host VM instances, file level access to shared storage is not needed. Each VM can have one or more virtual hard disks attached to it. But even virtual hard disks are nothing more than a set of blocks, just as the physical ones. So, it is a straightforward step to utilize parts of the block storage offered by a SAN appliance directly as VM disks, rather than using a filesystem and creating disks as files on top of it. As a result, host nodes reserve a set of blocks from a SAN, create a local block device that is backed by those blocks and offer the local block device as a virtual disk to the VM.

This approach not only avoids the need for a distributed file system, but it also eliminates the need for concurrency control and shared access. Each VM only reads and writes blocks from each own virtual disk, and these blocks are mapped to different physical SAN blocks, hence concurrent access scenarios are not possible by definition. It is exactly like treating the SAN appliance as a bunch of separate block devices. It is important to note however, that shared storage rules still apply. If for some reason, two VM

instances were to use the same block device backed by the same SAN appliance, then the reads and writes of the one would be visible to the other. It is the same concept of shared storage, applied to VM instances instead of cloud hosts.

2.3.4 Object Stores

File and block storage are well defined and have been in use for a long time, but a new type of storage is now available and offered by cloud providers, called Object Store. Object storage, also referred to as object-based storage, is a general term that refers to the way in which work is organized in units of storage, called objects.

Objects are highly scalable, simple, cheap, distributed storage for the cloud. Every object contains three things. Firstly, the data itself. The data can be anything, from a family photo to a 400,000-page manual for assembling an aircraft or even binary data, part of a virtual hard disk etc. Secondly, an object contains an expandable amount of metadata. The metadata is defined by whoever creates the object storage; it contains contextual information about what the data is, what it should be used for, its confidentiality, or anything else that is relevant to the way in which the data is used. There is no limit on the type or amount of metadata, which makes object storage powerful and customizable. Lastly, the object has a globally unique identifier. The identifier is an address given to the object in order for the object to be found over a distributed system. This way, it's possible to find the data without having to know the physical location of the data, which could exist within different parts of a data center or different parts of the world.

Using objects as building blocks, object stores can host files of any type and size, including Operating System Images and Virtual Machine Disks. Object storage provides programmatic interfaces to allow applications to manipulate data. At the base level, this includes CRUD functions for basic read, write and delete operations. Some object storage implementations go further, supporting additional functionality like object versioning, object replication, and movement of objects between different tiers and types of storage. Most API implementations are REST-based, allowing the use of many standard HTTP calls.

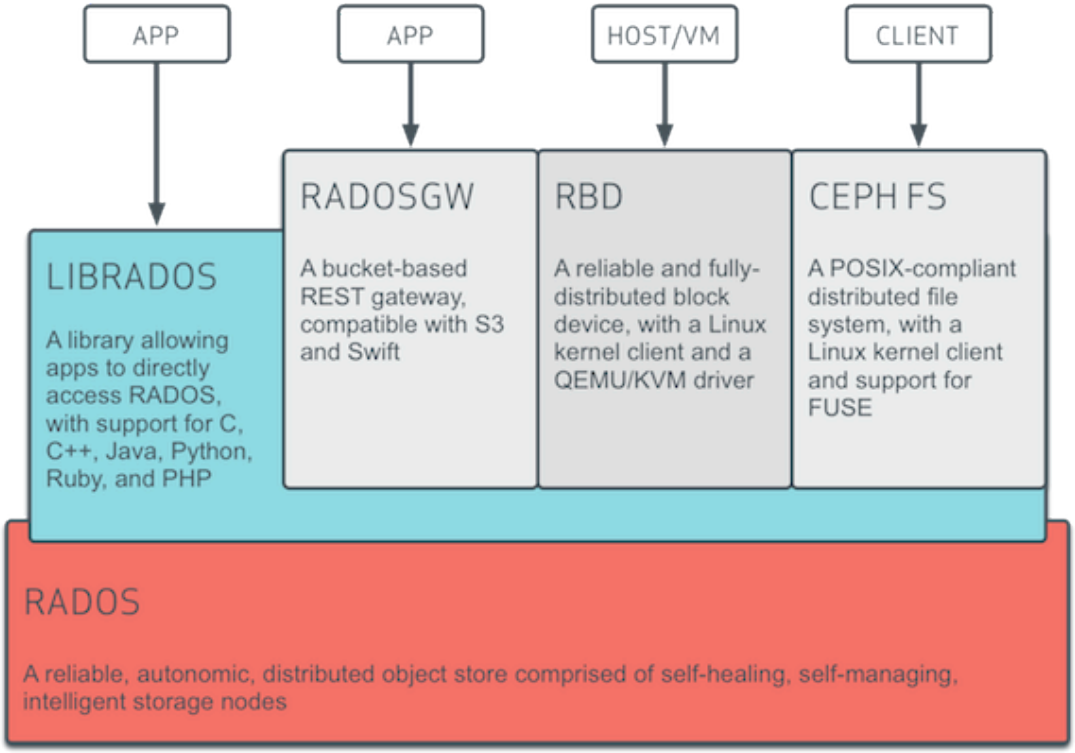
Pithos

Pithos+ is the Object Storage service offered by GRNET as part of the ~okeanos project[17]. It is based on OpenStack Object Storage API and it has a layered, modular implementation. It is built to scale, and it uses deduplication techniques such as content hashing for efficiency. It also supports object versioning and it can be queried for object metadata.

Ceph and RADOS

Ceph[1] is a free software storage platform that stores data on a single distributed computer cluster, and provides interfaces for object-, block- and file-level storage. Ceph aims primarily to be completely distributed without a single point of failure, scalable to the exabyte level, and freely available. Ceph replicates data and makes it fault-tolerant, using commodity hardware and requiring no specific hardware support. As a result of its design, the system is both self-healing and self-managing, aiming to minimize administration time and other costs. Ceph implements distributed object storage. Ceph's software libraries provide client applications with direct access to the reliable autonomic distributed object store (RADOS) object-based storage system, and also provide a foundation for some of Ceph's features, including RADOS Block Device (RBD), RADOS Gateway, and the Ceph File System.

The following figure, taken from the official Ceph Documentation, can best describe the Ceph layered architecture. All data are physically stored in the form of objects residing in a RADOS cluster; the nodes that actually save data to Hard Disk Drives are called "OSDs". On top of the object store, the Ceph Storage Stack software can offer direct object access through a programmable interface called "LIBRADOS", or it can offer higher level entities such as block devices and filesystems. The Ceph component that exposes usable block devices that are backed by objects is called "RBD" and stands for "RADOS Block Device". Such block devices can be made locally available with a procedure called "mapping" by any Operating System that supports the "LIBRADOS" library and then be used like a common block device. The Ceph component that exposes a POSIX-compliant distributed filesystem is called "CephFS" and can be used as a parallel filesystem over shared storage. Special Ceph nodes are responsible for administering access to filesystem metadata and they are called "MDS", which stands for "MetaData



Ceph Layered Architecture

Servers". Finally, some Ceph nodes act as monitors and they are called "MONs". It is their duty to keep track of the Ceph nodes that are up and running and provide administrative information.

3 System Analysis and Design

In this chapter we start with an examination of the fundamental concepts behind block-level caching using the EnhanceIO caching framework as a point of reference, since it is the framework used for our implementation. We then proceed to illustrate how data corruption is possible when a Virtual Machine utilizing a block-level cache is live migrated from a source host to a destination host. Next we present some naive approaches that fail to provide a working solution and finally we discuss our design, which should meet all requirements.

3.1 Block Caching

In this section we examine block-level caching. We present a detailed analysis of how it works, we provide extensive examples and we conclude with benchmark results that justify the usefulness of cache.

Caches are generally used in systems architecture to improve the performance of components that react slowly compared to their users. Any type of read/write memory can be essentially used as a cache, if it used to store data so that future requests for the same data can be served faster. CPU on-chip caches and content caches used in the World Wide Web are some of many well known examples of successful cache usage. Modern processors incorporate caches to save frequently used data and computational results, instead of reading and writing to main memory. The latter is typically an order of magnitude slower compared to the CPU. Respectively, Web caches store frequently requested content in order to serve clients faster and to reduce the traffic of main servers.

Following the same pattern, block-level caches are used to improve the efficiency of block devices.

Before going into details, it is important to note that a block cache is always transparent for applications and users. This means that only the host OS kernel knows about the existence of a cache device. Even if a cache is installed, applications will continue to see the same block device and direct to it any block I/O requests they produce. Only the cache framework, living at kernel level, knows about the existence of cache and it is responsible for mapping the requests either to cache or to disk.

Hard Disk Drives are organized as a set of fixed size chucks of data, called blocks or sectors, typically of size 512 bytes. The idea of block-level caching is to use faster disks as a cache for slower ones. Effectively, flash memory disks such as SSDs can save blocks and serve block I/O operations faster than slow spinning HDDs, and thus SSDs can work as block-level caches.

Cache memory is usually one or two orders of magnitude smaller in size compared to "main" memory, designed to hold only the most commonly accessed pieces of data. This is the case in block-level caching too, and SSDs are in fact smaller in size than HDDs. The former typically span a few hundred gigabytes, while the latter can store data in units of terabytes.

Using an example derived from the previous section, we can see the need for caches holding blocks coming from and going to networked block storage such as SAN appliances. SSD performance can be extremely fast when compared to network latency.

Just like HDDs, SSD drives can be seen as a set of of blocks. However, block caching does not involve storing individual blocks. Rather, caches save data in terms of a bigger size, called "cache block".

To avoid confusion, HDD device blocks and SSD device blocks will hereafter be mentioned as sectors, which is an equally valid name. The term "cache block" will be used throughout this dissertation to describe the basic data unit a cache can handle. Moreover, the term "disk" will be used to describe the big and slow pool of data such as SAN appliances, while the term "cache" will be used to describe the small and fast memory, such as SSD drives.

Cache blocks typically are a few sectors in size, for example 8, 16 or 32. In most Linux

systems, kernel manages memory by splitting it into "pages" of size 4KB or 4096bytes, which implies a very helpful alignment, if a cache block of size of 8 sectors is used. Locality of reference is another strong argument towards big cache block sizes. In a nutshell, it is a phenomenon describing the same value, or related storage locations, being frequently accessed. As a result, when storing more amount of related data in cache, it is highly likely that future requests will be served by the cache.

3.1.1 States of Cache Blocks

A cache search operation results either in a "cache hit" or a "cache miss". When a block is requested from disk, the cache framework performs a lookup to determine if the same block exists somewhere in cache or not. If the block is found to be present in cache, the operation is called "cache hit" and the request is served by the cache. Otherwise, it is called "cache miss" and the request is served by the disk. Blocks coming from disk can be stored in the cache for future references. Cache frameworks usually keep informational statistics about the hit/miss ratio.

To better organize cache space, cache blocks are grouped in sets. Sets allow for faster lookup times and enable easier cache management by cache frameworks. For example, a common technique is to create a many-to-1 mapping between disk sectors and cache sets. Then the cache framework can map each cache block, containing multiple disk sectors, to exactly one cache set, usually by performing bit operations on the sector number of the first cache block sector. The cache framework actually uses the sector number as a cache block identification, and a possible lookup is narrowed down into a cache set. Inside a cache set, the cache framework searches for cache blocks linearly or using a hash map. A usable cache set size is 256 or 512 cache blocks.

Although "cache hit" and "cache miss" provide a conventional binary description of a cache lookup result, the actual state of a cache block needs to be specified more accurately. A minimum of three states is required to correctly determine the status of a cache block. In the simplest case, where a cache block is unused and does not hold any data, we label it as INVALID. If a cache block holds data that are identical to the respective disk sectors, then the cache block is said to be VALID. Finally, if a cache block contains updated data compared to disk sectors, it is named DIRTY.

After defining the states of a cache block, it is easy to illustrate how a block transitions from one state to another. It is safe to assume that during the first installation a cache is initialized, possibly zeroed out, and thus all cache blocks start as INVALID. INVALID cache blocks cannot serve any requests, and cache misses are a frequent phenomenon. When a cache is in this phase, empty with the requested data constantly missing, it is labeled as a "cold cache". After a cache miss, the request is served from disk. Again, since a cache block is equal to many disk sectors, it is expected that a small block I/O request, for example a request for a single sector, will produce a greater block I/O request to disk, for example for a whole cache block, say 8 sectors. The cache block, is stored in cache after being retrieved from disk, and its state is changed from INVALID to VALID. Subsequent read requests for the same block will be served by the cache, and the cache block state will remain VALID.

If a write request is served by the cache, then the corresponding cache block will transition from VALID to DIRTY, because the cache block holds updated data relative to the disk. Under special circumstances, it is possible for a cache block to transition from state INVALID directly to DIRTY. For example when a layer above the case issues a write request without having previously issued a read request for the same cache block. When a cache contains a large number of VALID and DIRTY cache blocks it is called a "warm cache" and it expected to have a nice, high hit over miss ratio. DIRTY blocks are said to be "cleaned" when their contents are written back to disk. After a cleaning operation, cache block and disk sectors contain the same data, so the cache block state is changed from DIRTY to VALID.

Finally, for completeness, we mention the rare cases where a VALID or DIRTY cache block gets marked as INVALID. If, for a very special reason, a write operation is driven directly to disk bypassing the cache, then the cache blocks will hold stale data, and as such they should be marked as INVALID. This operation is called "write-around". The dual operation, is called "read-around", and it means to read data directly from disk, bypassing the cache. A read-around operation can be useful in cases when a very large amount of data is requested. In those cases, a cache controller may choose to direct these sequential read requests directly to disk, and save the cache throughput for heavier write operations, although some of the requested data could exist in the cache. However, a read-around operation is not permitted if the disk holds stale data compared to the cache, thus if the request involves cache blocks that are marked as DIRTY.

3.1.2 Side-effects of Reads and Writes

When using a cache, some situations occur that do not seem obvious at first read. For example, it is fairly counterintuitive but still possible, for a write operation to trigger a read from disk and respectively a read request to result in write operations. These cases can be better explained using some examples.

Consider the following scenario. A read block I/O request is issued by an application and the cache lookup signals a cache miss. The request is then directed to disk and served successfully. When the cache framework receives the relevant disk sectors, it realizes that there is no room available in the corresponding cache set for this new cache block. On such cases, a procedure called "eviction" is initialized. The cache framework will try to locate and remove another cache block, already residing in cache, in order to make room for the new cache block. If the chosen cache block is VALID, it is evicted by simply deleting its contents, and no further actions are needed. However, if the chosen cache block is DIRTY, it contains data that are not elsewhere available, and they cannot be discarded until the cache block is cleaned. As a result, before eviction, the DIRTY block must be flushed to disk. This completes the example and explains how a read operation can lead to a write I/O issued to disk.

Respectively, to prove the second point, let's assume the following scenario. An application issues a write request, but the I/O size is smaller than the cache block size. The blocks to be written do not exist in cache, and so the cache framework will try to locate an empty slot, effectively an invalid cache block, to write the new data. In our scenario, an unused cache block is indeed found, but the write operation cannot proceed. Cache blocks cannot be filled with fewer disk sectors than its actual size. Otherwise, a padding of zero bytes or even garbage would be later treated as valid data and written to disk, possibly overwriting useful information. To avoid this catastrophe, the complete cache block must be first read from the disk and then written into the cache. Then, the same cache block must be partially overwritten with the new data. This concludes this example and shows how a write operation can lead to a disk read.

3.1.3 Superblock and Metadata

The cache framework must keep a lot of organizing information including cache name, state, size, the current cache mode (which will be covered shortly), the set of current cache blocks and their state, and also which disk sectors are currently stored inside every cache block. To accurately keep track of all these data, the cache framework creates and maintains a very large set of metadata in memory. Moreover, in order to survive reboots, shutdowns and unplanned crashes, the cache framework stores a persistent copy of the metadata on the cache device.

Metadata kept in memory include cache block and set states, statistics and useful information for the replacement policy. A standard way to organize cache metadata is as an array of size equal to the number of cache blocks is stored in memory. For each cache block, the cache framework saves the current cache block state and the device sector number of the first sector residing in the cache block in the respective position of the array. This sector number works as an identification and allows the cache framework to know exactly which sectors of the disk exist in the cache block.

In-memory metadata are frequently updated, exactly before each operation is completed. For example, after each cache allocation, cache eviction and whenever a cache block is being dirtied, the in-memory metadata are updated.

More precisely, in-memory metadata implement a bigger and more complex state machine, than the one we have described earlier. In-memory metadata need to reflect the current state of cache for every given time, and as result it takes more than three states to describe asynchronous operations such as block I/O requests. Cache blocks can be marked in-memory as “Cache read in progress”, “Disk read and cache fill in progress”, “Disk write and cache write in progress” etc. This is mandatory so that the cache framework can continue to work and successfully manage all the cache blocks.

On the other hand, metadata kept on device are used to store information about the state of cache blocks persistently. Persistent metadata is the only way a cache can survive a sudden crash or power loss.

On-disk metadata are less frequently updated due to the cost of a block write I/O operation. Full on-disk metadata updates are only issued during a normal shutdown. While the cache is being used, it issues only the bare minimum of metadata updates required.

For example, after a read miss operation, a cache block is allocated and filled with the requested data. This operation does not trigger an on-disk metadata update. The rationale behind this design is that it is not critical to save this information. In a crazy scenario where this information is lost, it would cause no harm, since the data could be read again from disk. On-disk metadata updates are only triggered by actions that create new data. For example after a write hit, cache holds newer data than disk, and this state must be saved permanently. In case of a sudden crash and reboot, it must be clear that the cache block is holding the new data, and any subsequent read operation must be served from the cache, and not from the disk. Additionally, if for whatever reason, that particular block needs to be later evicted, it must be clear that it needs to be cleaned first. Another example that would trigger on-disk metadata update is a "write-around" scenario. In this case, a valid cache block must be permanently marked as invalid, so that future references, after a possible crash, are served from disk and not from cache.

Finally, a superblock structure is stored on the cache device, usually in adjacent sectors to the on-device metadata. The superblock contains useful information such as cache name and size, mode of operation. In addition, it holds other information such as the software version of the cache framework, the name of the backing disk device, etc. Superblock information is used when a cache is reloaded after a host reboot or after a sudden crush. In the first case, before reboot, cache metadata are flushed from memory to the cache device. Each cache block that is DIRTY is first cleaned, thus flushed to disk. As a result, cache metadata stored on the device include VALID cache blocks. After booting, cache metadata are read from the on-device metadata section and loaded into memory. The superblock provides extra information about cache name, size, mode of operation and others. Cache block metadata are used to find out which of the cache blocks are VALID and which are INVALID. This scenario, where VALID cache blocks exist in cache and can requests can be served right away is called 'warm boot'. The opposite is called 'cold boot', where a cache is started as if it was new, without any VALID blocks residing in it.

The superblock is useful in two more scenarios. First, the superblock can effectively be used to accurately detect unplanned crashes. This can work with a procedure called "Dirtying the Superblock". After the initialization of cache, the superblock is immediately updated on the cache device and marked as DIRTY. This superblock state is kept until one of two things happen, either a clean reboot or a sudden crash. In the first

case, the superblock is updated and marked as 'Clean'. In the second case, nothing happens since the crash is unexpected and the superblock remains DIRTY. In both cases, on next boot the cache superblock is read. If the superblock is found to be Clean, it is a clear indication of a clean reboot and all VALID cache blocks residing in cache can be used. So, cache metadata for VALID cache blocks are loaded in memory and cache performs a warm boot. However, if the superblock is found to be DIRTY, it can only mean that a sudden crash has occurred. In this case, only the DIRTY cache blocks are considered present; the VALID cache blocks are ignored. To understand this, suppose a write is directed to both cache and disk. Disk write completes, but before cache write is completed the host crashes. If after reboot, the same blocks are requested, reading from cache would return stale data. As a result, VALID blocks cannot be fully trusted after an unclean reboot. After checking the superblock state and deciding on whether the reboot was clean or not, the superblock is again marked as DIRTY.

Second, the superblock can be used for a quick reboot scenario. Cleaning the cache for every reboot is resource hungry, resulting in a lot of I/O requests being generated to write DIRTY cache blocks from cache to disk. It is possible to perform a cache "fast remove", where DIRTY blocks remain in cache rather than cleaned. This is possible, again by marking the superblock with a special flag called FAST. After reboot, the superblock is read from cache device. If it is found to be Fast, a fast reboot scenario is assumed and all cache blocks, both VALID and DIRTY are trusted, and all cache metadata are loaded in memory. This is also a scenario of a warm boot.

3.1.4 Modes of Operation and Replacement Policies

The most important aspect of a cache, is its behavior after a cache hit or cache miss. It is determined by the current mode of operation,

The way a cache deals with an I/O request depends on its current mode of operation, which can be one of 'Read-only', 'Write-through' and 'Write-back'. We will now explain each one of those.

Replacement policies are used to decide which cache block will be evicted when a new cache allocation must take place. 'Random' and 'FIFO' options exist, but the 'Least Recently Used' option is most commonly used.

In Read-only mode, the cache is only used to serve read requests and all write operations are directed to disk. For example, after when an application requests some data living in certain disk sectors, the host kernel puts that process to sleep until the block I/O operation is complete. Then a block I/O request is created and submitted to the disk block device. The cache framework is then notified and performs a lookup to see if the requested sectors exist in some cache block. In the case of a cache hit, the data are read from the cache and copied into kernel memory, the read operation is completed and the process will later be woken up. Otherwise, it is a cache miss and so another block I/O request is created and submitted to disk. Once the data are read from disk and copied to host memory, a write block I/O request is submitted to write the same data to cache. After that write operation is done, the read operation is completed, and the process will later be woken up. In the future, if the same sectors are again requested by a read operation, they will be served by the cache. A write block I/O request is always directed to disk. After the disk confirms that a write operation is done, the cache framework invalidates the relevant cache blocks if any, and then the write operation is completed and the application will later be woken up. Invalidation is mandatory since cache data are stale after a write around operation. We see that in Read-only mode there is no notion of DIRTY cache blocks. Only INVALID and VALID cache blocks exist. Read-only mode can help future read requests to be served faster, while writing all data back to disk for ultimate consistency.

Write-through mode behaves the same with read-only mode for read operations. However, write requests are handled differently; they are directed to both the cache and the disk. After an application requests to write some sectors, it is put to sleep by the host OS and the cache framework generates two block I/O requests, one towards the disk and one toward the cache. After both writes are done, the write operation is considered complete and the application will later be woken up. A write request in Write-through mode does not cause any invalidation. Same as in Read-only mode, there cannot be DIRTY cache blocks in Write-through mode, only INVALID and VALID cache blocks. Write-through mode is generally a better choice than read-only. It has all the benefits of Read-only mode, while offering a better cache hit ratio, by keeping written data into cache.

As far as Read requests concerned, Write-back mode behaves the same as with Read-only and Write-through modes. On a cache hit the cache serves the data, while on a cache miss data comes from disk and a cache block is allocated in cache. In Write-

back mode, writes are exclusively served by the cache and as long as data are written to the cache block the write operation is completed. This enables very fast writes but introduces data inconsistency between the cache and the disk; the cache holds the true data while disk is in stale state. In a Write-back cache, a clean thread is responsible for cleaning cache blocks and flushing their content to disk at a predefined interval, one minute for example. In contrast with Read-only and Write-through modes, data loss is possible when using a write-back cache. If the cache device is permanently destroyed after a write operation has been completed and before the cache block is cleaned, the data will be lost and irrecoverable.

3.2 Detailed Problem Statement

Let's assume a cloud provider, maintaining a computer cluster and offering Infrastructure-as-a-Service resources to its users. The cluster consists of many hosts connected to a Local Area Network and each host features high-end CPUs and a lot of RAM, in order to accommodate Virtual Machines. Block storage is offered by a Storage Area Network appliance through a separate dedicated high speed network. The SAN appliance supports advanced data replication and synchronous backup services and as a result block I/O performance is poor. Moreover the SAN appliance also includes a second storage pool, consisting solely of SSDs. Data storing in this caching tier is very fast and thus it can used as a block-level cache.

All hosts are assumed to run the Linux Operating System, running the Ganeti cluster management tool, and supporting the KVM hypervisor. A set of KVM instances are hosted in this Ganeti cluster, equally shared among hosts, and each instance has one or more virtual disks. The SAN appliance provides block storage to all nodes in the form of distinct block devices, and each node attaches the remote block devices as local disks. A host can request a block device to be created either in the caching tier or in the backing store. The hosts also support a caching framework such as EnhanceIO, and they set up caches using block devices from the caching tier, to enhance the performance of block I/O operations towards the featureful backing SAN appliance. Once the 'fast' and 'slow' block devices are combined, the augmented block device is then passed as a virtual disk to an instance, and the KVM hypervisor deactivates page cache support for that block

device. Therefore, all block I/O operations are directly serviced by the shared block storage, ensuring stronger data consistency. In most cases where a cache framework is used, the write-back cache mode is selected for the faster cache writes it can provide.

3.2.1 I/O Data Path

Let's use an example the describe the data path. An application running inside in the userspace of a KVM instance is generating I/O load by reading and writing data to files. The OS kernel running inside a VM produces block I/O requests and directs them towards a directly attached physical disk, as seen from its perspective. The virtual hardware is emulated by the QEMU/KVM process running in the userspace of the host. Block I/O requests issued to a virtual disk are intercepted by QEMU and translated into real I/O towards a physical local disk. The host OS kernel, who is aware of the mapping between local block devices and block devices residing in the SAN appliance, directs I/O to the remote shared block storage. Furthermore, if a cache is installed, then a pseudo-block device is used by the host OS kernel instead. When I/O operations are issued to this block device, the cache framework decides whether they should be served by cache or by the backing store and directs the requests to the corresponding block device.

The separation of control between the KVM instance and the cache is clear. The instance is managed by the hypervisor while the cache is managed by the host OS kernel. The hypervisor is responsible for creating the instance, providing virtual disks and network cards, emulating the hardware, migrating the instance if necessary and finally destroying it. The host OS kernel has no way of controlling the instance. Respectively, the host OS kernel is only aware of remotely-accessed shared-storage block devices. The kernel manages the remote-to-local mapping and generates a pseudo block device that represents sectors residing in the SAN appliance. Moreover, when a cache is used, the kernel merges two local pseudo block devices into one, creating a cached block device. It is agnostic to the kernel how this cached block device will be used, and of course the KVM hypervisor is not aware of the cache existence.

3.2.2 Live Migration

Before we explain why an instance utilizing a block-level cache cannot be live migrated, we present a live migration scenario for an instance that does not use a cache. Again, by using the term "an instance uses a cache" we do not mean that an instance knows anything about the cache, but rather, that the instance is given a special block device as disk, which block device is managed by a cache framework at the host OS kernel level. Live migration is used to move a guest instance from one physical host to another for load balancing reasons or to evacuate all instances from a physical node to enable scheduled maintenance operations to proceed. This is possible because guest virtual machines are running in a virtualized environment instead of directly on the hardware. So, live migration is a procedure that requires two hosts. The node currently hosting the instance is called the "source", while the node on which the instance is to be migrated is called the "destination" or the "target". Live migration is performed completely by the hypervisor and since we assume there is not a control communication protocol between host hypervisors, an external tool such as Ganeti is required to trigger the migration in both source and destination nodes. Ganeti initializes the migration on destination host first, who listens to a network port for incoming data. Then Ganeti fires up migration on the source host by informing it of the IP address of the destination node.

Migration works by sending the state of the guest virtual machine's memory and any virtualized devices to the destination host. In a live migration, the guest virtual machine continues to run on the source physical machine while its memory pages are transferred, to the destination host. During migration, KVM monitors the source for any changes in pages it has already transferred, and begins to transfer these changes when all of the initial pages have been transferred. KVM also estimates transfer speed during migration, so when the remaining amount of data to transfer will take a certain configurable period of time (10ms by default), KVM suspends the original guest virtual machine, transfers the remaining data, and resumes the same guest virtual machine on the destination host physical machine. The whole VM state is transferred and upon success the guest will continue to run on destination host with almost unnoticeable guest down time.

KVM expects that both source and destination hosts have access to the underlying storage used for the guests virtual disks, so that only guests memory and state is migrated. This greatly reduces migration time and network usage. If instance disks are residing on

shared storage such as a NAS appliance, all hosts have access to each instance disks, so live migration is possible. Before the migration takes place, the destination host maps the instance disk to a locally attached block device. After access to all the instance disks is established by both source and destination nodes, the administrator can initiate a live migration. Although cache devices are also stored in shared storage, it is clear that the cache is not part of the VM state, and therefore it cannot be managed like the instance disks by the KVM hypervisor.

One might argue that the KVM hypervisor does not in fact manage the instance disks during migration; it only confirms that both hosts have access to them. The instance is of course allowed to issue block I/O requests while being migrated and as a result action must be taken to certify that requests issued by the instance while on source host are served successfully and completed. Then the instance is temporally paused without ongoing block I/O requests pending, and then the instance is resumed on destination host and allowed to perform more I/O. It would be really nice if the KVM hypervisor performed such a clean and well defined procedure while live migrating an instance. However, that is not the case.

The KVM hypervisor introduces some ambiguity during the migration and allows block I/O operations to be performed by source and destination hosts at the same time. This phenomenon occurs because of two reasons. First, KVM running on the destination host must issue some read I/O requests to the instance disks to verify their existence and readability. KVM may also search the disks for superblocks indicating LVM volumes, partitions and certain filesystems. These actions produce read block I/O requests towards instance disks. Although write operations seem counterintuitive in this scenario, they are not excluded by the inner workings of a cache, as described in the previous section. Second, KVM is trying hard to keep track of clean pages that need to replicated to destination host while searching for dirty pages that may or may not have already transferred. This tedious task is extremely important and allows the instance pause time to be really small. However, to make this procedure as fast as possible, the KVM hypervisor is not transparent as to when exactly the instance is paused, the final pages are copied through the network to the destination host, and the instance is restarted The KVM hypervisor does not provide programmable hooks to let the administrator know exactly when live migration is taking place. This ambiguity leads to a phenomenon of reads and writes being issued by source host and then suddenly by destination host, without any

further notice. With these two facts combined, we can only create an abstract model of KVM's live migration. This model assumes that for a finite time interval, block I/O operations can be issued from the guest instance to the instance disks, by both source and destination hosts. In other words, from the SAN's perspective one may observe block I/O requests arriving for the same block device but originate at separate hosts.

3.2.3 Complications of using a shared cache

Quite interestingly, shared block storage semantics can comply with this model. Block I/O operations are considered atomic and concurrent access to shared storage is indeed allowed. Clients of shared storage, in this case the two hypervisor processes, are the only ones responsible for data integrity by synchronizing their access. As a result, this scenario although complex is not problematic, and it is exactly how KVM currently performs live migrations. However, the problem is introduced by using a block-level cache.

As previously stated, cache metadata are stored both in memory and on the cache device. The memory area the metadata are saved to, belongs to the host OS kernel and thus only the kernel can read and write them. The cache framework can write some important metadata updates to disk, but generally it uses in memory metadata to operate. It trusts in-memory metadata fully since they are easy to maintain, fast to update and secure; no one can alter them. On the other hand, on-disk cache metadata are only written occasionally, when there is a need to. They are not read except for a cache reboot operation. This behavior prevents the existing cache frameworks from working over shared storage and thus supporting access by multiple users. Recall that shared storage is defined as a medium where changes initiated by one user are visible by others. The caches over shared storage do not fall into this category, although the cache devices are indeed shared. The cache frameworks are programmed to never read on-disk cache metadata during normal operation, and so, changes to cache block data or cache metadata done by other hosts will never be acknowledged.

To further explain this situation, and to provide an example relative to our discussion about live migration, consider the following. Let two hosts, A and B, utilize the same cache device to provide a cache for two separate processes, each running on one node. Host A and Host B each keep in-memory metadata about the state of the cache blocks

and update them properly. However, data corruption is unavoidable, since there is no coordination between the two instances of the cache framework. Sooner or later, there will be contention for the same cache block. The last one to update its contents will force its own way, while the other will silently serve wrong data in future read accesses. There is no way of knowing which cache block "belongs" to which host. This still holds even if cache frameworks write data to cache blocks and then update the cache metadata. Caches never read on-device cache metadata and load them to memory during normal operation. So, even if a cache could atomically write new data to a cache block and then update the on-disk cache metadata, it would make no difference, as long as other caches wouldn't search for this information and read it.

3.2.4 Data corruption with a shared cache

Having shown how two cache frameworks fail to cooperate over shared storage, it is easy to conduct how the live migration of a "cached instance" can lead to data corruption. Consider the two migration hosts, source and destination, keeping separate cache metadata in their own memory. Let's assume for convenience that at a given point in time, source host and destination host have exactly the same metadata in memory, and their metadata are up to date with the on-disk metadata. Since different block I/O requests are issued by source and destination host during migration in our scenario, it is straightforward to see that the two hosts will tend to modify the shared cache, each in his own way. For example, a single read request early in migration time, issued by the destination node can lead to a cache miss and to a cache allocation, possibly to a cache eviction, too. This event will go unnoticed by the source host, which could later ask the cache for the data that have been evicted. Unaware of the change in cache block state, the wrong data will be served. In a different scene, suppose the instance issues a lot of write requests during the migration. A portion of them will go through the source host cache while the rest of them will be executed on the destination host. As a result, any possibly cache block dirtying that occurs in the source host node will not be visible to the destination host cache, even if the source host node updates the on-disk metadata. Consequently, the destination cache will treat DIRTY blocks as VALID, possibly evicting them without cleaning them first, therefore producing data loss.

It must now be clear how an instance live migration, from source host to destination

host can lead to cache metadata corruption and thus data loss.

The key to avoid this catastrophe, lies within the cache metadata.

3.3 Naive Workarounds

After having taken a quick look at the aforementioned problem, one may suggest to simply drop the cache during migration. That is, to destroy the cache on the source host before a migration takes place, and then create a new cache on the destination host after the migration has completed.

Indeed this is a valid workaround, but it introduces an unacceptable performance hit. Before the cache is destroyed for the migration, all DIRTY cache blocks must be cleaned. This is a long lasting procedure, requiring high I/O throughput between cache and the backing storage. Not only will it need a lot of time, it will also slow down other requests to the SAN appliance due to contention. During the actual migration, all block I/O requests issued by the instance will be served by the backing SAN appliance imposing a severe penalty to I/O latency which may be unacceptable depending on the nature of the application. And to make matters worse, performance will continue to suffer after the migration has completed, due to a cold cache. The instance will have to repopulate the cache with its working set data; until this happens, the efficiency drop will be noticeable and irritating. It is clear now, why although this workaround is correct, it is not enough for real-world use.

3.4 An optimal solution

In this section we provide an abstract description of an optimal solution. In other words, we clearly design the properties that an option must meet, in order to be accepted.

The optimal solution must meet 3 key requirements.

1. First and obviously data consistency is of utmost importance. Instances and the data they generate is invaluable, and so data integrity is a top priority in the context of this work.

2. Second, we seek for excellent performance. We would like to benefit from the cache at all occasions and our goal is to integrate the notion of VM instance with its cache and provide uniform control.

3. Last, we are after simple and elegant solutions. An option that is difficult to implement and would require many many man-hours is not really considered as an option.

Having these three goals in mind, we can see how the workarounds of the previous section are insufficient and we now present some alternatives.

One option would be to extend an existing cache framework to implement a full coherence protocol. Although it covers goals 1 and 3, it falls short on goal 3 as it requires a lot of work. Implementing a full-blown distributed cache manager would involve developing with concurrency in mind and integrating a Distributed Lock Manager. Moreover, by introducing locks a small performance hit would be taken.

Another option would be to modify the KVM hypervisor and add hooks to enforce a notification when the actual migration is about to happen. This would enable the source host to safely fast-remove the cache and notify in turn the destination host to load the existing cache. However, the KVM project has too many moving parts and is constantly evolving. So it would be extremely difficult to keep up with frequent updates and maintain a functional patch to the upstream Linux kernel. Also, this would introduce a dependency on the specific hypervisor being used; the solution would be KVM-specific.

Ultimately, we came to conclude that we could avoid such options. They key to a more elegant solution was the observation that data corruption was triggered by actions such as cache allocations and cache evictions. In the same manner, a possible change to cache metadata could lead to data loss. As a result, it is straightforward to deduce that if we could stop such actions from happening, cache metadata would stay unchanged and data corruption would be avoided. Driven by this idea, we developed our solution, which ensures data integrity, provides excellent performance and is easy to implement.

3.5 Our design: A new cache mode

As described before, the key to avoiding data corruption is to somehow prevent cache metadata from changing while cache is being managed by both source and destination host. Taking one more step forward, it is easy to keep the same metadata state, if we don't allow any new cache allocations and cache evictions to occur. To enforce this policy, we have developed a new cache operating mode, and we have called it 'Frozen-metadata' mode. As the name suggest, the cache metadata remain unchanged while the cache operates in our mode. In Frozen-metadata mode, the set of cached blocks is not modified, and the instance is expected to have built its own working set residing in cache. New cache allocations are not allowed, neither are cache evictions. Using this technique we preserve the same cache metadata state pre-, during and post migration, across source host, destination host and on-disk cache metadata. By not permitting cache metadata to change either in-memory or on the device, data corruption is avoided.

Effectively, our new mode is much easier to implement than the other approaches. We only need an existing working cache framework and we can expand it to implement the Frozen-metadata mode. When in our new mode, the cache framework serves read and write hits directly from cache. On the other hand, cache misses always trigger read-around and write-around operations, so that no new cache allocations occur, and thus no possible cache evictions or invalidations. In other words, our new cache mode is a "hit only" or "no-allocation" mode.

Moreover, we need to clearly define what happens when a cache transitions from Write-back mode to our Frozen-metadata mode, or switches back. When we move from a Write-back cache to a Frozen-metadata cache, we must flush the in-memory metadata to the cache device. Recall that in Frozen-metadata mode we do not want any metadata updates to occur. For that reason we make sure that when we switch to Frozen-metadata mode, we sync in-memory metadata with on device metadata. While in Frozen-metadata the previous paragraph describes how we manage to avoid all metadata updates. Finally, when switching from Frozen-metadata mode back to a Write-back cache, we have to read and load the on device metadata into memory. Although it is not clear now why this is needed, it will become clear in the next chapter.

4 Implementation

In this chapter we review the development iterations we have performed, until the Frozen-Metadata mode was fully functional and working. Our goal is to allow the reader to follow our steps, and stumble upon the same obstacles that we ourselves hit. At each step, we describe the problem and present the key idea enabling to overcome each issue.

For our implementation, we have chosen to use EnhanceIO[20], since it is the most advanced and featureful cache framework available. The EnhanceIO project is free and open source software. We used the latest version available at GitHub[**?**].

4.1 Migration using WB mode

At first, let us analyze the naive scenario where an instance is live migrated using a Write-back cache. The destination host prepares to receive the instance by attaching a remote block device backed by a SAN appliance locally and also fires up the KVM hypervisor to listen for incoming data. Furthermore, the destination host node OS will setup the cache framework by pointing to an existing block device, already containing the instance cache data and metadata. As a cache load operation instructs, cache metadata are loaded into memory from the on-device metadata section. This allows the cache framework to learn about the cache state and enables it to start serving requests immediately after loading has finished. While the destination host node reads and loads cache metadata into memory, the instance is up and running on the source host node. As a result, it is highly possible that the instance is issuing block I/O requests, which are serviced by the source host node cache. This leas to modification of cache metadata in the memory

of the source host; they must change for the cache to be consistent. The problem is, if the source host node operates in Write-back mode, new cache allocations and cache evictions are allowed and these actions modify the set of cached blocks residing in cache. However, the destination host node cache never acknowledges these updates and it will continue to operate using the wrong metadata, after the migration is completed. It is clear now, how data are corrupted: future read requests that will hit the destination host node cache, will erroneously return the new cache block instead of the evicted one.

4.2 Naive Migration using FM mode

As described in the previous chapter, the key for data integrity is to maintain unchanged metadata in both source and destination host node, and also on device. To achieve this, the two nodes must switch to the new Frozen-Metadata mode during live migration. While in Frozen-Metadata mode, the set of cached blocks remain unchanged and thus no metadata update is required. Cache hits are served by cache, while cache misses lead to read-around and write-around operations, respectively.

A live migration would then work as follows. First, the source host node should switch from Write-back mode to Frozen-metadata mode. This mode switch triggers a metadata flush from memory to the cache device. Next, the destination host node loads the existing cache as warm and starts operating directly in Frozen-metadata mode. As always when loading an existing cache, the destination host node reads and loads in memory the on device metadata. After that, we have the two host nodes having the same metadata in memory, and that metadata is consistent with the on device metadata. Now the instance is allowed to issue block I/O operations either to the source host node or the destination host node. After the migration has completed, the source host node must simply drop the cache, in other words fast remove it, and the destination host node can switch from Frozen-metadata mode to Write-back mode.

However, it turns out that this naive approach of a "hit only" cache is flawed. Let's assume that the instance issues some write requests through the source host node, and the cache signals a write hit. As a result, some VALID cache blocks are updated and their state transitions to DIRTY. The source host node updates the in-memory metadata and also issues a block I/O operation to the cache disk, to update the on device metadata. The

problem is, the destination host never acknowledges these updates and it is easy to spot how data can get corrupted. The destination host will continue to consider these cache blocks as VALID and not as DIRTY, because this is what its in-memory metadata show. At some point in time, these cache blocks may need to be evicted, and when this happens, the destination host node will simple delete them, thinking they hold the same values as with the backing store. This is how updates get lost and data is corrupted.

4.3 Updating on-device metadata

The previous section made clear that write hits reveal a flaw in our Frozen-metadata mode, that looks like a dead end. On the one hand, we cannot direct the writes to disk. This would lead to a write-around operation which must be followed by a cache invalidation if the same sectors reside in a cache block, and this would make the whole point of avoiding metadata updates meaningless. On the other hand, we cannot write the cache block with new data, because that would lead to a state update, from VALID to DIRTY, and as previously noted, metadata updates are not allowed in our new mode. It looks like we must break some Frozen-metadata rules, and so we did, by allowing write hits to dirty some cache blocks. In other words, we choose to allow write operations to hit the cache, and update the cache metadata, changing some cache block states from VALID to DIRTY. It is important to note two things. First, if a cache block is already dirty, and a write hit occurs, the cache block state is unchanged, and no metadata update is required. Second, the scenario described here is probably more likely to happen, than not to happen. So we must take action and comply with this impurity in our design. As always, a dirty operation leads to an on-device cache metadata update. This means that the source host node, and the cache device will get to know about the change of some cache blocks. In order to be consistent, we must inform the destination host node, too. For that reason, we choose to read and load the on-device cache metadata when we are switching from Frozen-metadata mode to Write-back mode. By reading the on-device cache metadata, the destination host node will acknowledge any possible dirtying of cache blocks issued by the source host, and treat them properly later, for example by cleaning them before evicting them.

4.4 Atomic cache block updates

It now seems like we have a working version of our new cache mode, but unfortunately there are still flaws that we need to take care of. This time, the error lies withing on-device metadata updates. In the previous analysis, we assumed that we can update the on-device cache metadata by editing a single cache block state, but this is in fact wrong. On device metadata are just like regular block device data, and as such they are read and written in terms of blocks or sectors, typically 512 bytes in size. As a result, we cannot atomically update the state of a single cache block and write the metadata on the cache device. It is straightforward to make up an example where two blocks, initially VALID, are both being dirtied by different hosts, respectively. If the two blocks happen to lie in the same cache block device sector, then the first of the two nodes to update the cache device metadata will mark the one block as DIRTY and the other as clean, based on the in-memory metadata it owns. Then the second host node, trying to mark the other cache block as DIRTY, it will also mark the first cache block as VALID as derived by the in-memory metadata, and thus destroying the work of the first host. This is another example of cache metadata updates happening on the cache device but that are not acknowledged by other hosts. The net result is that the first cache block dirtying will be lost and data corruption will occur as described in the previous section.

4.5 Everything is DIRTY

The analysis in the previous section states that we must find an efficient way to atomically update the on device metadata, but we have found that this is not a hard requirement. The only problematic scenario is when a cache block transitions from VALID state to DIRTY. We have figured out that these updates can be avoided, if cache blocks are already in DIRTY state. This can happen in the case where the instance working set fits into cache, the workload is very write-intensive, and the cleaning operation does run very often. So the cache can be expected to have mostly dirty cache blocks instead of VALID ones. Since we cannot count on these conditions, we have decided to manually treat each cache block as DIRTY when the cache switches from Write-back mode to Frozen-metadata mode. Using this trick, we change the in-memory metadata of the two host nodes and we change any VALID cache blocks to DIRTY, while the on-device

metadata are not required to be updated. This method completes the Frozen-metadata mode and presents a working paradigm. The destination host node, which will continue to operate the cache after the live migration has completed, will suffer from a performance penalty for treating everything as DIRTY; eventually, it will have to clean all the cache blocks, although some cleanings aren't really required. However, the overhead for cleaning will be amortized over time and thus the performance hit can be mitigated. This technique is the one we have developed and tested, and as it will be shown later, results seem promising.

4.6 An alternative: locking the cache

Another approach towards atomic metadata updates is to introduce a shared lock on the cache. The lock must be shared between source and destination host node and thus it must reside on the cache device. The idea is that before a metadata update is issued, a host must acquire the lock. Acquiring and releasing the lock happens by writing a special value in a specific part of the superblock. When a host holds the lock, it may write the new metadata to the cache device knowing that no other host may access the cache metadata at the same time. After having atomically updated the metadata, the host may now release the lock by clearing a specific member of the superblock structure. Although this approach introduces a performance overhead for acquiring and releasing the lock, it may be a better alternative to treating the whole cache as DIRTY. Furthermore, special hardware support may be required in order to successfully implement this solution, for example an atomic COMPARE_AND_WRITE storage operation.

5 Integration with Ganeti

In this chapter we describe how the Ganeti cloud cluster management tool can be used in combination with our new expanded version of EnhanceIO to perform live migration of instances utilizing a block-level cache.

5.1 Ganeti ExtStorage Providers

Ganeti was initially designed to store instance disks as logical volumes using LVM[14]. Each node would have one or more physical disks, and it would create a logical group named "xenvg" on top of it, allowing Ganeti to create logical volumes as instance block devices. It is clear that an instance could only be hosted on the same node that was initially created; this was the only node having access to the instance block devices. To support instance migration and fail-over, Ganeti would employ DRBD to allow replication of data between hosts, and particularly instance disks. Borrowing from DRBD terminology, to each instance was assigned a master or primary node and a secondary or fail-over node. The instance would run on the primary node, and if the instance was to be migrated, it could only be moved to its secondary node.

Later, Ganeti was extended to support externally-mirrored shared storage. This includes two distinct disk templates. Either instance disks as regular files residing on a cluster filesystem such as NFS and Ceph, or instance images as shared block devices, typically LUNs residing on a SAN appliance. The use of a centralized storage provider offers high availability and replication features, removes the limitation of a 1:1 master-slave

setup and provides shared storage without the administrative overhead of DRBD. So, Ganeti does not need to take care about the mirroring process from one host to another, anymore.

Ganeti added the "External Storage Interface" to support such storage solutions. The Ganeti interface for any type of storage provider consists of a set of files, that is executable scripts and text files, contained inside a directory which is named after the provider. This directory must be present across all nodes so that the provider is usable by Ganeti. Of course, the external shared storage hardware should also be accessible by all nodes, too.

An `extstorage` provider must offer executable scripts able to create, resize and destroy a disk, edit the disk metadata if it is supported, locally attach a remote disk and create a local block device, detach a disk and take a snapshot of a disk. Optionally an "extstorage" provider may support opening and closing a block device, which can be used for enabling and disabling I/O to the device, respectively. Thus, the `extstorage` API exposed by Ganeti is a set of six executable scripts, which will be used to used to manage the external storage.

5.2 Our `eio_rbd` provider

The Ganeti "External Storage Interface" can easily be used to integrate a cache framework with Ganeti. We have developed our own ext provider called `eio_rbd` which complies with the Ganeti external storage provider API. The name is a concatenation of `eio`, which stands for EnhanceIO, and `rbd` which refers to a RADOS cluster and RBD images that are backed by objects. It uses the `rbd` command line tool to manage remote block devices residing in a RADOS cluster and uses our extended EnhanceIO cache framework to create and manage a host-side cache. `eio_rbd` consists of six executable scripts that will be called by Ganeti when needed. The six scripts are: `create`, `attach`, `open`, `close`, `detach` and `remove`.

CREATE

The `create` script is responsible for communicating with the RADOS cluster and requesting the creation of a new block device. The name of the new device and its size are passed as arguments to this call. Upon success the RBD image is created

and it can be mapped to a local block device. The `create` script is also responsible for creating the cache block device. In our setup, cache block devices reside in the same RADOS cluster as normal block devices, but on a different storage pool that is is only backed up by SSDs and thus it is faster. Similarly, the `create` script requests the creation of a cache block device, specifying the device name, size, and the cache pool to be created in. Overall, the `create` script is used to construct the block devices needed for the instance disks and cache.

REMOVE

The `remove` script is the dual inverse of the `create` script. It is used to destroy the block devices used as disks and as cache. It connects twice with the RADOS cluster, requesting the deletion of the cache block device and the disk block device. The same names as with the `create` script are used and upon success, the block devices are permanently deleted and their resources deallocated.

ATTACH

The `attach` script is used by a host to make a remote block device locally available. This procedure is called block device mapping and upon success an rbd image residing in a RADOS cluster can be access through a local block device, for example `/dev/rbd0`. The new block device can be used in the same way as any block device, such as given to an instance as a disk or used along with another block device to create a cache. As with the `create` script, the `attach` script also performs the map operation twice, once for the instance disk and once for the cache device. The output of the `attach` script is the name of the local block device created.

DETACH

The inverse operations are performed by the `detach` script. This script unmaps the given local block devices thus making it unavailable. The local block devices cannot be in use when this script is called, for example by a KVM instance, otherwise an error will be produced. Likewise, the cache block device cannot be in use by the host cache framework. To be sure, the `detach` script first checks if a cache exists, and if it does, it deletes it in a similar manner to the `close` script.

OPEN

The `open` script is the most complex script of our `ext` provider. It is supposed to make a block device ready for receiving block I/O requests. In our context there is no specific action to be made for enabling block I/O operations, but the `open` script is responsible for creating the cache mapping. In other words, it uses the two local block devices created by the `attach` script and combines them with a call to the cache framework. The call provides the disk block device and the faster cache block device and requests the creation of the cache. In the context of EnhanceIO, no new block device is created, rather the block device representing the disk is enhanced and after the call it successfully utilizes the cache device. The call to the cache framework also specifies the desirable cache mode of operation. For reasons that will become obvious in the following paragraphs, the `open` script also accepts one more argument, a binary flag called EXCLUSIVE. If this flag is set, then the `open` script runs in context of a single master, that is, only one host node has access to the cached block device being opened. If the flag is not set, then the `open` script assumes that the block device is being shared, for example during a live migration. In our work, exclusive access to a block device is equivalent to a Write-back cache mode, while a shared block device implies a cache operating in Frozen-metadata mode. So effectively, the `open` script is used for cache creation and cache mode switching between Write-back and Frozen-metadata modes.

CLOSE

The `close` script is the opposite of the `open` script. It is used to delete a cache mapping consisting of two local block devices. The `close` script does not require any extra arguments as the `open` script. In our work, it always issues a fast cache remove operation in order to complete fast, without waiting for a full cache clean to happen.

5.3 Workflow

After having introduced the `eio_rbd` scripts, it is now time to explore how Ganeti utilizes those scripts and in which context is each script called.

When a new instance is created, the `create` script is first called to create the block devices that will be used as instance disks. In our scenario, the script creates two block

devices, one for the actual disk and one for the cache device. Next, the `attach` script is called to make those block devices locally available, for example `/dev/rbd0` and `/dev/rbd1` in our work. Lastly, the `open` script is called by Ganeti with the argument EXCLUSIVE set, to allow the devices to receive block I/O requests. In our case, the `open` script first creates the `EnhanceIO` cache, using the two local block devices, and then signals success. The Write-back mode is used as a result to the EXCLUSIVE flag being set. This concludes how an instance is created using the `eio_rbd` ext provider.

The inverse operation, that is the removal of an instance follows the dual path. First the `close` script is called to remove the cache mapping and to disallow further block I/O requests towards the block devices. In our work, a cache fast remove is performed and so the remove operation is quickly completed. Metadata are flushed as is and no cleaning takes place; so both Valid and Dirty blocks reside in cache. After that, the `detach` script is called to unmap the remote block devices and finally the `remove` script is called and permanently deletes the RADOS images.

Sometimes, an instance needs to be powered off. This case is similar to the removal scenario, and so after the instance is shut down the `close` script is called to remove the cache mapping and to disable future block I/O operations to the device. During cache removal, the cache is not cleaned and so the cache device holds both Valid and Dirty cache blocks. Next the `detach` script is called to unmap the local block devices and then the power off operation is completed.

An off-line instance can be rebooted in any Ganeti cluster host. In this scenario, Ganeti first calls the `attach` script to map the remote block devices as local devices to the node that will host the instance. Then the `open` script is called, and since the instance already owns a cache, the script reloads the cache by reading cache metadata into memory, thus performing a warm boot, and signals success.

Finally, the most important instance operation targeted by this work is instance live migration. This procedure takes place in two hosts rather than one. Initially, the destination host calls the `attach` script to gain access to the remote block devices. Then, the source host calls the `open` script with the EXCLUSIVE flag not set, indicating "shared" mode. In our scenario, a "shared open" operation is interpreted as "switch to Frozen-Metadata mode", and so the source node switches from Write-back to Frozen-metadata, thus getting ready for the migration. The `open` script is then also called to the destina-

tion node, again with the EXCLUSIVE flag being cleared. The destination host checks to see if an existing cache mapping exists, and since it there is not, it reloads the cache using the cache device obtained from the `attach` script by reading and loading into memory the cache metadata. At this point, the two nodes are both operating in Frozen-Metadata mode, and so live migration is safe to proceed. The KVM hypervisor performs the actual transfer and when the migration is completed, it notifies Ganeti which in turn calls our `eio_rbd` scripts. After migration, the `close` script is called on the source host, and as a result the cache mapping is simply dropped. The `EnhanceIO` framework performs a fast cache remove and any metadata updates are flushed to the cache device. Then, the destination host must switch from Frozen-metadata mode to Write-back, and to do this, the `open` script is called again, with the EXCLUSIVE flag set. Our cache framework performs the mode switch and signals success. Ultimately, if everything happened as planned, live migration is completed and instance is now successfully hosted on destination node.

6 Results and future directions

6.1 Testbed

The setup we have used for testing is a private Ganeti cluster, where each node supports the KVM hypervisor and has the EnhanceIO

The testbed used in our work is a simple two node Ganeti cluster hosting a small number of Virtual Machines and a Ceph cluster offering shared block storage in two different pools, one for instance disks and one for disk caches. The Ganeti cluster nodes provide the KVM hypervisor for virtualization and also have our modified `EnhanceIO` module installed. Finally, the `eio_rbd` ext storage provider is present on the two Ganeti nodes.

The testing of our cache framework involved mode switching from write-back mode to frozen-metadata and back, while an instance is running and issuing block I/O operations. We have used the `fio`[5] load generator inside the instances to stress the block I/O datapath and we have created a number of different I/O patterns, for example sequential write-only scenarios or random reads and writes of different size. On the host level, we have used the `eio_cli` tool to switch between cache modes and the ganeti command line tools to create, destroy and live-migrate the hosted instances.

In order to reason about the correctness of our solution, we have developed a pair of complementary tools; one in the kernel-space where `EnhanceIO` lives, and one for the user-space. The former tool reads the in-memory cache metadata from the `EnhanceIO` address space and exports them to user-space for further processing. This tool is actually a part of our extened `EnhanceIO` cache framework, and it is mandatory for real-

time cache metadata inspection. The latter tool translates the binary data exported from kernel-space into human-readable form. Effectively, this tool is aware of the data structures used in-kernel by the `EnhanceIO` framework and uses this information to print for each cache block, which disk block is cached there and the state of the cache block; for example VALID or DIRTY.

6.2 Results

As a result, we have been able to inspect the state of cache metadata at any given point in time, like immediately before or after a mode switch, during heavy I/O load from the instance or during a live migration.

We have tested the transitions between a write-back cache and a frozen-metadata cache and using our metadata extraction tool we have verified that frozen-metadata mode behaves as planned. The set of cached blocks residing in cache have remained unchanged, thus no allocations or evictions have taken place. The state of those blocks have either stayed the same or transitioned to a predictable state, for example from VALID to DIRTY.

Consequently, we have concluded that metadata corruption can be avoided using our approach and thus data loss can be evaded. This promising result clearly confirms our hypotheses and validates our work as a whole.

6.3 Future Work

Our work have created a need for further development and improvement of cache frameworks.

Definetely we need to extensively test and reason about the correctness of our approach under a very large spectrum of deployment scenarios and usecases, such as different I/O workloads produces by instances.

After being heavily tested, we will deploy our cache framework in production environments and we will evaluate the performance results, providing helpfull feedback for additional improvements.

Finally, we plan to add supplementary features to our cache framework, that enables the administrator to enforce different policies per instances trading-off between the importance of data and the acceptable performance loss.

Bibliography

[1] A next-generation platform for petabyte-scale storage. https://www.redhat.com/en/technologies/storage/ceph. Accessed: 2015-11.

[2] Amazon Web Services (AWS) - Cloud Computing Services. https://aws.amazon.com/. Accessed: 2015-11.

[3] Ceph Filesystem. http://docs.ceph.com/docs/master/cephfs/. Accessed: 2015-11.

[4] Docker - Build, Ship, and Run Any App, Anywhere. https://www.docker.com/. Accessed: 2015-11.

[5] Flexible I/O Tester. https://github.com/axboe/fio. Accessed: 2015-11.

[6] Free VMware vSphere Hypervisor, Free Virtualization (ESXi). https://www.vmware.com/products/vsphere-hypervisor. Accessed: 2015-11.

[7] ganeti - Cluster-based virtualization management software - Google Project Hosting. https://code.google.com/p/ganeti/. Accessed: 2015-11.

[8] Ganeti 2.15.1 documentation. http://docs.ganeti.org/ganeti/current/html/. Accessed: 2015-11.

[9] Ganeti Home Page. http://www.ganeti.org/. Accessed: 2015-11.

[10] Google Cloud Computing, Hosting Services and Cloud Support - Google Cloud Platform. https://cloud.google.com/. Accessed: 2015-11.

[11] History of Virtualization. http://www.everythingvm.com/content/history-virtualization. Accessed: 2015-11.

[12] Kernel Virtual Machine. http://www.linux-kvm.org/page/Main_Page. Accessed: 2015-11.

[13] LinuxContainers.org Infrastructure for container projects. https://linuxcontainers.org/. Accessed: 2015-11.

[14] LVM is a Logical Volume Manager for the Linux operating system. http://tldp.org/HOWTO/LVM-HOWTO/. Accessed: 2015-11.

[15] Mark Fasheh, OCFS2: Oracle Clustered File System, Version 2 , Proceedings of the 2006 Linux Symposium, July 2006, pp. 289–302. https://www.kernel.org/doc/ols/2006/ols2006v1-pages-289-302.pdf. Accessed: 2015-11.

[16] OpenStack Open Source Cloud Computing Software. https://www.openstack.org/. Accessed: 2015-11.

[17] Pithos+. https://okeanos.grnet.gr/services/pithos/. Accessed: 2015-11.

[18] QEMU Open Source Processor Emulator. http://wiki.qemu.org/Main_Page. Accessed: 2015-11.

[19] Schmuck, Frank; Roger Haskin (January 2002). "GPFS: A Shared-Disk File System for Large Computing Clusters" (pdf). Proceedings of the FAST'02 Conference on File and Storage Technologies. Monterey, California, USA: USENIX. pp. 231–244. ISBN 1-880446-03-0. Retrieved 2008-01-18. http://www.usenix.org/events/fast02/full_papers/schmuck/schmuck.pdf. Accessed: 2015-11.

[20] STEC EnhanceIO SSD Caching Software. https://github.com/stec-inc/EnhanceIO. Accessed: 2015-11.

[21] Synnefo. https://www.synnefo.org/. Accessed: 2015-11.

[22] The History of Virtualization - Supercomputers and Mainframes. http://www.servethehome.com/virtualization-long-history. Accessed: 2015-11.

[23] The Xen Project, the powerful open source industry standard for virtualization. http://www.xenproject.org/. Accessed: 2015-11.

[24] Wikipedia : ZFS. https://en.wikipedia.org/wiki/ZFS. Accessed: 2015-11.

[25] Wikipedia: Cloud computing. https://en.wikipedia.org/wiki/Cloud_computing. Accessed: 2015-11.

[26] Wikipedia: Container (virtualization). https://en.wikipedia.org/wiki/Operating-system-level_virtualization. Accessed: 2015-11.

[27] With long history of virtualization behind it, IBM looks to the future. http://www.networkworld.com/article/2254433/virtualization/with-long-history-of-virtualization-behind-it--ibm-looks-to-the-future.html. Accessed: 2015-11.

[28] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008.

[29] Wikipedia. Timeline of virtualization development — wikipedia, the free encyclopedia, 2015. [Online; accessed 22-November-2015].