Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Υλοποίηση ενός Μηχανισμού Προσωρινής Αποθήκευσης Δεδομένων με Ο(1) Μέση Πολυπλοκότητα για ένα Κατανεμημένο Σύστημα Αποθήκευσης Αντικειμένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΕΞΙΟΣ ΠΥΡΓΙΩΤΗΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 9999

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής
και Υπολογιστών

Υλοποίηση ενός Μηχανισμού Προσωρινής Αποθήκευσης Δεδομένων με O(1) Μέση Πολυπλοκότητα για ένα Κατανεμημένο Σύστημα Αποθήκευσης Αντικειμένων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΑΛΕΞΙΟΣ ΠΥΡΓΙΩΤΗΣ

Επιβλέπων : Νεκτάριος Κοζύρης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 9η Σεπτεμβρίου 9999.

.....	.....	.....
Νεκτάριος Κοζύρης Καθηγητής Ε.Μ.Π.	Νικόλαος Παπασπύρου Επίκ. Καθηγητής Ε.Μ.Π.	Δημήτριος Φωτάκης Λέκτορας Ε.Μ.Π.

Αθήνα, Σεπτέμβριος 9999

Αλέξιος Πυργιώτης

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

TODO: Φτιάξε την ελληνική περίληψη και τις λέξεις κλειδιά

Λέξεις κλειδιά

Λέξη-κλειδί 1, λέξη-κλειδί 2, λέξη-κλειδί 3

Abstract

The storage service of a cloud infrastracture is a very performance critical part. Hard disks are usually inadequate in providing the performance that is needed for large deployments and the storage engineers must commonly resort to other ways to counter these issues. This thesis presents cached, an in-memory caching mechanism with average O(1) complexity, that aims to improve the performance of Archipelago, a distributed storage service. Moreover, this thesis also introduces synapsed, a complementary tool that paves the way for the creation of a fully distributed cache. Early performance evaluations look promising and show that cached can provide up to 400% speedup over the current Archipelago performance.

Key words

synnefo, okeanos, rados, archipelago, cached, cache, storage, ram, replication, synapsed

Ευχαριστίες

TODO:

Αλέξιος Πυργιώτης,

Αθήνα, 9η Σεπτεμβρίου 9999

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-*-*, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Σεπτέμβριος 9999.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The racing track was clear and the computer hardware companies were tentatively positioning themselves on the starting blocks, on the April of 1965. Gordon E. Moore was holding the starter pistol, a paper where he famously stated that:

> ”The complexity for minimum component costs has increased at a rate of roughly a factor of two per year. Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.”[15]

Until that year, the integrated circuits (or microchips, as they are commonly called) were used prominently in embedded systems. Around the time the paper was published however, they were also starting to being adopted by computer manufacturers. The replacement of vacuum tubes with microchips marked the passage of the mainframe computer to the minicomputer and on to the personal computer that we know today.

Whether Moore’s statement was a very accurate prediction of the future of microchips or a self-fulfilling prophecy that hardware companies used to market their products, is unsure. What is sure though is that his statement propelled the development of technology in general, since what became later on as a *”law”* has been applied to numerous other technologies, unrelated to microchips such as the pixels of a camera or network capacity. Wherever Moore’s law was applicable, the industry would excel itself to adhere as much as possible to the projected growth, be it in microchip density, pixel density or performance.

Storage components, e.g. Hard Disk Drives (HDDs), Random Access Memory (RAM) and Flash Memory, had also entered the race and their capacity has been increasing ever since. In the performance track however, hard disks seem old and gasping to catch up the much faster RAM and CPU caches. For decades now, their sub-par performance has been the bottleneck of every IO-intensive application and the headache of storage designers [4].

As exaggerated as this might seem, their limitations have shaped the way storage is built; hardware solutions such as the RAID technology, battery-backed volatile memory in large servers and software solutions such as the Linux’s page cache, memcached, bcache, are all notable examples which show that there is a tremendous effort that is being invested in sidestepping hard disks and finding alternative methods to store data.

The HDD’s industry answer to this is the continuous drop of their prices. In 2011, HDDs reached their all-time low price of $0.053/GB [11]. Moreover, the emerging movement of greener data centers has benefited hard disks, since their lower energy costs than RAM is attractive to enterprises. Yet, for how long can the HDD industry keep lowering their costs to mitigate their lack of performance?

The answer came very fast and unfortunately in a tragic way. The end of July of 2011 marked the beginning of a 6-month turmoil for Thailand, with a flood that was described as ”the worst flooding yet in terms of the amount of water and people affected” [8]. The hard disk industry also suffered a huge hit due to the fact that 25% percent of the global hard disk production was from factories in Thailand, that were largely affected by the flood.

The result was an overnight 40% percent increase of hard disk prices. The reasons behind this increase were in one part to compensate for the flood damages and in another part to seize the opportunity to increase the profit margins of the two biggest producers, Western Digital and Seagate, from 6% and 3% to 16% and 37% respectively [20].

The timing could not have been worse for the HDD industry. The price increase led indirectly to the introduction of the more expensive but faster SSDs to the enterprise world. Their vast price drop [26, 23] in the last few years has made them viable candidates at least for peripheral storage tasks such as journaling and caching, and has led experts to consider them as the successors of HDDs.

On the other hand, HDDs are unable to retort performance-wise and can only marginally improve their performance. As their rotational speed approaches the speed of sound, their production will be rendered at best difficult, and their heat generation, power consumption and lack of long-term reliability will make their adoption prohibitive [21, 22].

Our prediction is that in some decades from now, when the dust will settle, the data centers will probably migrate from HDDs to SSDs. Till date however, the storage landscape is baffled with uncertainty as the tug-of-war between SSDs and HDDs is still at large. Moreover, besides SSDs, there are various other flash memory types, such as the IOdrive of Fusion IO, that are being utilized in performance-intensive environments, albeit for higher prices. Besides hardware solutions, there have also been developed various caching and buffering techniques to increase the performance of databases and storage in general.

To sum up, the current storage landscape provides the storage designer with various choices, each of which has its own merits and disadvantages. It is up to the storage designer to weigh these choices and implement the solution that fits the most to the profile of storage he/she builds and the budget of the storage service.

1.1 Thesis motivation and background

The motivation behind this thesis emerged from concerns about the storage performance of the Synnefo [1] cloud software, which powers the ~**okeanos** [2] public cloud service [12]. We will briefly explain what ~**okeanos** and Synnefo are in the following paragraphs.

~**okeanos** is an IaaS (Infrastructure as a Service) that provides Virtual Machines, Virtual Networks and Storage services to the Greek Academic and Research community. It is an open-source service that has been running in production servers since 2011 by GRNET S.A. [3]

Synnefo[13] is a cloud software stack, also created by GRNET S.A., that implements the following services which are used by ~**okeanos** :

- *Compute Service*, which is the service that enables the creation and management of Virtual Machines.

[1] www.synnefo.org/
[2] https://okeanos.grnet.gr/
[3] Greek Research and Technology Network, https://www.grnet.gr/

- *Network Service*, which is the service that provides network management, creation and transparent support of various network configurations.
- *Storage Service*, which is the service responsible for provisioning the VM volumes and storing user data.
- *Image Service*, which is the service that handles the customization and the deployment of OS images.
- *Identity Service*, which is the service that is responsible for user authentication and management, as well as for managing the various quota and projects of the users.

This thesis will deal exclusively with the Storage Service of Synnefo and more specifically with the part that handles the VMs' data and volumes, which is called Archipelago and is presented in Chapter 3.

As we have mentioned at the start of this section, the motivation behind this thesis was the unsatisfactory performance of Archipelago. More specifically, we measured the performance of VMs with and without caching enabled on the host and the results showed that host-side caching improved greatly the Archipelago performance.

Since Archipelago had no caching mechanisms and data were committed directly to a replicated object storage (see more in Section 3.4) that was using hard disks, we concluded that this was the main reason behind the performance issues of Archipelago and decided to implement a caching system that would use a much faster storage medium.

As a result, this thesis presents the implementation of an in-memory caching system with average-case O(1) complexity and documents all the design decisions and the thinking process that has led to this implementation.

1.2 Thesis structure

The thesis is organized as follows:

Chapter 2:
We provide the necessary theoretical background for the concepts and entities that are being discussed throughout the thesis.

Chapter 3:
We present the architecture of Archipelago and explain how Archipelago handles I/O requests. Moreover, we provide information about RADOS, one of the storage backends of Archipelago, as well as sosd, an Archipelago component that has been created to communicate with RADOS and has been the subject of a previous CSLAB thesis [27].

Chapter 4:
We explain three of the most discussed concepts in the cloud world; scalability, tiering and caching, and depict how they can be used in a real-life scenario. Then, we present the most popular solutions for increasing the storage performance of an application, such as bcache, memcached, flashcache. Finally, we weigh their pros and cons explain why they are inadequate for our purposes.

Chapter 5:
We explain the design of cached and the building blocks that is consisted of (xcache, xworkq, xwaitq). Moreover, we illustrate how cached handles I/O requests.

Chapter 6:

We present the cached implementation, such as the structures and methods that we have used, in the form of code snippets. Furthermore, we accompany them with the necessary commentary.

Chapter 7:

In this chapter, we benchmark the sosd and cached peers under various scenarios and evaluate their performance. Also, we present the methodology behind our benchmarks and the specifications of the test beds.

Chapter 8:

We present synapsed, a complimentary component to cached, whose purpose is to transfer Archipelago requests over the network and allow cached to run in other nodes than the host.

Chapter 9:

We provide some concluding remarks about our thesis and assess in what extend has it managed to achieve the goals that where set. Also, we discuss our plans for future improvements and deeper integration with Archipelago.

Chapter 2

Necessary theoretical background

In this chapter, we will explain the main concepts and mechanisms that are used by Archipelago as well as our implementation. The concepts that will be discussed are basic as far as Operating Systems are concerned and should pose no comprehension difficulties to a reader with elementary background on the subject.

More specifically, Section 2.1 discusses what is multithreading and lists its advantages and disadvantages. Section 2.2 introduces the Interprocess Communication mechanisms that are employed by Linux and concentrates on the ones that are used in Archipelago and our implementation. Finally, Section 2.3 explains the various concurrency control methods that exist to synchronize threads or processes.

2.1 Multithreading

Mutlithreading is a programming concept that has been the subject of research long before the emerge of SMP systems [1] . More specifically, temporal multithreading has been introduced in the 1950s whereas simultaneous multithreading (SMT), which is the current invocation of multithreading programming, was first researched by IBM in 1968[16].

The difference between these two types is the number of threads that can run simultaneously on the system. For simultaneous multithreading, there are commonly more than one threads that can run in parallel, whereas for temporal multithreading, there is only one. This means that the running thread must be descheduled to let the other threads run. Temporal multithreading was an old concept that predates SMP systems and this limitation mirrors the limitations of the hardware of that era.

Before threads, programs could utilize the concurrency of SMP systems using forked processes that would communicate with each other. The introduction of threads did not render this practice obsolete, but instead provided an alternative technique to speed up applications.

Threads and process have some fundamental differences, which are shown in the following list:

- Threads are always parts of a process, whereas processes are independent from each other and may only have a parent-child connection between them.
- Forked processes have their own address space and resources, which are inherited by the parent process with CoW semantics. Multiple threads, on the other hand, usually share the same memory and resources with the other threads in the same process.

[1] Symmetric multiprocessing systems, commonly systems with multiple processors

From the above differences, we can see that there are no clear advantages of a multithreaded approach over a multiprocess one. To better demonstrate our point, we will present the advantages and disadvantages of multithreading programming in the following lists:

The advantages are:

- Context switching is generally faster between threads, mostly due to the fact that the TLB [2] cache does not need to be flushed. The TLB cache misses are expensive and are avoided as much as possible[2].
- Sharing data between threads is easier, due to the fact that they use the same memory by default.

Whereas the disadvantages are:

- Processes are more isolated than threads, which means they are guarded against two things: i) thread-unsafe functions and ii) data corruptions, which if they happen to one thread they bring the whole process to a halt.

Regardless of the chosen method, at some point the programmer will have face two of the biggest challenges of multithreading/multiprocess programming; interprocess communication, which is discussed in Section 2.2, and concurrency control, which is discussed in Section 2.3.

2.2 Interprocess Communication - IPC

Interprocess Communication is a concept that predates the SMP systems that we all use nowadays. It is a set of methods that an OS uses to allow processes and threads to communicate with each other. Archipelago for example, uses various IPC methods to synchronize its different components.

The full list of Linux's IPC methods is presented below:

- *Signals,* which are sent to a process to notify it that an event has occurred.
- *Pipes,* which are a one-way channel that transfers information from one process to another.
- *Sockets,* which are bidirectional channels that can transfer information between two or more processes either locally or remotely through the network.
- *Message queues,* which is an asynchronous communication protocol that is used to exchange data packets between processes.
- *Semaphores,* which are abstract data types that are used mainly for controlling accesses on a same resource.
- *Shared memory,* which is a memory space that can be accessed and edited by more than one process.

We will concentrate on the following IPC methods: i) signals, ii) sockets and iii) shared memory, since these are the methods that Archipelago and our implementation use.

[2] Translation Lookaside Buffer, a hardware cache that speeds up the translation of virtual addresses to physical RAM pages.

2.2.1 Signals

Signals are notifications that are sent to processes and can be considered as software interrupts. The signal's purpose is to interrupt the execution of a process and inform it that an event has occurred.

Given that there more than one events and exceptions that can occur in a system, there are also various signals that match to each one of these events. For more information about the signals that Linux supports as well as the conditions on which they are raised, the reader is prompted to consult the man pages for signal(7) or read the POSIX.1-1990, SUSv2 and POSIX.1-2001 standards.

Moreover, the above standards dictate the standard behavior of a process when a signal is received. The standard actions that a process can take, fall roughly in the following categories:

- ignore the signal,
- pause its execution,
- resume its execution or
- stop its execution and/or dump its core

Finally, a process is not limited to this set of actions. It can instead do one of the following things for each signal, with the exception of SIGKILL and SIGSTOP signals:

- ignore the signal
- block the signal, which is part of the Archipelago IPC and its usage is described in Section ?
- install a custom signal handler function, which essentially passes the signal handling task to the process.

2.2.2 Sockets

Sockets are a bidirectional means of sending data between processes. The processes can be in the same host but most commonly, they are in remote hosts and the data are sent over the network. Furthermore, from all the IPC methods that we have described above, sockets are the only method that enables remote communication.

There are many socket implementations for different purposes, which are divided in several communication domains, most of which are rather obscure. The three communication domains, however, that are supported by most UNIX and UNIX-like operating systems are:

- *IPv4* domain, which allows communication between processes over the Internet Protocol version 4 network.
- *IPv6* domain, which allows communication between processes over the Internet Protocol version 6 network.
- *UNIX* domain, which allows communication between processes in the same host

The above three communication domains are further divided in two types, based on the transport layer protocol that they use.

- *Stream sockets*, which use the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP),
- *Datagram sockets*, which use the User Datagram Protocol (UDP),

The TCP/UDP protocols are only one layer out of the four layers of the TCP/IP protocol stack that the RFC 1122[1] defines, and we will explain them in detail in the following sections. Although a thorough explanation of the TCP/IP protocol stack is out of the scope of this thesis and is not needed to understand the following sections, we will provide a brief explanation of it for the sake of completeness.

The TCP/IP protocol stack is the basis for the World Wide Web and the most used form of networking. It specifies all the stages of the data processing that need to happen in various levels and entities, such as operating systems, network cards, routers etc. in order to connect two machines over the network. For this reason, the data that are sent are encapsulated in layers, which can be seen in Figure 2.1.

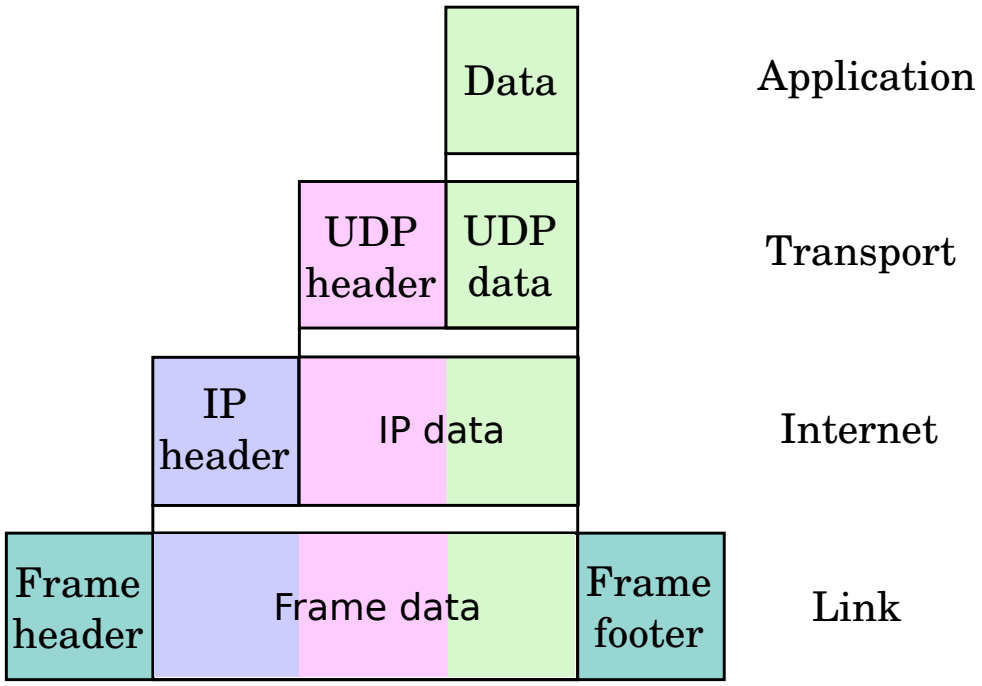

Figure 2.1: Data encapsulation for a UDP packet

We now continue with an presentation of TCP and UDP protocols.

TCP

The Transmission Control Protocol is connection-oriented, i.e. it provides unique connection between two sockets, and has the following key features:

Reliability The data will arrive to the receiver as a whole, or they will not arrive at all. In the latter case, the receiver may receive sparious packets but it will not acknowledge them until it has received all of them.

Ordered transfer The data will arrive in the same order that they were sent.

Error-checking The data are checksummed to allow the receiving end to check if there was any data corruption.

Rate-limiting When the receiver accepts packets with slower rate than the sender, the sender will adjust its rate to ensure packet delivery and less congestion.

Byte-stream The data that are sent do not have a boundary.

UDP

The User Datagram Protocol on the other hand is a much simpler protocol. It is used to send *datagrams*, which are basically extended IP packets with some extra features. The UDP protocol has the following differences from TCP:

- It is connectionless, meaning that the socket can receive requests from anyone.
- It provides no guarantees about the delivery of the messages.
- The messages can arrive in other order than the one they were sent.
- There is no rate-limiting, meaning that the congestion control must be handled in the application level.
- It cannot send streaming data since datagrams are bounded.

The UDP protocol is often preferred over TCP by applications that value speed over data loss (e.g. video streaming applications) due to its low overhead.

2.2.3 Shared memory

When two or more processes share the same memory segment, they can exchange data by placing it in a region of the segment. The data then becomes instantly visible to the other processes too, since their page-table entries for this segment point to the same physical RAM pages.

A popular way of mapping shared memory to a process's address space, which is also used in Archipelago, is with POSIX `mmap()`. There are two mapping types of mapping:

- *Private mapping*, in which case the mapping contents will not be visible to other processes that have mapped the same file and
- *Shared mapping*, in which case the mapping contents will be visible to all processes that map this file and changes to the mapping will be propagated to the shared memory.

Finally, an issue with mappings is that the start of the shared memory is not always mapped in the same virtual address for all processes. For this reason, when processes want to share data, they should not pass direct pointers to them, but relative pointers (i.e. offsets) from the start of the segment, which are common for all processes and can be translated to the correct direct pointers.

2.3 Concurrency control

Concurrency control is the set of methods that a program uses to ensure that concurrent accesses to the same data will leave them in a consistent state.

There are several techniques that are used for concurrency control and are listed below:

- *Spinlocks*, which are locks that protect a critical segment. Typically, a thread acquires a lock at the start of the critical segment and releases it at the end of it. Threads that are waiting for the lock essentially ”spin”, i.e. they busy-loop until the lock is released.
- *Mutexes*, which are locks that protect a critical segment in the same fashion as spinlocks. Their difference from spinlocks, however is that if a thread cannot get the lock, it will block instead of busy-loop.
- *Semaphores*, which are also an IPC method. In concurrency control context, they are abstract data types that restrict the number of simultaneous accesses to a resource or a critical segment. When the number of times is one (1), they essentially degenerate to mutexes, with the main difference that they have no concept of an owner.
- *Atomic operations*, which are hardware-assisted operations whose purpose is to atomically update a value as fast as possible. The atomicity is usually achieved with implicit hardware locks on the bus or cache-line. Atomic operations come in many flavors such as ”add-and-fetch”, ”compare-and-swap” etc.

Concurrency control - and locking in particular - have three important caveats that the programmer needs to know before he/she decides on the techniques that will be used:

Lock overhead
Lock overhead is the overhead that the locking mechanism introduces. For example, semaphores are a mechanism with big overhead, since they must be read and written to using system calls. If the critical segment they protect is simply the update of a variable, then the programmer is probably better off using a spinlock or atomic operations.

Lock contention
Lock contention can be considered as the overhead of the coarseness of the lock. There is contention for a lock when it is requested by many threads, to the point that the waiting time is longer than the execution time. This has a big performance impact to the implementation, since threads may consume their scheduled time spinning until they acquire a lock, or sleeping while they could do something more useful. The solution to this problem is commonly to redesign the locking scheme in order to break such locks into smaller ones.

Deadlocks
Deadlock is a situation in a multi-lock scenario where each process in a group of processes needs to acquire a lock which is held by another process in the same group. Since no process can continue, the operation of the group is essentially stalled. As a rule of thumb, the circular dependency of the locks can break if the locks are acquired in a predefined order. This however is only possible in less complex scenarios and in general, a more well-thought design is required.

Chapter 3

Archipelago

3.1 Overview

Archipelago is the Volume Service of the Synnefo cloud software. It is responsible for creating Copy-on-Write, snapshottable volumes for VMs. Archipelago can be considered as a storage layer (see Figure 3.1) that is positioned between the VM's block device, and the underlying storage.

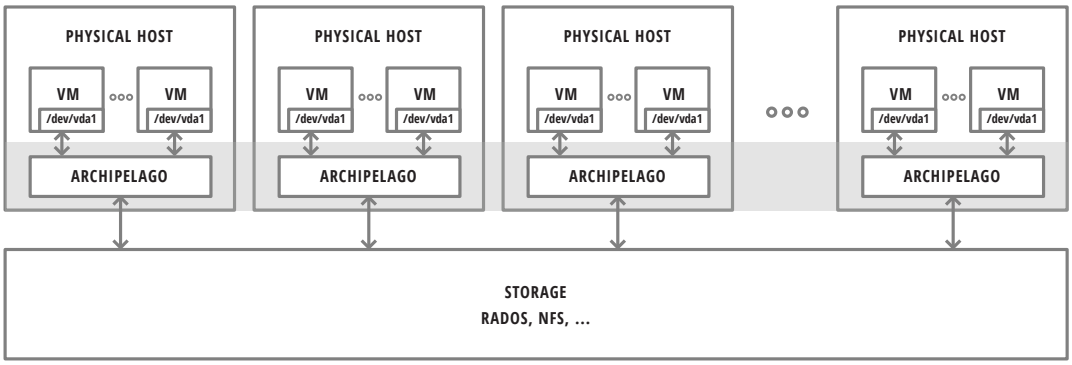

Figure 3.1: Archipelago overview

Archipelago has the following objectives:

- Thinly provision volumes to VMs with zero data movement.
- Snapshot VM volumes and use them as system images with, again, zero data movement.
- Allow VM migrations between Archipelago nodes with no restrictions.
- Be agnostic to the actual storage backend used.

3.2 Architecture

Archipelago has a modular architecture, which allows it to categorize its operations and assign them to distinct components. The IPC between these components, which are called **peers** in the Archipelago dialect, is facilitated by XSEG.

XSEG can be considered as the kernel of Archipelago and is a custom mechanism that: i) defines a common communication protocol for all peers, regardless of their type (userspace/kernelspace, singlethreaded/multithreaded) and ii) builds a shared memory segment, where peers can share data using zero-copy techniques. The above are provided to the peers by the `libxseg` library.

In Figure 3.2, we present the architecture of Archipelago. Moreover, we show the Archipelago peers, the communication channels between them and we briefly explain the operations that they are responsible for.

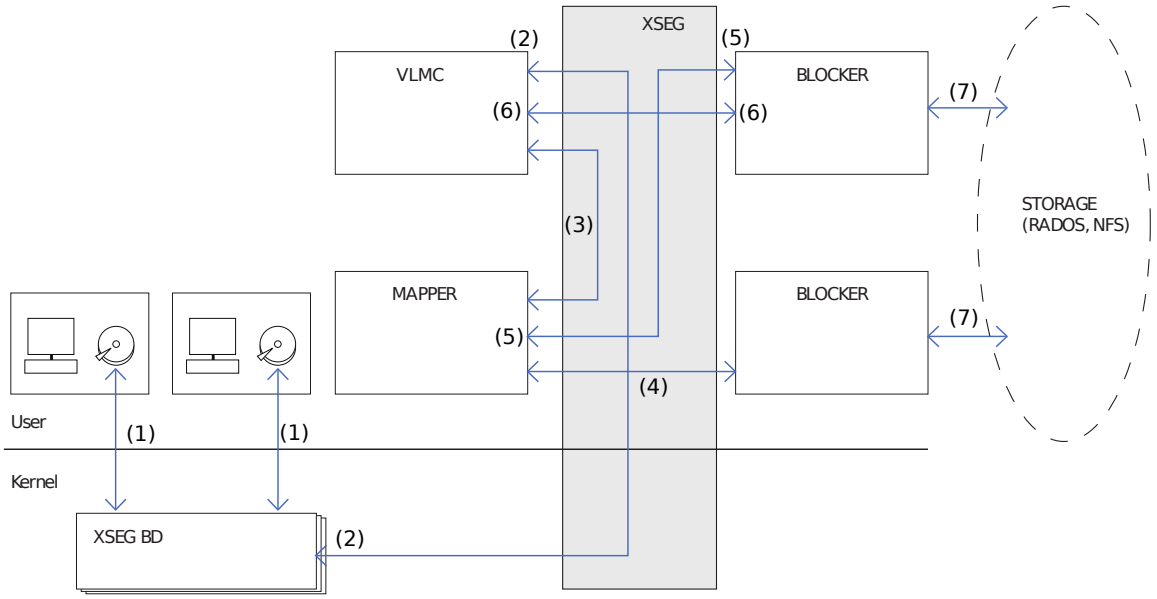

Figure 3.2: Archipelago components

XSEG Block Device (xsegbd)
xsegbd is a kernel module that exposes a VM's volume as a block device. For each VM, Archipelago registers an xsegbd block device. This block device provides the entry point for the requests that enter the Archipelago layer.

VoLuMe Composer Daemon (vlmcd)
vlmcd accepts requests from the various xsegbd devices and translates them to object requests, with the help of mapperd.

Mapper Daemon (mapperd)
mapperd is responsible for the mapping of volumes to objects. This means that it must tackle a broad set of tasks such as knowing the objects that a volume consists of, cloning and snapshotting volumes and creating new ones.

File Blocker Daemon (blockerd)
blockerd is not a specific entity but a family of drivers, each of which is written for a specific storage type. File blockers have a single purpose, to read/write objects from/to the storage. Currently, there are file blockers for NFS and the RADOS object storage.

3.3 Requests

We have already explained that Archipelago manipulates the requests that reach to the VM's block device. There are two main reasons behind this approach:

1. Since Archipelago has its own CoW policy, the only way to enforce it is to translate these requests to object requests and then consult its mappings to find the actual data.

2. In order to be able to use different storage backends, the requests must be translated to native requests for the respective backend.

The request flow can be seen in Figure 3.2. We will explain what happens in every step of the process, with the same order as they are presented in Figure 3.2:

1. The VM issues an I/O request, which is essentially a block request. The hypervisor forwards this request to xsegbd, the block device that is attached to every VM.

2. The xsegbd's purpose is to bring this requests to the user level, by copying its data to the shared memory segment and sending an XSEG request to vlmcd

3. Once the vlmcd receives the request, it translates it to an object request. The object name can be constructed from the volume name and the dividend of the request offset divided by the object size. Once vlmcd knows the target object, it asks the mapper if this object is unique or a copy of another object.

4. The mapperd generally has the mappings of an object cached in its memory, so it can respond quickly. In the rare event that they are not cached, it queries a special purpose blocker to retrieve them.

5. At this point, mapperd has two courses of action:

   i) it can either respond to vlmcd the correct object that it must read from or write to (3) or

   ii) perform a CoW operation (5)->(4)->(3)

   The CoW operation will be performed if the VM has asked to write on an object which is a copy of another object. In this case, mapperd will send a copy request to the data blocker and will create a new object with the same data (5). Then, mapperd will update its mapping and store it using the blocker for mappings (4). Once all of the above have finished, mapperd will respond to vlmcd the name of the copied object (3).

6. At this point, vlmcd can send the request for the correct object to the data blocker.

7. Finally, the blocker will read/write the requested data to the storage type that it has been written for.

Once the storage backend completes the request, the data blocker notifies the vlmcd, which in turn notifies the xsebd, which finally notifies the VM.

3.3.1 Request polling

There is a part that has been omitted in the previous section and that is how peers send and receive requests. This is a rather intricate aspect of the Archipelago IPC, but we will explain it here since it is needed to understand mainly the synapsed implementation.

To begin with, one of the features of the shared segment is that it provides a number of **custom ports**. These ports are similar to network ports, in the sense that peers bind them and can listen for requests through them. Each port has essentially two queues: i) a **request queue**, where new requests are accepted, and ii) a **reply queue**, where replies to requests are received.

Thus, when a peer wants to send a request to another peer, it merely needs to enqueue that request to the request queue of the peer's port. Likewise, when a peer responds to a request that was sent by another peer, it will enqueue that request in the reply queue of that peer.

Moreover, peers are notified about new requests via signals. When a peer receives a signal, it wakes up and then accepts the request. However, context switching is expensive, so once the peer has served a request, it will not sleep immediately. Instead, it will iterate its ports for new requests for a small number of cycles and if no request is received until then, it will sleep.

3.4 RADOS and sosd

As we have mentioned above, Archipelago is not restricted to a storage backend and can work with any backend for which a blocker can be created. There is one backend however that provides many important features that are used to make Archipelago more scalable and reliable.

This backend is RADOS[25], which is the object store component of the Ceph [1] system. Ceph is a free distributed object store and file system that has been created by Sage Weil for his doctoral dissertation [24] and has been supported by his company, Inktank, ever since.

The architecture of Ceph can be seen in Figure 3.3. [2]

Ceph utilizes RADOS to achieve the following:

- *Replication*, which means that there can be many copies of the same object so that the object is always accessible, even when a node experiences a failure.
- *Fault tolerance*, which is achieved by not having a single point of failure. Instead, RADOS uses elected servers called **monitors**, each of which have mappings of the storage nodes where the objects and their replicas are stored.
- *Self-management*, which is possible since monitors know at any time the status of the storage nodes and, for example, can command to create new object replicas if a node experiences a failure.
- *Scalability*, which is aided by the fact that there is no point of failure, which means that adding new nodes theoretically does not add any communication overhead.

In a nutshell, RADOS consists of the following components:

[1] ceph.com

[2] Picture retrieved from the official website on September 24, 2013. All rights go to the respective owners.

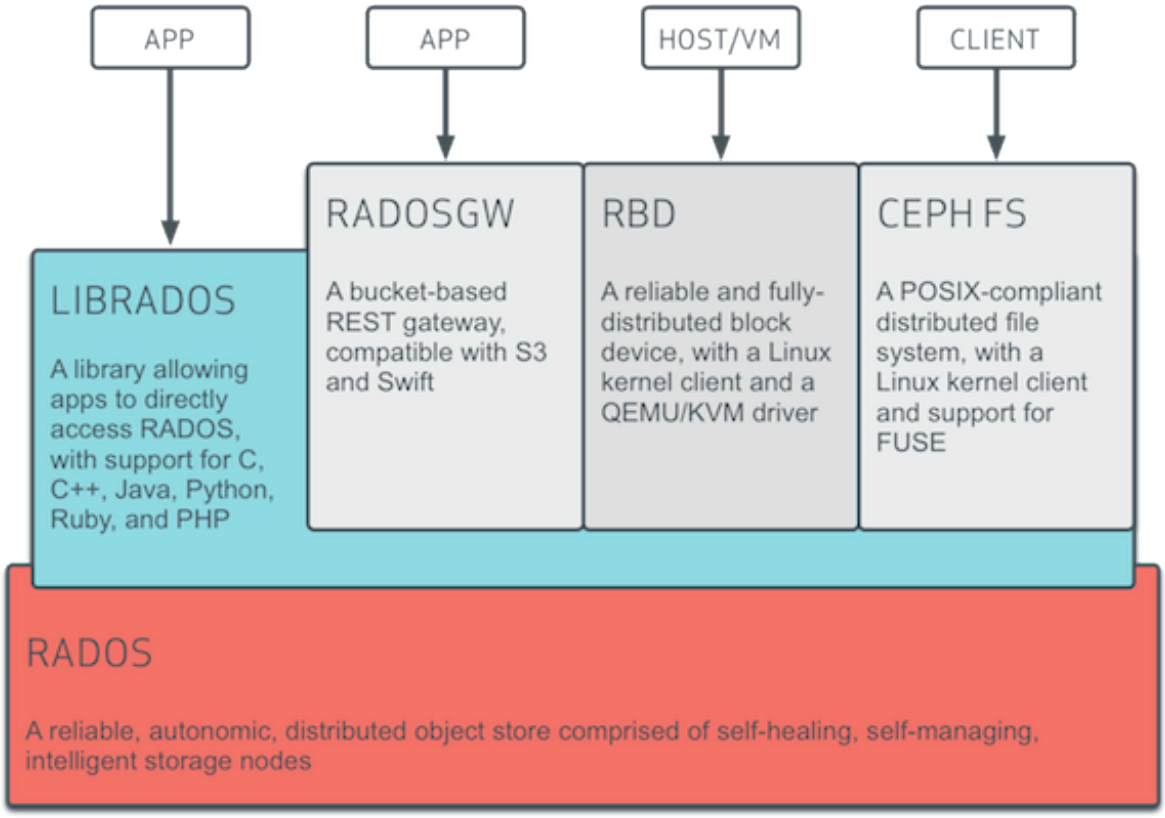

Figure 3.3: Ceph architecture

- *object store daemons*, which are userspace processes that run in the storage backend and are responsible for storing the data.
- *monitor daemons*, which are monitoring userspace processes that run in an odd number of servers that form a Paxos part-time parliament[14]. Their main responsibility is holding and reliably updating the mapping of objects to object store daemons, as well as self-healing when an object store daemon or monitor daemon has crashed.

The mapping of objects to object store daemons is done indirectly with the help of the *CRUSH map* and *placement groups (pgs)*. Generally speaking, placement groups can be considered as logical buckets where more than one values can be assigned to.

RADOS uses placement groups in the following manner: On initialization, it is configured to have a fixed number of placement groups and a range of values assigned to each of them. When it is asked to find an object, the object name is hashed and its hash value is used to find in which placement group it belongs. The relationship between placement groups and object store daemons is stored in CRUSH maps that each monitor daemon holds. This way, the objects are pseudorandomly distributed to the storage backend, which in turn implicitly guarantees load balancing.

There are various entry points for RADOS, as is evident from Figure 3.3. Archipelago uses librados for I/O requests and more specifically, a blocker called sosd[27] has been created by Filippos Giannakos to facilitate the communication between RADOS and Archipelago.

Chapter 4

Scalability, Tiering and Caching

In this chapter, we will discuss the challenges of today's data storage and will attempt to explain the role of scalability, tiering and caching in mitigating costs and increasing performance. Moreover, we present the current solutions for boosting performance and we evaluate if they can be used in conjunction with Archipelago.

The structure of this chapter is the following. Sections 4.1, 4.2 and 4.3 explain what scalability, tiering and caching mean respectively. Section 4.4 attempts to exhibit the need for these techniques by providing a typical real-life scenario in which they can be used. Finally, Section 4.5 lists and evaluates some of the 3rd party, open-source solutions that employ the aforementioned techniques.

4.1 What is scalability?

Scalability, in storage service context, is the ability of the service to achieve two specific things:

1. accommodate the growth of load in a manner that does not impact the quality of the service and
2. utilize the addition of new resources to their full extend, in order to improve its performance.

There are two methods of scaling, horizontal (scaling out) and vertical (scaling up), which are explained below:

- *Horizontal scaling* applies to distributed services. It relies on the principle that adding more nodes to a system will mitigate the high load of the other nodes.
- *Vertical scaling* applies to all types of systems and refers to the addition of more resources such as better hardware, more RAM etc. to a node of the system.

The rule of thumb about these methods is that scaling up is the simpler solution for a service, albeit its performance cannot be increased much due to hardware limitations. On the other hand, scaling out is far more complex and requires a robust method of managing many nodes as well as their failures, but it may have lower costs (if nodes are made of commodity hardware), and has theoretically no limitations in performance gain (especially in share-nothing architectures).

4.2 What is tiering?

Tiering is the organization of different storage types in levels (or tiers) depending on their performance. These storage types usually differ in one of the following attributes: capacity, price or performance.

Medium	Access time (ns)
CPU registers and cache	< 10
RAM	$< 10^2$
SSD	$< 10^5$
Hard Disk	$< 10^7$

Table 4.1: Access times of storage mediums

Tiers such as SSD arrays or caches are necessary in most medium or larger deployments, in order to bridge the performance gap between RAM and magnetic disks, which can be seen in Table 4.1. To understand the need for tiering, consider the fact that when data do not reside in RAM and SSDs are not used, the performance penalty is x10,000 times the access time of RAM.

Tiered storage is analogous to the computer architecture model of memory hierarchy, which can be seen in Figure 4.1. Tiered storage is based on the same principles as memory hierarchy, in the sense that its objective is to keep ”hot” data, i.e. data that are requested frequently, in the higher tiers.

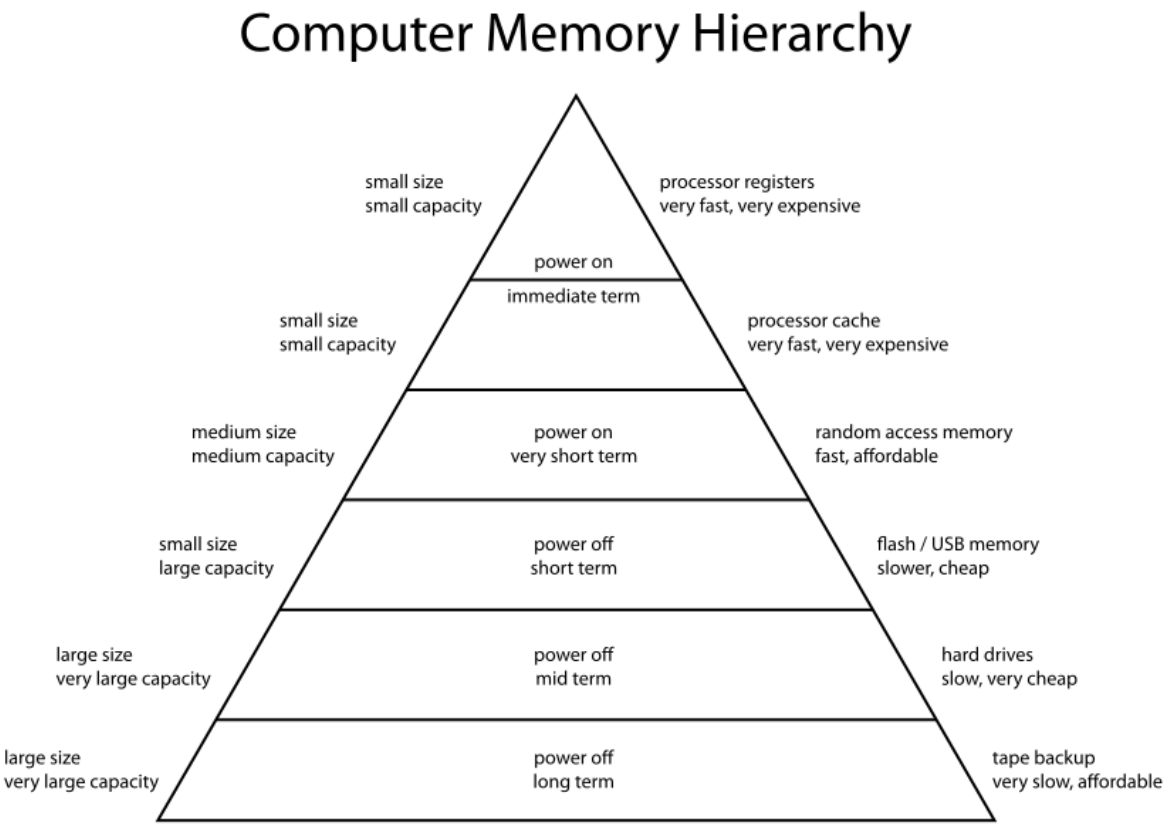

Figure 4.1: Computer Memory Hierarchy

4.3 What is caching?

In the context of I/O requests, caching is the addition of a fast medium in a data path, whose purpose is to transparently store the data that are intended for the slower medium of this data path. The benefits from caching is that later accesses to the same data will be faster than fetching them from the slower medium.

Caching is extensively used in computer architecture, as is evident from Figure 4.1. Besides the memory hierarchy model, it is a widely employed concept in storage services where fast mediums are used as journals for slower ones, essentially caching the data, or where dedicated servers are used to cache data, like in memcached's case (see more in Section 4.5.4).

4.3.1 Write policies

Cache write policies dictate the behavior of the cache when it receives a write request. There are two main write policies:

Write-through
: The write is acknowledged only when the data are written both on the cache and on the slower medium.

Write-back
: The write is acknowledged when data are written in cache. The slower medium is later updated with the correct data, when they need to be flushed or replaced by new ones.

Moreover, there are also policies that affect the behavior of a cache in write miss schenarios:

Write allocate
: Cache loads the corresponding data block from the slower medium and the data are written to it.

No-write allocate
: The write request bypasses the cache and writes directly to the slower medium.

Although the above policies can be combined as we wish, there are two main combinations that make more sense: i) write-back with write-allocate, so that writes are cached to benefit from subsequent reads and ii) write-through with no-write allocate, because the extra load of write allocate will not benefit us, since we write directly to the slower medium.

4.3.2 Caching limitations

In our introduction we have explained that fast mediums like RAM and SSD drives cost more dollars/GB than slower mediums, such as hard disks. For this reason, caches always have smaller capacity than the mediums they cache. So, when a cache reaches its maximum capacity, it must evict one of its entries. However, which entry is the one that must be evicted?

This is a very old and well documented problem that still troubles the research community. It was first faced when creating hardware caches (the L1, L2 CPU caches we are familiar with). In 1966, Lazlo Belady proved that the best strategy is to evict the entry that is going to be used more later on in the future[3]. However, the clairvoyance needed for this strategy is a little difficult to implement, so we resort to one of the following, well-known strategies:

- **Random:** Evict a randomly chosen entry. This strategy, although it seems simplistic at first, is sometimes chosen due to the ease and speed of its implementation. It is preferred in random workloads where freeing quickly space for an entry is more important than the entry that will be evicted.

- **FIFO (First-In-First-Out):** Evict the entry that was inserted first. This is also a very simplistic approach as well as easy and fast. Interestingly, although it would seem to produce better results than Random eviction, it is rarely used since it assumes that cache entries are used only once, which is not common in real-life situations.

- **LRU (Least-Recently-Used)** Evict the entry that has been less recently used. This is one of the most common eviction strategies, however, it is not simple to implement since the application needs a way to track and index fast the last references to all entries.

- **LFU (Least-Frequently-Used)** Evict the entry that has been less frequently used. There have been many derivatives of this algorithm that also use parts of the LRU algorithm which have promising results, but this algorithm itself is not commonly used. The reason is because it over-estimates the frequency of references to an item and it performs poorly in cases when an item is frequently accessed and then is not used at all.

The fact that write requests that have spawned the evictions cannot continue until the dirty data have been safely written to the storage backend, means that when the cache has no space left, its speed deteriorates to the speed of the slower medium.

The above observation indicates that the challenge for caching algorithms is how friendly their flushes are to the underlying storage. To elaborate on that a bit, hard disks excel on sequential payloads, so if a cache could flush in a more sequential way to the disk, it would boost its performance in these scenarios.

4.4 Real-life scenario

Usually, when a small deployment makes its first steps, it doesn't use SSDs due to management/hardware costs and since it is an investment that is actually needed when the deployment has proved that it will attract traffic. Instead, the most common setup is an array of RAID-protected commodity hard disks or fast SAS drives.

When the storage demands start to increase and more users use the service, the OS caching system of the storage nodes will soon prove ineffective and the randomness in the requested data will skyrocket the access times.

At this point, the administrators must take one (or more, if the budget allows it) of the following decisions:

1. Add more storage nodes in order to lower the load on the existing ones (horizontal scaling).

2. Buy battery-backed array controllers with volatile memory on-board, to improve access times (vertical scaling).

3. Put time-critical storage operations, such as journaling, in higher tiers (tiering)

4. Add RAM or SSD caches in write-back mode that will ACK the requests before they reach the slower mediums (caching).

The employment of one of the aforementioned techniques (scaling, tiering, caching) is of paramount importance for the future of the service.

4.5 Current Solutions

For the thesis purpose, we have evaluated a numerous of caching solutions. The results of our evaluations are presented below:

4.5.1 Bcache

Overview

Bcache has been designed by Kent Overstreet since 2011 and has been included in the Linux kernel (3.10) since the May of 2013.

Bcache allows one to use one or more fast mediums as a cache for slower ones. Typically, the slow medium is a RAID array of hard disks and the fast medium are SSD drives. Bcache has been specifically built for SSDs and has the following characteristics:

1. The data are written sequentially and in erase block size granularity, in order to avoid the costly read-erase-modify-write cycle.
2. It takes special care to mitigate wear-leveling by touching equally all SSD cells
3. It honors TRIM requests and uses them as hints for its garbage collection.

Installation and usage

Bcache is a kernel driver that needs a patched kernel and intrusive changes to the backing device.

On a nutshell, bcache edits the superblock of both the cache and backing devices in order to use them, rendering existing data unreadable. Then, it exposes to the user a virtual block device, which can be formatted to any file-system. This virtual block device is the entry point to the bcache code. Then, the caching device is attached to the backing device and at this point the virtual block device is ready to accept requests.

At any point, the bcache parameters can be further tuned via the sysfs interface.

Features and limitations

The most striking bcache feature is that it uses a custom built B+tree as an index, which has the added benefit that dirty data can be coalesced and flushed sequentially to the slower spinning medium. This provides a considerable performance speed-up for hard disks.

Some other noteworthy features of bcache are the following:

1. It can be used to cache more than one devices
2. It can operate in three modes, write-through, write-back and write-around, which can be switched on/off arbitrarily during normal usage or when the fast medium is congested.
3. It utilizes a journal log of outstanding writes so that the data are safe, even when an unclean shutdown occurs.
4. It can bypass sequential IO and send it directly to the backing device, since this workload is tailored for spinning disks.

4.5.2 Flashcache

Overview

Flashcache has been designed by Facebook and has been open-sourced in the April of 2010. It is a kernel module that is officially supported for kernels between 2.6.18 and 2.6.38 and is based on the Linux Device Mapper, which is used to map a block device onto another.

Installation and Usage

Flashcache's installation is not system-intrusive, in the sense that it needs only to compile the module against the kernel's source, modprobe it and then map the cache device upon the backing device, without making any changes to the latter.

Features and limitations

Flashcache uses a set-associative hash table for indexing. It has three modes of operation, write-through, write-back and write-around, and some basic performance tuning options such eviction strategies and dirty data threshold. Also, it has the following limitations:

1. It does not provide atomic write operations, which can lead to page-tearing.
2. It does not support the TRIM command.

4.5.3 EnhanceIO

Overview

EnhanceIO has been developed by STEC Corp. and has been open-sourced in the December of 2012. It is a fork of Flashcache which does not use the Linux Device Mapper and has some major re-writes in parts of the code such as the write-back caching policy.

Installation and Usage

The installation method is similar to the Flashcache's method. The source code is compiled again the kernel's source, which produces a module that can be modprobed. After that, the utilities provided can be used to map the cache device on the backing device.

Features and Limitations

Similarly to Flashcache, EnhanceIO uses a set-associative hash table for indexing. It also has improvements upon the original Flashcache implementation in the following areas:

1. The page-tearing problems have been solved.
2. Dirty data flushing using background threads.

4.5.4 Memcached

Overview

Memcached is a distributed memory caching system that is being widely employed by large sites such as Youtube, Facebook, Twitter, Wikipedia. It has been created in 2003 by Brad Fitzpatrick while working in LiveJournal and to date there have been numerous forks of the code, most notably including Twitter's twemcache and fatcache, Facebook's implementation etc.

When memcached came into existence, many social sites like LiveJournal were experiencing the following problem:

User pages would often have queries that would be executed hundreds of times per second or would span across the database due to a big SELECT, but whose nature would be less critical or would not change rapidly. Queries such as "Who are my friends and who of them are online?", "What are the latest news in my feed?" etc. which could be easily cached, crippled instead the database by adding a lot of load to it.

To tackle this problem, memcached can be used to utilize the unused RAM of the site's servers to cache these kinds of queries. Ten years later, memcached has become the defacto scale-out solution, and has use cases such as Facebook's, whose 800 dedicated memcached servers can serve billions of request per second for trillions of stored items[17].

Installation and usage

Memcached adheres to the client server model, with N clients connecting to M servers. Memcached, which is a user space daemon, runs on every server and listens for requests typically on port 11211. The installation is very easy since there are packages for most known distros. Once memcached has been installed, the administration needs to specify only the port and several performance options such as cache size and number of threads.

The clients on the other hand communicate with the memcached servers using native libraries. There are libraries that are written for most programming languages such as C, PHP, Python, Haskell etc. The clients can then specify which queries - or keys in general - want to be cached and the actual caching is done in runtime.

Features and limitations

Architecturally, memcached tries to do everything in O(1) time. Each memcached server consists of a hash table that indexes the keys and their data. Since the data size can vary from 1 byte to 1MB, memcached uses SLAB allocation in order to prevent memory fragmentation. In SLAB allocation, memory is reserved in fixed-sized pages, e.g. 1MB for memcached, which are divided in blocks of equal size. Then, items are stored to the SLAB whose block size is closer to their size.

Moreover, each memcached must be able to handle tens of thousands connections from clients, so it relies in libevent to do the asynchronous polling.

What's more interesting about memcached is that its main strength is actually its biggest limitation. Memcached has no persistence and in fact, data can be evicted in numerous ways:

1. Cached data have an expiration time after which they are garbage-collected.
2. Data can be evicted before their expiration time, if the cache has become full.

3. When memcached is out of SLAB pages, it must evict one in order to regain space. This leads to the eviction of more than one keys.

4. When adding or removing memcached servers, the Ketama algorithm that maps keys to servers will assign a portion of the existing keys to other servers. This change in mapping, however, will not actually move the existing keys to these servers and the data are essentially invalidated.

To sum up, the lack of persistence means that memcached will never hit the disk bottleneck due to flushes and will always be very fast, as long as the cache hit rate is high. On the other hand, its unreliable nature means that it is not a general purpose software and only specific workloads will be benefited from it.

4.5.5 Couchbase Server

Overview

Couchbase server, a NoSQL database which has been under active development by Couchbase Inc. since the January of 2012, is actually the product of the merge of two independent projects, CouchDB and Memebase, with CouchDB continuing as an Apache funded program. Couchbase aims to combine the scalability of memcached with the persistence of a database such as CouchDB.

Installation and usage

Couchbase provides two versions, a community edition, that lacks the latest bug fixes, and an enterprise edition. The community edition has an open-source license and can be installed easily in all major distributions from the official packages.

Once Couchbase Server has been installed, it can be configured through a dedicated web console or the command-line or the REST API. Its configuration has to do with the amount of RAM it will use and most importantly the cluster that it will join. Clusters are a deviation from the classic memcached architecture. They are logical groups of servers (or nodes) that are used for replication and failover reasons.

Like memcached, the communication with the servers is done through client libraries. These libraries are written for many different programming languages such as C, Python, Java, PHP, Ruby etc.

Features and Limitations

Couchbase Server adds the following important features to memcached feature list:

1. It can provide persistence for the data.

2. It uses data replication, which is one of the persistence guarantees.

3. It re-balances the data on resizes, so that they are evenly distributed across the database.

4.5.6 Honorary mentions

Repcached

Repcached is a memcached 1.2 fork that aims to provide asynchronous data replication. It didn't catch up however for the following reasons:

1. The added data replica merely slims the margins of losing the data but not erases them.
2. It is based in memcached 1.2, which has been released four years ago. Since then, there have been numerous performance improvements.
3. The synchronization cost of replication was high.

Ramcloud

RAMCloud[19] is a project that is being directed by John Ousterhout at Stanford University. RAMCloud is not a caching system, but a distributed storage system that uses primarily DRAM, whereas hard disks are used only for crash recovery scenarios[18].

RAMCloud aims to be persistent, meaning that writes to a server are acknowledged only when the data are replicated to at least three other servers and are in the process to be written asynchronously to disk. Moreover, since DRAM has faster access times than network, RAMCloud requires fast Infiniband connections between nodes for lower access times as well as for quick data migration in recovery scenarios. Finally, in order to retain data even after a power loss, RAMCloud needs either battery-backed servers or DIMM modules with integrated super-capacitors.

Although RAMCloud is an aspiring project, we have chosen not to investigate it further for two reasons:

1. it requires non-commodity hardware, which is the opposite of our goal.
2. it is not production-ready yet.

Page-cache

Theoretically, our search for a production-ready, fast cache that scales and can be used in conjunction with Archipelago can stop, if we allow the VM's hypervisor to use Linux's page cache. In fact, we have measured its performance (see Figures 7.10 and 7.11) and we can expect a 10x performance increase for writes and a 6x performance increase for reads.

However, there are several reasons why we have not favored this approach. The main reason is that this type of caching, as we will explain in the following section, is orthogonal to the way Archipelago handles data. It is agnostic to the CoW policy of Archipelago and as a result, it caches duplicated blocks wasting valuable memory.

Moreover, we have little or no control over the Linux page cache. For example, we cannot enforce a limit to the cache size or apply a replication policy.

4.5.7 Evaluation

The above solutions fall into two broad categories; block store and key-value store. Both of these categories can be used in Archipelago, since there are peers that use block store semantics, e.g. when xsegbd receives a request, and peers that use key-value store semantics, e.g. when vlmc has translated a block request to object request.

We will start our evaluation with the block store techniques first. These methods have in common that they are kernel modules which cache requests that are targeted to a single (or more) slow block device. So, we can use them in this way:

1. Add SSDs to the host machine where the VMs are running.
2. Partition the SSDs so that there is one partition for each volume that is running in this host.
3. Install the kernel module and when a VM is created, run the necessary commands to map a partition of the SSD to the virtual block device of the VM.
4. Use the block device that the kernel module exposes and pass it to the hypervisor.

The main issue with this approach (and host caching in general) is this: If the cloud software is distributed then, when a host crashes, the VMs can be restarted in another host. This is possible in deployments where the instances' attributes are known by the respective cloud management software and their data are stored in a distributed storage system.

The issue arises when caching in write-back mode, where the VM's most recent data will be down with the host. In this case, the VM will not be able to start in another host or worse, it will be in inconsistent state with whatever implications this may have.

Even if we ignored the above, there are other issues too, such as:

1. If a user process segfaults, it can be restarted promptly, without interrupting the rest of the VMs. If however the kernel segfaults, the host will go down.
2. Caching at xsegbd level does not take advantage of the fact that large parts of a VM's volume are shared between other volumes due to Copy-on-Write. This means there will be lost space in the SSD for data that are actually duplicate.
3. Flashcache has page tearing issues, which we want to avoid.
4. Bcache is for newer kernels and to date it still has some bugs.
5. Having a fixed partition for each volume does not scale, since for each VM with high activity, there can be 10 other stale VMs that practically eat up cache space.

We will continue with the second category, the key-value store solutions. The programs that fall in this category have two important advantages:

1. They are distributed by nature and try to eliminate any SPOF [1], in the same way that RADOS does.
2. They can utilize the extra RAM of a node, which is plenty in the RADOS nodes.

[1] Single Point of Failure

However, there are also some fundamental problems with them:

1. Memcached has no concept of persistence. Not only that, it basically relies on the fact that has no persistence that hacking our way through that issue would create a different software.
2. Couchbase Server has no way to use RADOS as its backing device and has its own concept of replication.

For the above reasons, we have decided to roll out our own implementation, which is presented in the following chapter.

Chapter 5

Design of cached

In Chapter 4, we have addressed the importance of scalability, tiering and caching for the performance of a storage service. We have also presented various caching solutions and explained why we rejected them.

This situation ultimately lead to the design of a new caching entity for Archipelago. This entity is called ”cached”, which simply means **cache d**aemon. We have decided to invest time in creating our own implementation mainly for two reasons: i) to create a peer that understands the Archipelago logic and can integrate naturally with it and ii) to measure the best possible performance gain that we can achieve and evaluate if it is worth to further pursue and improve upon it.

Thus, in addition to the observations of the previous chapter, we will provide some stricter requirements that our solution must have. These requirements are:

1. **Nativity:** Cached must be native to Archipelago i.e. not need any translation layers to communicate with it.

2. **Pluggability:** Cached must be able to provide a caching layer between peers that are already in operating mode without restarting Archipelago. Also, it must be removed without disturbing the service.

3. **In-memory:** Our main goal is to measure the maximum performance gain that we can achieve, which means that caching in DRAM is our best option.

   This also means that we will temporarily hand-wave the issue of data volatility and we will concern ourselves with it if cached has promising results.

4. **O(1) complexity** For performance reasons and since we already cache data in memory, the complexity of the indexing mechanism should be as small as possible.

The following two chapters are the main bulk of this thesis as they present the design and implementation of cached that aims to fill the above requirements.

More specifically, this chapter provides an in-depth description of the design of cached. Section 5.1 provides the design rationale of cached and explains how its design meets the above requirements. Section 5.2 presents the building blocks of cached while Sections 5.2.2, 5.2.4 and 5.2.5 provide a detailed explanation of their design. Moreover, Section 5.2.3 illustrates the request flow of one of the most important components of cached, the xcache. Section 5.3 explains how cached utilizes the aforementioned components and further introduces some unique ones that have been tailored specifically for cached. Finally, in Section 5.4 we illustrate the request flow for cached.

5.1 Design rationale

One of the first architectural decisions was to implement cached as an Archipelago user-space peer. This choice was the most natural one since it provides the smallest possible communication overhead with the other Archipelago peers. Also, this design decision covers the **nativity** requirement we posed at the beginning of this chapter.

The above design choice has another advantage too; we can plug on-line the cached peer between the vlmcd and blocker and unplug it when we want to. This opens up numerous possibilities such as plugging cached for QoS[1] reasons, when there is a peak in I/O requests. This also means that the **pluggability** requirement is also being met.

The next important design decision was what will cached index. Given that it will reside between the vlmcd and blocker, where the VM's requests have already been translated to object requests, the natural choice is to cache objects. To understand why our caching peer must be close to the Archipelago's logic, i.e. the translation of blocks to objects, consider the following two points:

- Like bcache, cached must not only cache object requests fast but also try to coalesce them so that, when needed, they will be flushed to the slower medium in a more sequential fashion. The fact, however, that a VM's volume is partitioned into different objects, means that sequential data (in volume context) which reside in different objects will probably not be sequential in the storage backend too.

  Thus, in Archipelago it makes sense only to coalesce data in the object range (commonly 4MBs). If our implementation was caching in block level (e.g. bcache), it would be unaware of that fact.

- We can use cached to further improve the performance of Copy-on-Write. If cached receives a CoW request and has the object cached, we can acknowledge the copy on the fly, without waiting for the slower medium. Note that this is a feature that has not been implemented yet, but will be in the future.

Having decided that cached will cache objects, the next step is to decide i) on the index mechanism and ii) on **what** exactly will we index.

As for what we will index, it would be an overkill to further partition the objects and index the regions within them. Moreover, this would make sense only if the objects were large (e.g. like volumes). So, we index object names solely.

As for the index mechanism, we have chosen to use a very fast in-memory hash table that will keep the data in a preallocated space in RAM. This covers the **in-memory** requirement that we have set above. Also, the choice of the hash table is one of the reasons that our implementation has **O(1) complexity**.

Finally, another important decision was whether cached would be a multi-threaded peer. We have decided that we will implement it this way and then evaluate the performance of the implementation to find out if we are benefited by multi-threading or not.

Thus, cached must be able to work with multiple threads which will accept requests from cached's request queue and serve them concurrently with the other threads. Of course, multi-threading can be very tricky, especially when we are dealing with I/O requests and simultaneous accesses to the same object. So, in order to achieve a balance between safety and speed, we use a fine-grained locking scheme in critical sections, which is discussed in detail in Section 5.2.4.

[1] Quality of Service

5.2 Cached components

At this point, we must do an intermission before we show the design of cached. Specifically, we will show first the design of the cached's components, since many cached operations rely on them and the reader needs prior knowledge of them to grasp the cached design.

5.2.1 Overview

In this section, we will list the main components that cached relies on. Per Archipelago policy, most of these components have been written in the xtypes fashion. Xtypes can be considered as modules for Archipelago that are designed to be reusable and to mostly reside in the shared memory segment.

The components of cached can be seen below:

- xcache, an xtype that provides indexing support, amongst many other functionalities.
- xworkq, an xtype that guarantees atomicity for execution of jobs on the same object.
- xwaitq, an xtype that allows conditional execution of jobs.

and their design will be discussed in-depth in the following sections.

Also, we must note that the above components predate our cached implementation and are not a contribution of this thesis[2]. They are presented however in this thesis for clarity reasons.

5.2.2 The xcache xtype

xcache is the most important component of cached. It is responsible for several key aspects of caching such as:

- entry indexing,
- entry eviction,
- concurrency control and
- event hooks

In Figure 5.1, we can see the design of xcache:

Xcache consists of two hash tables. In the first hash table (a), xcache keeps an index of the currently active entries, whereas on the second hash table (b) it keeps an index of the entries that have been recently evicted but not removed, meaning entries that are in the process of removal but still have pending jobs.

The cache entries are located in a contiguous space (c) where each hash table points to. In order to acquire or release an entry, all we need is to pop or push an entry index from a special purpose stack (d). A more closer inspection of their design is presented in Figure 5.2. In this figure, each entry is marked with a different color for clarity reasons.

[2] xcache is an exception since we have extended its functionalities for our purposes

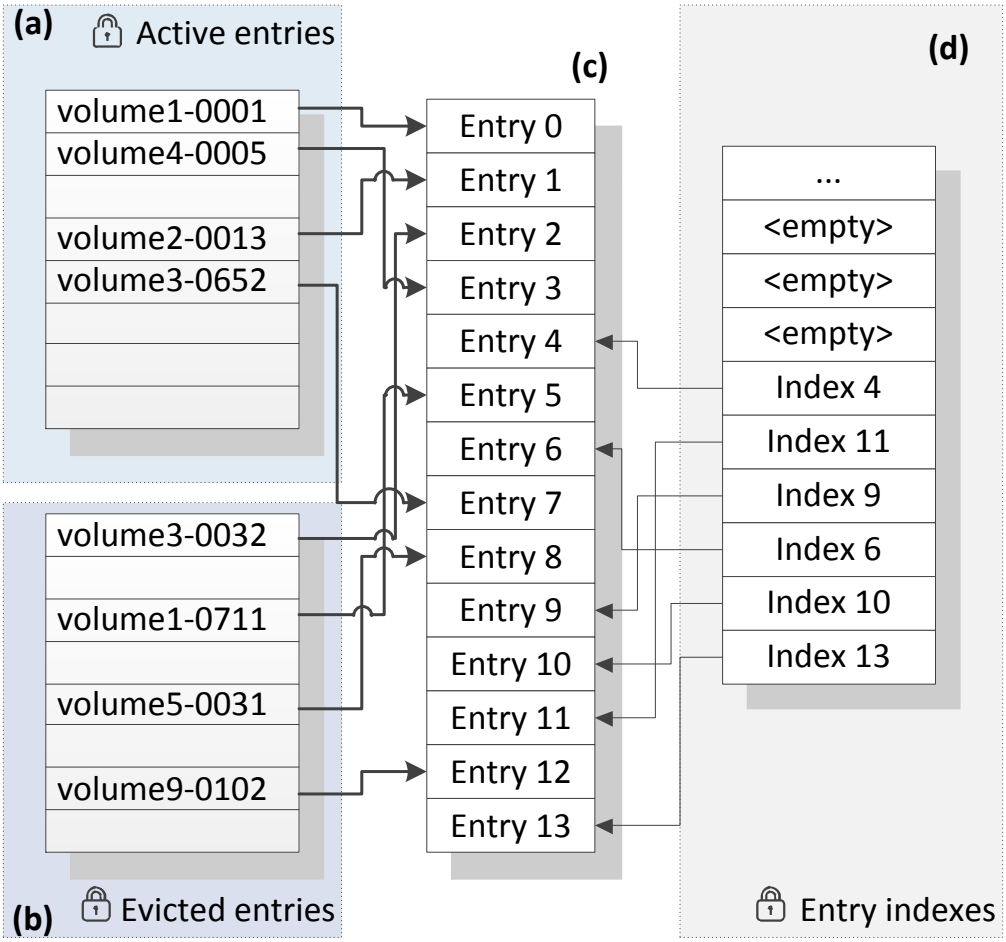

Figure 5.1: Xcache design

Every entry has the following important fields: i. Pointers to the previous (O) and next (Y) entries in the LRU list (see more in Section 5.2.2), ii. a reference counter and iii. the entry's name

Moreover, each entry has its own lock as well as a pointer to the entry's data, as set by the peer.

On the following subsections, we present the features of xcache as well as their design.

Entry Preallocation

Since xcache indexes a bounded number of entries, there is no need to allocate them on-the fly using malloc/free. Considering that we are caching at RAM level and not at SSD level, the system call overhead will have a considerable impact on performance. Thus, in our case, we preallocate the necessary space in advance.

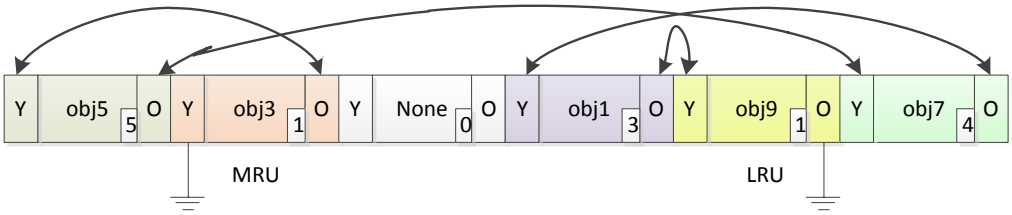

Figure 5.2: Xcache entry design

Also, note that we will use the term ”cache node” from now on when we refer to a chunk of the preallocated space and ”cache entry” when we refer to a cache node that has been associated with an object.

Entry indexing

The index mechanism that xcache uses is a hash table named xhash, which is also an xtype. The reasons why we have chosen to use a hash table are:

1. Given that volume names are created pseudorandomly, we expect that the hash values of the object names will have approximately uniform distribution and by association, practically zero hash collisions, as in most real-life scenarios. If an adversary, on the other hand, was allowed to choose its own names, that would be a different case.

   Thus, the insert, lookup and delete operations should occur in constant time.

2. Hash tables can preallocate the space needed whereas tries/b-trees/bst will allocate nodes as new entries are inserted. There is actually an exception to this which will be discussed below.

3. We don’t need to do substring matches (which is an advantage of tries)

4. We don’t need to traverse the entries sequentially (which is an advantage of B-trees and BSTs)

The hash table that is used is heavily based on dictobject[6], the Python dictionary implementation in C. What is interesting about dictobject, and is a feature of other hash tables too, is that it dynamically resizes once it has reached the two thirds (2/3) of its capacity. This is the exception that we have mentioned in (2). We will attempt to explain what is resizing and how it affects us.

Resizing occurs to prevent performance degradation from the hash collisions that will likely ensue once the hash table reaches its maximum capacity. On resizing, the entries are typically rehashed and the hash table is reallocated to double its capacity.

Resizing may seem as an expensive operation, but it is actually not, if we consider the following:

> [10]Suppose that we insert n entries in a hash table that was created to hold only a fraction of n, one (1) entry in the worst case. In this case, the number of rehashes will be $n + n/2 + n/4 + n/8 + ... + 1 = 2n$. This means that the **amortized** cost of rehashes will be $2n/n = 2$ rehashes for each item, which is practically O(1) and thus negligible. The same amortized cost applies to the number of entry copies on reallocation.

Besides the hash table, which answers to the question ”Where is the entry?” we also need another mechanism to answer the question ”Is the entry still referenced?”. xcache has such a mechanism which is commonly called ”reference counting”. Specifically, each entry has a counter that is incremented/decremented when a user accesses/releases an entry.

To sum up, when an entry is inserted, we use its name as a key and we update its refcount to 2, one reference from the user and one standard reference from the hash table. When we lookup for an entry, we use the entry's name as a key and then increment by 1 its refcount.

Entry eviction

The decision to have xcache index a bounded number of entries means that when it reaches its maximum capacity and is requested to index a new entry, it has to resort to the eviction of a previously cached entry. Evicted entries are not removed immediately from xcache. They are instead set in an ”evicted” state and they reside in a special-purpose hash table until the user confirms that they can be removed.

xcache handles evictions in an interesting way. More specifically, evictions occur implicitly and not explicitly, meaning that the user (peer) does not have to evict entries manually. For example, when a user tries to insert a new entry to an already full cache, the insertion will succeed and the user will not be prompted to evict an entry manually. Moreover, the user will be notified via specific event hook that is triggered upon eviction.

The scheme of implicit evictions and later on notification of the user, has the advantage that lookups, inserts and evictions can occur atomically by xcache. This would not be the case if the user was responsible for the evictions.

As for the eviction strategy, we have utilized an LRU queue. We have mitigated the cost of keeping the last reference for each entry by creating a simple LRU algorithm, which has O(1) complexity for all update actions.

The LRU queue that we have created is a very simple structure. It is a doubly linked list whose pointers reside in the xcache entry and its head (MRU) and tail (LRU) entries are stored in xcache.

There are four operations that are supported by this LRU:

1. *Insertions*, where we simply append the new entry to the head of the list.
2. *Evictions*, where we remove the tail of the list.
3. *Updates*, in which case we already know the entry we want to update and have instant access to its pointers. We can thus unlink this entry from its neighbors and append it to the head of the list.
4. *Removals*, which are similar to *updates*. In this case we also know the entry that is going to be removed, meaning that we can unlink it and remove it safely.

An illustration of this list can be seen in Figure 5.2. In this example, the references order is the following:

The least recently used entry is **obj9**, next is **obj1**, then **ob7**, then **obj5** and finally the entry that has been mostly recently used is **obj3**.

More about the implementation of the LRU algorithm can be found in Section 6.1.5.

Concurrency control

The concept of concurrency control has been discussed in Section 2.3. The goal of xcache is to handle safely - and preferably fast - simultaneous accesses to the shared memory.

In order to do so, we must first identify which are the critical sections of xcache, to wit, the sections where a thread modifies a shared structure. These sections are the following:

- **Most xhash operations:** Inserts and removals can modify the hash table (e.g. they can resize it, add more entries or delete existing ones). This also means that lookups must not run simultaneously with the above two operations.
- **Cache node claiming:** Before an entry is inserted, it must acquire one of the pre-allocated nodes from the cache-node pool and we must ensure that this can happen concurrently from all threads.
- **Entry migration:** An entry can migrate from one hash table to the other e.g. on cache eviction. This migration involves a series of xhash operations; removal from one hash table and subsequent insertion to the other. These two operations must occur atomically.
- **Reference counting:** Every entry must have a reference counter. Reference counters provide a simple way to determine when an entry can be safely removed. Since many threads can have access to the same entry, we must provide a way to update the reference counters atomically.
- **LRU updates:** Most actions that involve cache entries must subsequently update the LRU queue. These updates must also occur atomically.

Let's see what guarantees we provide for each of the above scenarios:

- **xhash operations:** We provide a lock for each hash table. Only one thread can access each hash table at any time.
- **Cache node claiming:** The cache-node queue is also protected by a lock.
- **Entry migration:** When an entry is migrated from one hash table to the other, we always acquire the lock of the hash table of active entries first and then the lock of the hash table of the evicted entries. The order on which we take the locks is very strict to avoid deadlocks.
- **Reference counting:** For the atomic increases and decreases of a counter, we don't need a lock and its added overhead. Instead, we can use the atomic get and atomic put operations that the CPU provides.
- **LRU updates:** Since the majority of LRU updates take place when a new entry is inserted in the hash table, we can protect our LRU under the same cache lock.

Event hooks

Since xcache is created to provide core caching functionalities for other peers, it must also notify them when it takes an implicit action that the peer is not aware of. In Section 5.2.2 we have seen one implicit action that xcache takes when a user inserts an entry, namely eviction.

Besides this event, there are others. The complete list is the following:

cache node initialization: This hook is triggered when a cache node is initialized. It is triggered once only for each node, during the initialization phase of xcache.

cache entry initialization: This hook is triggered when a cache entry has been inserted in the cache.

cache entry eviction: This hook is triggered when a cache entry has been evicted from the cache.

cache entry reinsertion: This hook is triggered when an evicted entry has been reinserted in the cache.

cache entry finalization: This hook is triggered when an evicted entry's refcount has dropped to 0. This serves as a warning for the user who has the opportunity to let the cache entry go or increment its refcount.

cache entry put: This hook is triggered when an evicted entry has been totally removed from the cache.

cache entry free: This hook is triggered when a removed entry's cache node has been sent back to the cache node pool.

For each of the above events, we have created the respective event hook. The peer that uses xcache may choose, if it wants, to use them and if so, it can plug its own event function for each hook which will be called when the event is triggered.

5.2.3 The xcache flow

To make the way xcache works a bit more clearer, we will see the flow for four of the main xcache operations; insertion of a new entry, lookup of an entry, eviction and removal of an entry:

Insertion

The insertion process is described below:

1. In order to insert a new entry, we must first allocate the space for it. This means that we need to pop an entry index and use it to initialize the entry it points to. If there are no indexes left, we inform the peer.
2. Next, we search the active entries to ensure that no other thread has already inserted our entry. If however our entry is there, we return and free the entry we have previously allocated.
3. Then, we search the evicted entries to likewise ensure that our entry has not been just evicted. In this case, we reinsert it and free the entry we have previously allocated.
4. If our entry is not in any of these two hash tables, we try to insert it to the hash table of active entries.
5. If the hash table is full, we resort to the eviction of another entry and inform the peer.
6. In either case, we update the reference count of our entry by 2, one for the hash table and one for the request.

Lookup

The lookup process is a lot simpler. To lookup an entry, we only need its name. If this name exists in the hash table of active entries, we update the reference count of the entry by 1 and we return with the entry index.

Eviction

When a cache entry is evicted, we do the following:

1. Having acquired the lock of both hash tables, we remove the entry from the active entries and put it to the evicted entries.
2. We release these locks and inform the peer using the eviction hook.
3. Once the peer has been informed, we decrease the refcount of the entry by 1, since it is no longer referenced by the hash table of active entries.

Removal

The removal of an entry occurs only when its refcount has reached 0. More specifically:

1. We inform the peer that the entry is about to be removed with the finalization hook.
2. We then proceed to check again the refcount of the entry.
3. If it is more than zero, this means that the entry has been reinserted or that the peer has taken an action in the finalization hook. In this case, we can leave.
4. Else, we can safely remove that entry from the evicted entries and inform the peer.

5.2.4 The xworkq xtype

The xworkq xtype is a useful abstraction for concurrency control. Its purpose is to enqueue ”jobs” (protected by a lock) and ensure that only one thread will execute them. There is no distinction as to which thread this will be, as well as no execution condition. The executive thread is simply the one that acquires the lock first.

xworkq is generally used when multiple threads want simultaneous access to a critical section. Instead of spinning indefinitely, waiting for a thread to finish, they can enqueue their job in the xworkq and resume processing other requests. xworkq is also generic by nature, since the ”job” is simply a target function and its input data.

On Figure 5.3, we can see the design of xworkq:

It consists of a queue where jobs are enqueued. The thread that enqueues a job can attempt to execute it too, by acquiring a lock for the xworkq. If the lock is unheld, the thread will acquire it and will be able to execute the enqueued job. Else, it can safely leave and its job will be executed by the thread that holds the lock.

In cached context, every object has an xworkq. Whenever a new request is accepted/received for an object, it is enqueued in the xworkq and we are thus ensured that only one thread at a time can have access to the object’s data and metadata.

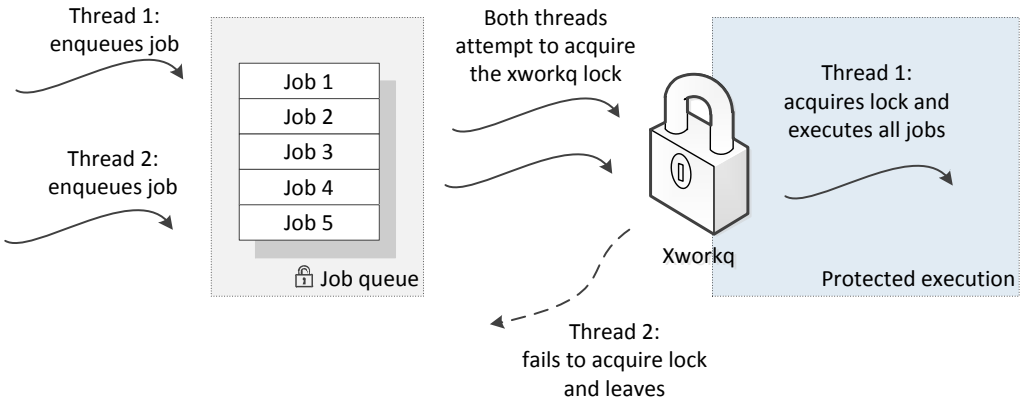

Figure 5.3: Xworkq design

5.2.5 The xwaitq xtype

The xwaitq xtype bears some similarities to the xworkq xtype. Like xworkq, it is also an abstraction where ”jobs” are enqueued and dequeued later on to be executed. Unlike xworkq though, jobs are executed only when a predefined condition is met. Another distinction is that the jobs in xwaitq are assumed to be thread-safe and can be executed concurrently by any thread.

xwaitq is commonly used in non-critical code sections that can be executed only under specific, predefined circumstances. The ”jobs” that are enqueued in xwaitq are the same as the jobs of xworkq.

The design of xwaitq is illustrated in Figure 5.4.

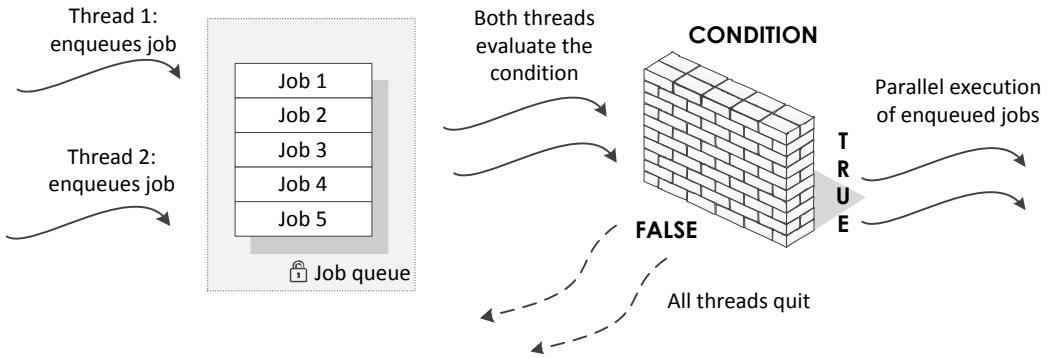

Figure 5.4: Xwaitq design

Unlike xworkq, before a job enters the waitq, the thread can attempt to execute it by checking the execution condition. Only if the condition is **not** met does the thread enqueue the job to the queue. Before the thread leaves, it ”signals” the queue and essentially rechecks the condition to ensure that it can’t be executed. It can then safely leave since its job will be executed when another thread signals the queue successfully.

In cached context, xwaitqs are used to enqueue jobs which cannot be executed immediately. Common cases are when we have run out of space, when we have run out of requests etc.

5.3 Cached Design

At this point, we have discussed in depth the design of the cached components. Having the above sections in mind, we can present how cached has been designed. We will illustrate the design of cached from two different perspectives: the operational perspective, which can be seen in Figure 5.5, and the component perspective, which is properly discussed in Section 5.4 and can be seen in Figure 5.6.

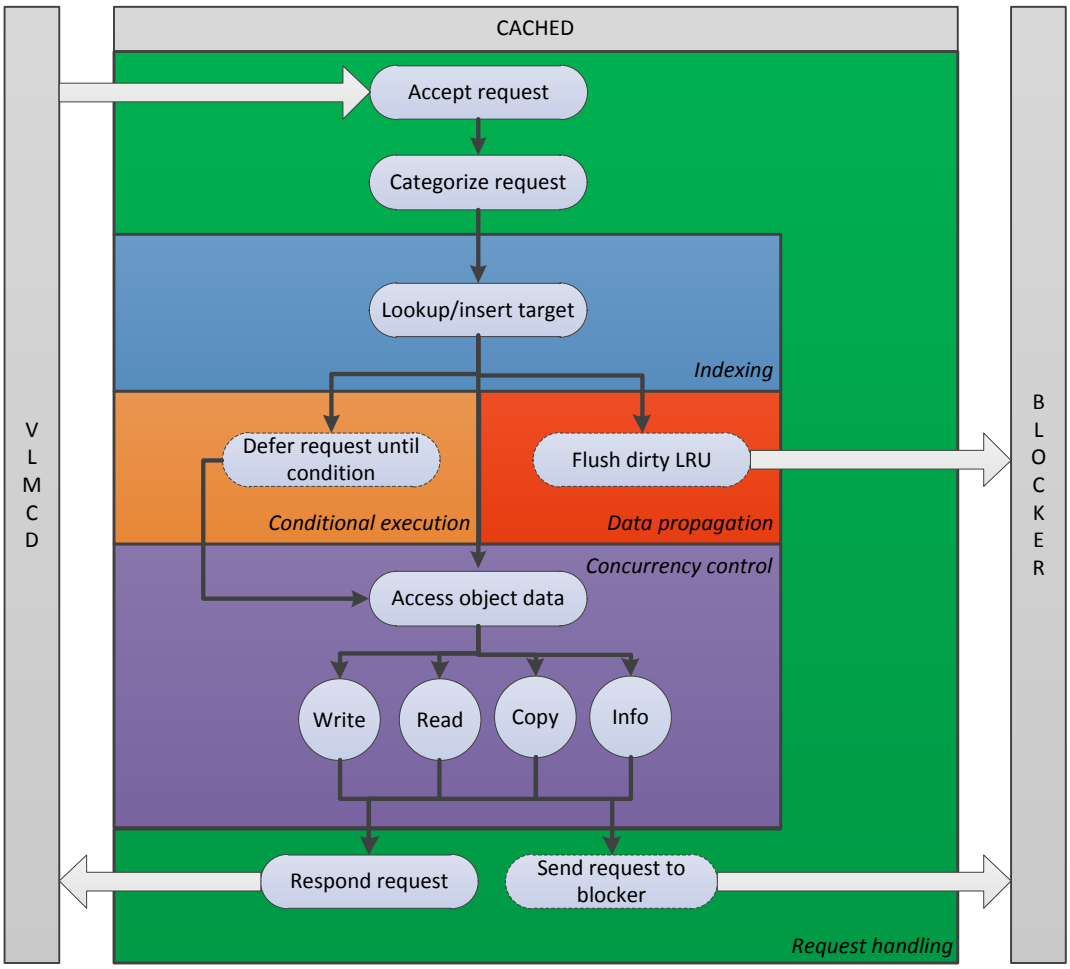

Figure 5.5: Cached design - operations

Cached has been been designed mainly as the orchestrator, a peer that utilizes several different components to handle various tasks such as indexing (xcache), concurrency (xworkq) and deferred/conditional execution (xwaitq). Cached however is not limited to the above role as these components do not cover all of the needed tasks. There are several other key tasks that cached must undertake, namely:

- *Request handling*, which is how cached handles requests from other peers and sends its own.

- *Write policy enforcing*, which is the enforcing of a cache write policy (write-through, write-back).
- *Data propagation*, which controls how data changes are propagated to the slower medium.

Moreover, cached extends its repertoire using some unique components that have been specifically crafted for it:

1. Bucket pool, which is a preallocated space from where cached can draw resources for the object's data.
2. Utilities for asynchronous task execution.
3. Utilities for various book-keeping purposes.

We will further explain how cached manages the above new components and tasks in the following sections.

5.3.1 Request handling

Cached operates as a peer that receives requests from the vlmcd. The majority of these requests will be read/write requests, but there are other types of requests too such as copy requests (sent when an object is copied-on-write) and info requests i.e. queries on what is the size of an object. Each of these requests must be handled independently, using special-purpose functions.

Furthermore, cached will also issue requests to the blocker mainly on two occasions: when it flushes a dirty object and when operating in write-through mode.

This means that, cached must be able to categorize requests and send them to the appropriate functions. Moreover, it must be able to create requests of its own, as well as handle cases such as running out of requests.

5.3.2 Write policy enforcing

The user defines beforehand the write policy of cached. There are two options: write-through and write-back. These policies aren't new and have already been discussed in Section 4.3.1, but let's see what these policies translate to in cached context.

- In **write-back** mode, cached caches writes, immediately serves the request back and marks the data as dirty. When a read arrives, it either serves the request with the dirty data (read-hit) or forwards the request to the storage peer and caches the answer (read-miss).

  This policy is used when we want to improve read and write speed and can sacrifice data safety.
- In **write-through** mode, cached forwards writes to blocker, servers the request when blocker replies, caches the data and marks them as valid. When a read arrives, it either serves the request with the valid data (read-hit) or forwards the request to the storage peer and caches the answer (read-miss).

  This policy is used when we want to improve read speed and want to make sure that no data will be lost.

These policies are specified once during cached's deployment and cannot be switched on/off later on.

5.3.3 Data propagation

In order to ensure that we have no data corruption, cached must provide the following two guarantees: i) consistency of cached data and ii) correct propagation of data updates to the storage backend.

As for the first guarantee, we have explained in section 5.2.4 that the consistency of the cached data is greatly secured using the xworkq as the guard of parallel accesses to the object's data.

As for the second guarantee, one might consider that flushing the dirty data of an object would be sufficient to correctly propagate data changes to the blocker. However, consider the scenario when a dirty object is evicted and then subsequently reinserted, overwritten and evicted again. The result would be two flush requests for the same data, sent to the blocker. Although the blocker guarantees that there will be no page-tearing, we cannot be sure about the order on which the data will be written.

To solve this problem, flush jobs must be deferred (using an xwaitq) if another flush for the same object is in flight.

5.3.4 Bucket pool

We have already explained in Section 5.1 the reason why cached caches objects. There is, however, one important design issue that we must address. This issue is how will cached perceive the object's data. To make it a bit clearer, given that an object typically has 4MB of size, what should cached do when a user requests e.g. a 16KB chunk of it?

If we perceived the object's data as a monolithic chunk, we would need to read and cache the whole object just to reply to the user's request. If the user then requests a chunk from another object, we would have to cache that object too and in the end, we would thrash our cache [3].

The solution we propose is to further divide objects to the next and final logical entity, buckets (typically 4KB of size). Each bucket consists of its data and metadata and cannot be half-empty, or half-allocated. This way, we can also know which parts of the cached object are actually written, or are in the process of being read etc.

Thus, our solution to the hypothetical problem we have posed above is to request 16KB from the blocker, store the result in 4 buckets and then respond the request to the user.

The above answer, however, is not entirely complete. It implies that buckets are something readily available or attached to the object. Although each object *could* have its buckets pre-allocated, this would limit the objects that we can cache since that, even if the user requested only a small chunk, each object would statically need 4MB of space.

Ideally, we would like to be able to cache thousands of objects but i) allocate a much smaller amount of buckets and ii) strictly when the user requests to. To make things even faster, we would also like the buckets to be preallocated (like cache nodes in Section 5.2.2) to avoid the overhead of malloc/free system calls.

We have achieved the above by creating a bucket pool. The design of the bucket pool is the same as the design of cache entries, as shown in Figure 5.1. More specifically, the bucket pool size is static and has been set during initialization by the administrator. After the necessary space has been allocated, it is divided in buckets and all the bucket indexes are pushed in a lock-protected stack. Then, each thread can pop bucket indexes from the pool and attach them to an object when needed. When that object is evicted, its attached buckets indexes are pushed back to the pool.

[3] cache thrashing occurs when we aggressively cache data that is only used once and effectively leads to a snowball of evictions

5.3.5 Asynchronous task execution

There are cases when a request cannot be processed immediately e.g. when it needs to allocate the necessary buckets for the object's data. In these cases, we use a utility called `cache-io`, which i) holds the original request and ii) describes the asynchronous task that will be executed. . In order to execute a task, we simply need a function pointer to the entry point of this task and the necessary arguments.

Using Cache-IOs, the request can break in more than two flows of execution, since the asynchronous task can allocate new requests for its own purposes. In order to know when the asynchronous task has finished, we keep track of the number of pending requests that it has spawned.

5.3.6 Book-keeping utilities

In order to know information such as when an object is in flushing state, when a bucket has been allocated or when a bucket is being read etc., cached must employ some sort of book-keeping of their states. The entities that require to track their states are:

1. **Buckets:** Each bucket has two different states that must be tracked. The first is its allocation state:

   i) free, meaning that the bucket is not allocated
   ii) claimed, meaning that the bucket is allocated

   The second is its data state:

   i) invalid, meaning that it currently holds no data
   ii) valid, meaning that it has data that are in sync with the storage backend.
   iii) loading, meaning that a read request that includes this bucket has been sent to the blocker
   iv) writing, meaning that the bucket contents are being written/flushed to the blocker.
   v) dirty, meaning that its data are newer than the data of the storage backend.

2. **Objects:** The bucket states of an object provide a good indication of the object's status, yet not a complete one. The statuses we keep for the objects are:

   i) ready, meaning that it is ready to accept data
   ii) invalidated, meaning that it is not ready to accept data
   iii) flushing, meaning that it is currently flushing dirty data to the blocker
   iv) failed, meaning that a request has failed for this object and we must stop using it

3. **Cache-IOs:** Cache-IOs also have states. They are the following:

   i) reading, meaning that the task wants to read data
   ii) writing, meaning that the task wants to write data
   iii) served, meaning that the task has been served the data it needed
   iv) failed, meaning that the task has failed

Moreover, we keep global and per-object counters of every object's bucket states. The benefits from this approach is that we can know at any time if an object (or cached in general) has a bucket in a certain state.

5.4 Cached Flow

In this section, we will discuss how cached handles read/write requests. Through the explanation of the cached flow, we hope to reassemble in the reader's mind the puzzle pieces that we have laid out in the previous sections of this chapter.

In Figure 5.5, we have essentially assorted the flow of cached in a set of logical operations. Now that we have explained all cached components, we can link these operations to the components that we have designed. The connection between cached operations and components is illustrated in Figure 5.6.

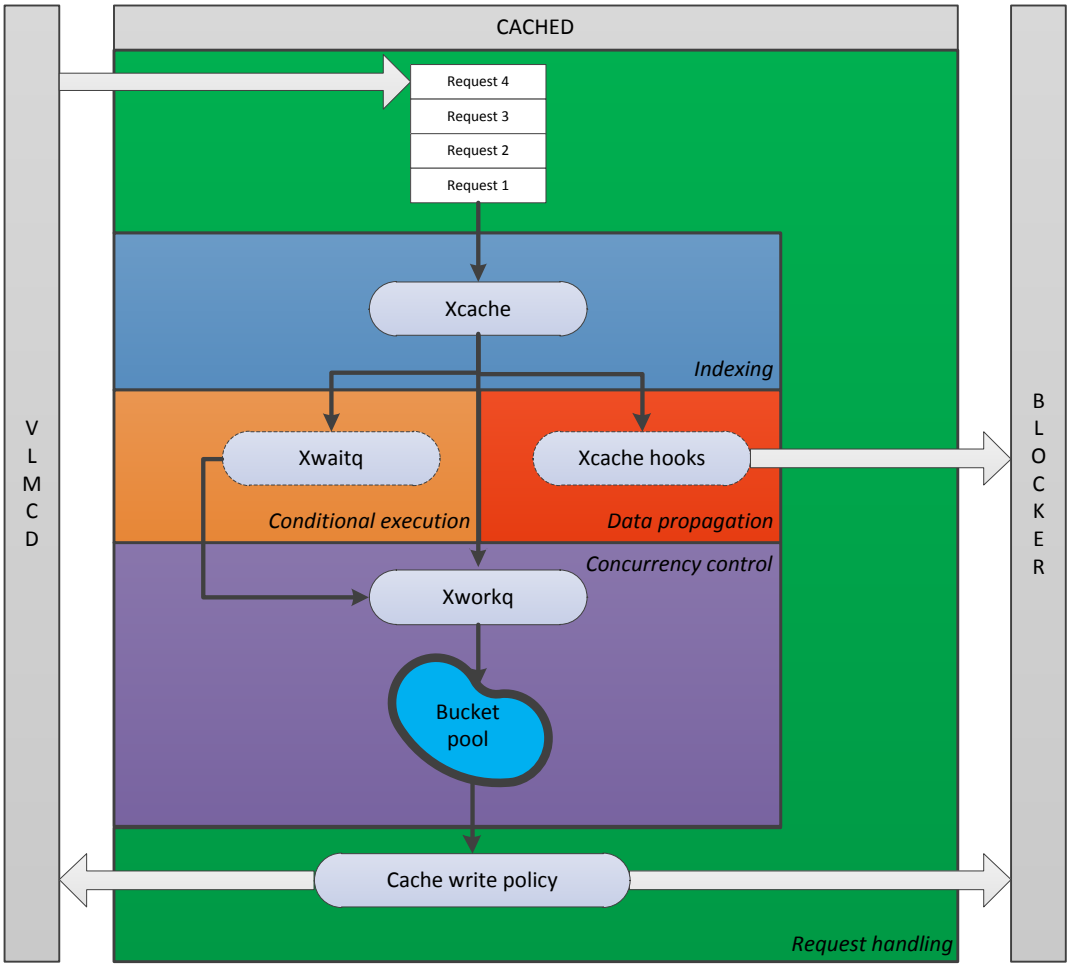

Figure 5.6: Cached design - components

We will explain each step of the cached flow, as is presented in Figures 5.5 and 5.6. Note that we will focus on the read/write requests and not the copy/info requests.

5.4.1 Main steps

Request handling

Initially, cached receives an XSEG request from vlmcd. Cached then checks the type of the request and it forwards it to the right channels.

Indexing

Next, cached uses the name of the request's target to check if it has been indexed (lookup).

- If the object is in the cache, it stores the returned handler for later references to the object data.
- If not, it allocates a new entry handler and indexes this handler in xcache (insert).

In either case, the returned handler is used for later references to the object's data.

The indexing operation may not always succeed, since the cache may be full. In this case, we store the original request in a cache-io struct and wait until an entry becomes evicted. This is covered in the ”Conditional execution” step.

Concurrency control

After cached has managed to acquire a valid entry handler for the object, it enqueues the appropriate read or write work to the object's workq. Once this job is executed, it runs exclusively and can modify safely the object's data.

The first thing that cached attempts to do is to claim the necessary buckets for the request. Knowing the following: i) bucket size, ii) request size and iii) request offset, cached can determine the bucket range that it must claim.

To claim a bucket, cached pops bucket handlers from the bucket pool. If the pool has been depleted, the request cannot proceed and it must wait until a bucket is freed. Again, this is covered in the ”Conditional execution” step.

Once cached has claimed the necessary buckets, it can perform the requested operation on them. The way an operation is executed however, is tightly coupled with the cache write policy. Moreover, depending on the cache write policy, cached may need to issue a new request for the object's data. Thus, the request handling step inevitably blends in this step.

There are four possible combinations, which are in part explained in Section 5.3.2:

1. **{Write operation, write-back policy:}**
   The data are written to the object, they are marked as dirty and the request is completed.

2. **{Write operation, write-through policy:}**
   Cached forwards the write request to the blocker. Once the request returns, it fills the claimed buckets with the object data and completes the request.

3. **{Read operation, write-back policy:}**
   {Read operation, write-through policy:}
   The behavior for reads is the same, regardless of the policy. Essentially, if the requested data are cached, cached can instantly complete the request. Else, it creates a new request, using cache-io, and sends it to the blocker. After the request is replied, it loads the missing data in the claimed buckets and then it completes the original request.

5.4.2 Optional steps

Conditional execution

As we have seen in the previous steps, since resources are finite, the requests cannot always be processed immediately. The common way that cached handles these cases is through the usage of xwaitqs; the request enqueues to the xwaitq the function that wants to be called as well as its arguments.

Data propagation

When an object is evicted, cached does the following:

1. It quickly checks if the object is dirty or not (using the cache entry finalization hook)
2. If the object is dirty, cached updates its reference count and it checks if there is a pending flush for it.
    i) If there is a pending flush, we wait until that flush is finished (waitq).
    ii) Else, we mark the buckets as **WRITING** and we flush their contents to the blocker.
3. After the flush has finished, we once again check if the object is dirty (the cache entry finalization hook is once more triggered):
    i) If the object is clean, then we do nothing and the object will be freed.
    ii) Else, we go back to step 2.

Chapter 6

Implementation of cached

In the previous chapter, we have discussed in length the design of cached and its components. In this chapter, we will present how the above design has been implemented. To aid us in this task, we will use code snippets from cached, and xcache and we will comment where necessary.

More specifically, Section 6.1 presents the implementation of xcache, the main cached component. Next, section 6.2 presents the implementation of cached and demonstrates the structures and functions that are used.

6.1 Implementation of xcache

For this section, we will attempt to provide a top-down view of the xcache implementation, starting from the functions that xcache exposes to peers and moving on to the more intrinsic details, such as the concurrency control.

6.1.1 xcache initialization

In order to use xcache, the peer must first initialize an xcache structure using `xcache_init`, which can be seen in Listing 6.1.

```
1 int xcache_init(struct xcache *cache, uint32_t xcache_size,
2         struct xcache_ops *ops, uint32_t flags, void *priv);
```

Listing 6.1: Definition of `xcache_init`

`xcache_init` requests the following information from the peer:

cache: Simply, an allocated xcache struct.

xcache_size: The number of objects xcache will index.

ops: The trigger functions for xcache's event hooks.

flags: Optional flags that tune the following two things:

1. The LRU algorithm. For the cached implementation, we use the O(1) LRU, but xcache also allows to use two more LRU algorithms, a binary heap (O(log(N))) or an LRU array (O(N)).

2. The usage of the hash table for evicted entries. Although our cached implementation relies heavily on it, this does not account for all the other peers that use xcache and by default is not used.

priv: A pointer (void *) to a structure that will be returned when an event hook is triggered. As most priv fields, it is irrelevant to the xcache struct and relevant only to the top caller. We initialize it with the peer struct.

The purpose of xcache_init is to process the above data, populate the xcache struct and create the necessary entities, such as the hash table, the cache entries etc. On Listing 6.2, we can view the xcache struct and its respective fields.

```
struct xcache {
    struct xlock lock;          /* Main xcache lock */
    uint32_t size;              /* Upper limit of entries */
    uint32_t nr_nodes;
    struct xq free_nodes;       /* Free cache nodes */
    xhash_t *entries;           /* Hash-table for active entries */
    xhash_t *rm_entries;        /* Hash-table for evicted entries */
    struct xlock rm_lock;       /* Lock for evicted entries */
    struct xcache_entry *nodes; /* Data segment */
    struct xcache_entry *lru;   /* O(1) lru implementation-specific */
    struct xcache_entry *mru;   /* O(1) lru implementation-specific */
    struct xcache_ops ops;      /* Hooks */
    uint32_t flags;             /* Flags */
    void *priv;                 /* Pointer to peer struct */
};
```

Listing 6.2: Main `xcache` struct

Each of the above `xcache` struct fields is used to implement a design feature that has already been discussed in Section 5.2.2. In the following sections, we will revisit these design features and present their implementation.

6.1.2 Cache entry preallocation

When xcache is initialized, it preallocates the necessary cache entries. The relevant xcache fields for this purpose can be seen in Listing 6.3.

```
struct xcache {
    ...
    uint32_t size;              /* Upper limit of entries */
    ...
    struct xq free_nodes;       /* Free cache nodes */
    ...
    struct xcache_entry *nodes; /* Data segment */
    ...
};
```

Listing 6.3: `xcache` struct fields for preallocated entries

The **size** field is the number of entries. The **free_nodes** is a stack where all entry indexes are initially pushed and subsequently popped when a new entry is inserted. Finally, **nodes** is the space allocated for the cache nodes and where the entry indexes point to.

Moreover, the definition of the `xcache_entry` struct is shown in Listing 6.4.

```
struct xcache_entry {
    struct xlock lock;                  /* Entry lock */
    volatile uint32_t parallel_puts;    /* Concurrency control */
    volatile uint32_t ref;              /* Reference counter */
    uint32_t state;                     /* Evicted or active state */
    char name[XSEG_MAX_TARGETLEN + 1];  /* Entry name */
    xbinheap_handler h;                 /* Index in data segment */
    struct xcache_entry *older;         /* Less recent entry in LRU queue
    */
    struct xcache_entry *younger;       /* More recent entry in LRU queue
    */
    void *priv;                         /* Pointer to data contents */
};
```

Listing 6.4: `xcache entry` struct

We will comment briefly on the relevant `xcache_entry` fields for this section, which can be seen in Listing 6.5. The rest of the fields will be discussed in the following sections.

```
struct xcache_entry {
    ...
    volatile uint32_t ref;              /* Reference counter */
    uint32_t state;                     /* Evicted or active state */
    char name[XSEG_MAX_TARGETLEN + 1];  /* Entry name */
    ...
    void *priv;                         /* Pointer to data contents */
};
```

Listing 6.5: `xcache entry` fields, relevant for preallocation

The description of the fields follows:

ref: The reference count of the entry, initially set to zero.

state: The state of the entry. It can either be ACTIVE or EVICTED and is initially set to the first.

name: The name of the entry. Since we cannot know its length beforehand, we allocate as much space as possible, typically 256 characters. During initialization, the entry name is cleared out of junk values.

priv: The private contents of the cache entry. On initialization, the cache node creation hook is triggered and cached initializes the private contents of cache entry with its data (more on Section 6.1.8)

6.1.3 Cache entry initialization

Before a peer can index a new entry, it must first allocate it from the cache entry pool and then initialize it. Xcache has a special function for this purpose, which can be seen in Listing 6.6.

This function attempts to claim an `xcache_entry` index from the `free_nodes` stack. Then it initializes it with the name given by the peer. Moreover, it triggers the cache entry initialization hook which cached uses to further initialize the entry.

```
xcache_handler xcache_alloc_init(struct xcache *cache, char *name);
```

Listing 6.6: Allocation/initialization function for `xcache_entry`

An added benefit of this function is that it doesn't need to acquire the main cache lock, so it does not slow down the indexing functions that rely on that lock.

6.1.4 Cache entry indexing

This is the core feature of xcache. In Listing 6.7, we present the fields of the xcache struct that are relevant to the indexing task.

```
struct xcache {
    struct xlock lock;          /* Main xcache lock */
    ...
    xhash_t *entries;           /* Hash-table for active entries */
    xhash_t *rm_entries;        /* Hash-table for evicted entries */
    struct xlock rm_lock;       /* Lock for evicted entries */
    ...
    struct xcache_entry *lru;   /* O(1) lru implementation-specific */
    struct xcache_entry *mru;   /* O(1) lru implementation-specific */
    ...
};
```

Listing 6.7: xcache struct fields for entry indexing

As we have mentioned in Section 5.2.2, we utilize two hash tables, one for the active entries and one for the evicted entries. These hash tables can be accessed from xcache and are the `*entries` and `*rm_entries` fields respectively.

More importantly, in Listing 6.8 we can see the functions that are related to indexing and xcache exposes to the peer:

```
xcache_handler xcache_lookup(struct xcache *cache, char *name);
xcache_handler xcache_insert(struct xcache *cache, xcache_handler h);
int xcache_remove(struct xcache *cache, xcache_handler h);
```

Listing 6.8: Indexing functions

All of these functions need a pointer to the xcache struct. Here's a brief description of them:

xcache_lookup: Takes the target's name as an argument and searches for it in cache.
Returns on failure: NoEntry
Returns on success: the requested handler.
Note: Looks only in `entries`.

xcache_insert: Takes the handler of an allocated entry as an argument and uses it to index that entry.
Returns on failure: NoEntry.
Returns on success: i) the same handler or, ii) if the same entry already exists in cache, the handler of that entry.
Note: It looks up first if the entry exist in `entries` or `rm_entries`. The later case can lead to re-insertions.

xcache_remove: Takes the handler of an allocated entry as an argument and uses it to remove that entry.
Returns on failure: -1.
Returns on success: 0.
Note: Removes only active entries .

Moreover, in Listing 6.9, we show the `xcache_entry` fields that are related to indexing, and comment on how they are used by each function.

```
struct xcache_entry {
    ...
    volatile uint32_t ref;              /* Reference counter */
    uint32_t state;                     /* Evicted or active state */
    ...
    struct xcache_entry *older;         /* Less recent entry in LRU queue
    */
    struct xcache_entry *younger;       /* More recent entry in LRU queue
    */
};
```

Listing 6.9: xcache entry struct relevant indexing

ref: The reference counter of an object is increased on lookups and inserts, since it is essentially referenced in these operations.

state: The state of the entry is set to ACTIVE.

older/younger : These fields show the neighbors of the entry in the LRU queue. The LRU queue is sorted by reference time order, so the neighbors are essentially the entries that have been referenced right before and right after our entry.

6.1.5 Entry eviction

The relevant fields for this purpose can be seen in Listing 6.10.

```
struct xcache {
    ...
    struct xq free_nodes;       /* Free cache nodes */
    xhash_t *entries;           /* Hash-table for active entries */
    xhash_t *rm_entries;        /* Hash-table for evicted entries */
    struct xlock rm_lock;       /* Lock for rm_entries */
    ...
    struct xcache_entry *lru;   /* O(1) lru implementation-specific */
    struct xcache_entry *mru;   /* O(1) lru implementation-specific */
    ...
};
```

Listing 6.10: `xcache` struct fields for eviction

As we have mentioned in Section 5.2.2, we resort to eviction when the cache is full and new entries can't be inserted. By xcache policy, we evict the least recently used entry. The necessary fields for the doubly-linked list that we maintain for this purpose can be seen below:

```
struct xcache {
    ...
    struct xcache_entry *lru;   /* O(1) lru implementation-specific */
    struct xcache_entry *mru;   /* O(1) lru implementation-specific */
    ...
};

struct xcache_entry {
    ...
    struct xcache_entry *older;         /* Less recent entry in LRU queue
    */
    struct xcache_entry *younger;       /* More recent entry in LRU queue
    */
    ...
};
```

Listing 6.11: Doubly-linked LRU list

The last entry of the list (oldest) is usually the LRU. When an object is referenced, it can be instantly transferred to the head of the list (MRU), since we know its position via the hash table (alternatively, we would need to search all entries, which would require O(N) time).

Another feature of this LRU queue is that it doesn't require timestamps, so we can avoid the unnecessary system call.

When we have found out which entry to evict, we migrate it from the hash table of active entries to the hash table of evicted entries. This process is explained in depth in Section 5.2.2.

Finally, when a cache entry is evicted from the hash table, it triggers the cache entry eviction hook.

6.1.6 Entry reinsertion

We have seen in the previous section how an active entry is evicted. However, what happens when xcache receives a request for an evicted entry which hasn't been freed yet?

In this case, the entry migrates back to the hash table of active entries. Also, its reference counter is incremented by 1, since the xcache can now reference this entry again.

The advantage of reinsertion is that we do not stall until the evicted entry has flushed all its data and instead, we can use it immediately.

6.1.7 Concurrency control

Concurrency control is an extremely important aspect of xcache, if we want to utilize an SMP system to its full potential. Although parts of the concurrency control have already been discussed in previous sections, in this section, we will provide an in-depth explanation of how xcache implements them.

The relevant fields for concurrency control can be seen in Listing 6.12, both for the xcache and `xcache_entry` structs.

There are three main techniques xcache uses for concurrency control. The first one is the usage of locks, the second one is reference counting and the third one is the tracking of parallel puts to an entry, a more esoteric method that counters the ABA problem.

```
struct xcache {
    struct xlock lock;          /* Main xcache lock */
    ...
    struct xlock rm_lock;       /* Lock for rm_entries */
    ...
};

struct xcache_entry {
    struct xlock lock;                  /* Entry lock */
    volatile uint32_t parallel_puts;    /* Concurrency control */
    volatile uint32_t ref;              /* Reference counter */
    ...
};
```

Listing 6.12: Concurrency control fields

Locking

With xcache, we have tried not to use a BKL [1] type of lock, but instead use many smaller ones.

Specifically, we have used:

1. a lock that protects the cache entry pool from concurrent accesses. Since the only operation this lock protects is the push and pop of cache entry indexes, we expect that there will be no contention on it.
2. the `lock` lock, as seen in the xcache struct, which is our main lock as it protects the hash table of active entries (`entries`) from concurrent accesses. This lock is used during lookups, inserts and evictions, so it is the lock with the most contention.
3. the `rm_lock`, which protects the hash table of evicted entries (`rm_entries`) from concurrent accesses and is used during insertions, evictions and puts.
4. a lock in every entry, which is specifically used when an entry is put.

Of major importance is also the issue of deadlocking. More specifically, during inserts or evictions, we need to have access on both hash tables. If a thread acquired the lock of one hash table and another thread acquired simultaneously the lock of the other, we would have a deadlock since both would need a lock that the other thread has.

To this end, xcache strictly acquires the locks in the following order: `lock` –> `rm_lock` –> entry lock. With this policy we are sure that there will be no deadlocks.

Reference counting

Each cache entry has a volatile uint64_t field which is atomically get and put. The type is volatile to inform the compiler that it might be changed outside the current execution context and therefore do not cache it in a register.

Furthermore, the atomic gets and puts are executed using the GCC builtins[9] which are shown in Listing 6.13.

The refcount model in xcache should be familiar to most people:

[1] BKL stands for Big Kernel Lock and was a giant lock in kernel space that inhibited the performance of SMP systems and remained until the late stages of the 2.6 Linux kernel

```
type __sync_add_and_fetch (type *ptr, type value, ...)
type __sync_sub_and_fetch (type *ptr, type value, ...)
```

Listing 6.13: Atomic operations of GCC

- When an entry is inserted in cache, xcache holds a reference to it (ref = 1).
- Whenever a new lookup for this cache entry succeeds, the reference is increased by 1 (ref++)
- When a request has been completed and no longer needs the entry, the reference is decreased by 1. (ref–)
- When a cache entry is evicted by xcache, its ref is decreased by 1, since it no longer holds a reference to it. (ref–)

Moreover, some common refcount cases are:

- active entry with pending jobs (ref > 1)
- active entry with no pending jobs (ref = 1)
- evicted entry with pending jobs (ref > 0)
- evicted entry with no pending jobs (ref = 0)

Unlike most refcount cases, however, the entry is not put when its refcount drops to zero. The reason is that the entry can be reinserted at any time. In the following section, we explain how we have handled that case.

Parallel puts

The scenario of putting the entry has proved the most tricky one and deserves its own section in the concurrency control implementation.

For this scenario, we aimed to avoid the usage of our two biggest locks: `lock` and `rm_lock`. Avoiding the first one was easy since the hash table of active entries was not used. However, the same did not hold true for the `rm_lock`, since the evicted entry must be reliably removed from the hash table of evicted entries. However, we have managed to reduce its usage to the absolute minimum.

The put procedure is the following:

1. We decrement the refcount of the entry.
2. If the refcount is more than zero, we leave.
3. Else, we proceed by triggering the on_finalize hook. With this hook, the peer is given a chance to increment the refcount of the entry if it deems it necessary (e.g. the entry is dirty).
4. After we have returned from this hook, we check if the refcount is zero. This time, we use the `rm_lock` to prevent reinsertions and remove the entry from the hash table of evicted entries.
5. If the refcount is not zero, then someone in the meantime has reinserted the entry and we can safely leave.

6. Else, we remove the entry.

This procedure looks simple at first glance. However, looks are deceiving and so is the ABA problem. In our case, consider the following:

1. Tread 1 → evicts a clean entry **(ref 0)**
2. Thread 1 → triggers the on_finalize hook
3. Thread 2 → reinserts the entry **(ref 2)**
4. Thread 2 → makes the entry dirty and then puts it **(ref 1)**
5. Thread 3 → evicts the now dirty entry **(ref 0)**
6. Thread 1 → checks if the reference count is zero and removes the entry
7. Thread 3 → triggers the on_finalize hook, increments the refcount of the entry and issues a flush for its contents **(ref 1)**

Although the final refcount is correct, the entry has been removed midway the put procedure. The fault in our logic was that we expected that, if the refcount is zero before and after the on_finalize hook, we could safely remove the entry. This is commonly referred to in the bibliography as the ”ABA problem”[5]. Generally, it occurs when a thread reads twice a ”guard” variable and gets the same value, but is unaware of the fact that it has changed in between these two checks.

The ABA problem is mostly encountered in lock-free structures and since xcache does not aim to be one of them, we can simply lock the put procedure with the `rm_lock`. However, this can have a performance impact, especially if a peer plugs a slow on_finalize trigger function.

We have chosen instead to use an extra `xcache_entry` field called `parallel_puts`. This number is incremented when a thread enters the put function and has dropped the reference of the object to zero i.e. between steps one and two. This way, we can also check during step five the number of parallel puts if and leave if it is more than one.

6.1.8 Event hooks

The hooks that xcache provides to users are stored in an xcache_ops struct that can be seen in Listing 6.14.

```
struct xcache_ops {
    void *(*on_node_init)(void *cache_data, void *data_handler);
    int (*on_init)(void *cache_data, void *user_data);
    int (*on_evict)(void *cache_data, void *evicted_user_data);
    void (*on_reinsert)(void *cache_data, void *user_data);
    int (*on_finalize)(void *cache_data, void *evicted_user_data);
    void (*on_put)(void *cache_data, void *user_data);
    void (*on_free)(void *cache_data, void *user_data);
};
```

Listing 6.14: xcache_ops struct

The design of these hooks has been presented on Section 5.2.2. The functions that are attached to each hook return two values: 1. the private field of xcache (the peer structure in our case) and 2. the cache entry’s private data (the object in our case) for which the hook was triggered.

6.2 Implementation of cached

In this section, we will present the implementation of cached. A presentation of the design of cached is provided in Section 5.3. Similarly to xcache, we will begin with the initialization process, we will continue with the request handling and finish with presenting the challenges we faced and the solutions we implemented.

6.2.1 Cached initialization

We have mentioned in the previous chapters that cached can be multi-threaded, have different write policies, maximum number of objects, cache size etc. All these variables are given from command-line and used during cached initialization. The necessary arguments are:

- Number of threads
- Max objects to cache
- Total cache size
- Object size
- Bucket size
- Blocker port
- Write policy

and the cached structure that is initialized is presented in Listing 6.15.

Moreover, on cached initialization we also initialize xcache as well as the necessary xworkqs and xwaitqs.

Some of the above cached fields are the same with the command-line arguments and are self explanatory. We will briefly comment on the less obvious fields, which will be discussed in length in their respective sections.

cache: The initialized xcache struct is stored here.

max_req_size: The maximum request size that can be sent to the blocker.

workq: A lockless xworkq where non-critical jobs from threads who are in a critical section are enqueued.

pending_waitq: An xwaitq for jobs that need to allocate a cache entry to. continue

bucket_waitq: An xwaitq for jobs that need to allocate a bucket to continue.

req_waitq: An xwaitq for jobs that need to allocate a request to continue.

bucket_data: The bucket pool of cached, Its size is the total_size of cached.

bucket_indexes: The stack where bucket indexes are pushed.

```
struct cached {
    struct xcache *cache;           /* xcache struct */
    uint64_t total_size;            /* Total cache size (bytes) */
    uint64_t max_objects;           /* Max number of objects (plain) */
    uint64_t max_req_size;          /* Max request size to blocker (bytes
    ) */
    uint32_t object_size;           /* Max object size (bytes) */
    uint32_t bucket_size;           /* Bucket size (bytes) */
    uint32_t buckets_per_object;    /* Max buckets per object (
    object_size / bucket_size) */
    xport bportno;                  /* Blocker port */
    int write_policy;               /* Cache write policy */
    struct xworkq workq;            /* xworkq for deferred jobs */
    struct xwaitq pending_waitq;    /* xwaitq for when cache entry pool
    is empty */
    struct xwaitq bucket_waitq;     /* xwaitq for when bucket pool is
    empty */
    struct xwaitq req_waitq;        /* xwaitq for when we are out of
    requests */
    unsigned char *bucket_data;     /* allocated space for buckets (
    bucket pool) */
    struct xq bucket_indexes;       /* stack of bucket indexes (bucket
    pool) */
};
```

Listing 6.15: Main cached struct

```
struct ce {
    uint32_t status;                        /* ce status */
    uint32_t *bucket_alloc_status_counters; /* counters for bucket
    allocation status */
    uint32_t *bucket_data_status_counters;  /* counters for bucket data
    status */
    struct bucket *buckets;                 /* object buckets */
    struct xlock lock;                      /* ce lock */
    struct xworkq workq;                    /* xworkq for the entry */
    struct xwaitq pending_waitq;            /* xwaitq for pending
    requests on the entry */
    struct peer_req pr;                     /* Pre-allocated peer request
    */
};
```

Listing 6.16: Cached entry struct

Furthermore, during the xcache initialization that takes place inside the cached initialization, the cached node initialization hook is triggered and cached can create its objects, which are as many as the max objects. Programmatically, cached objects are called ”ce”s and their structure can be seen in Listing 6.16.

The explanation of the above fields follows:

status: The object status, as seen in Section 5.3.6.

lock: The lock for the ce’s and its buckets’ data.

workq: The xworkq that is used for concurrency control over parallel access to the ce’s and its buck-

ets' data. It uses the aforementioned lock.

pending_waitq: The xwaitq that is used when a request cannot be executed due to the ce's state. It will allow job executions only when the object is not in FLUSHING state.

We have intentionally left out the bucket related fields that will be discussed in length in Section 6.2.2.

6.2.2 Bucket pool

The initialization of the bucket pool is covered in Section 6.2.1. In this section, we will explain how this bucket pool is connected with the buckets of each ce.

When the cache node initialization hook is triggered, the ce's buckets are initialized. Essentially, this means we do (once only) the following:

1. First, we allocate an array of struct buckets. The array has `buckets_per_object` length, which is typically 1024 (4MB objects / 4KB bucket size). The struct bucket is a very simple struct and is presented in Listing 6.17.

```
struct bucket {
    unsigned char *data;
    uint32_t flags;
};
```

Listing 6.17: Bucket implementation

2. Second, we allocate two more arrays, the `bucket_alloc_status_counters` array and the `bucket_data_status_counters` array, whose length is the number of allocation states (2) and data states (5) respectively.
3. Third, we initialize each bucket's allocation state to FREE and data state to INVALID. The allocation and data state are stored in the `flags` section of struct bucket, which is actually a custom bit-field with support for variable field lengths.
4. Finally, we initialize all the counters to zero, besides the allocation counter for FREE buckets and the data counter for INVALID buckets, which are set to `buckets_per_object` (1024).

None of the above operations, however, interact with the bucket pool. This is because we do not initially attach the bucket indexes to the ce's buckets.

The way buckets are attached to the object is analogous to the way a function maps to its address space a large memory chunk which has previously allocated; the chunk is internally divided to smaller chunks and when the function attempts to "touch" them, it is trapped and then the buckets are mapped to the function's address space.

Similarly, when cached accepts a request for a target, the request's range is translated to bucket range. If any of the buckets within that range are not attached to the ce, the request is "trapped" and the needed buckets are claimed from the bucket pool.

Finally, the bucket claiming and release procedure is the following:

1. We pop a bucket index from the `bucket_indexes` stack,

2. We translate it to the actual data pointer and store it to the `data` field of the struct bucket,
3. When the bucket is released, we translate the data pointer back to the bucket index
4. We push the bucket index back to the bucket pool.

6.2.3 Request handling

Cached uses the request polling scheme that we have described in Section 3.3.1, with the addition of the following:

1. Checks for the state of the bucket pool. If the bucket pool has been depleted, we force flush the LRU entry to acquire its buckets.
2. Periodic signals to the cached's xworkq.

When a request is accepted/received, it is forwarded to the appropriate handle function based on its xcache operation type.

More specifically, for accepted (new) requests, we index the request target (object) and store its xcache handler on the request's cache-io and we proceed according to its operation type. For received requests, the request's cache-io holds the xcache handler for the object, so we can proceed immediately according to its operation type. The way the request is handled next is documented in Section 5.4.

Chapter 7

Performance evaluation of cached

"There are three kinds of lies:
lies, damned lies,
and ~~statistics~~ benchmarks."
Mark Twain (modernized)

It may seem as an ironic statement, considering that we are about to provide benchmark results for cached, but it is actually a valid one. In our case, we will try not to smear the next pages with diagrams but first explain the benchmarking methodology behind them, provide a concrete depiction of cached's performance under various workloads, and finally explain the results in depth.

The skeleton of this chapter is the following:

Section 7.1 explains the methodology behind our measurements. Section 7.2 provides details about the hardware on which we have conducted our benchmarks. Section 7.3 presents comparative benchmarks for cached and sosd and quantify the performance gain of our implementation. Section 7.4 sheds some light on the internals of cached with the use of several special-purpose synthetic benchmarks. Finally, Section 7.5 evaluates the actual performance of Archipelago and cached for a real VM and compares the results with the results of the previous sections.

7.1 Benchmark methodology

The benchmarks that have been executed and whose results are presented in this chapter, will be split in three categories, all of which have their own distinct goals:

The first category is the comparison between using cached **on top** of the sosd peer (sosd has been discussed in Section 3.4) and using solely the sosd peer.

In order to effectively compare the performance of cached and sosd, we must consider the following:

1. The comparison of the two peers should try to focus on what is the best performance that these peers can achieve for a series of tough workloads.
2. The circumstances under which both peers will be tested need not be thorough but challenging. For example, it may be interesting to test both peers against sequential requests, but i) such patterns are rarely a nuisance for production environments, ii) they do not stress the peers enough to provide something conclusive and iii) they are out of the scope of this section as there can be many tests of this kind and adding them all here will impede the document's readability.
3. Both peers must be tested under the same, reasonable workload, i.e a workload that can be encountered in production environments.

4. If a peer doesn't show a consistent behavior for a workload, it must be depicted in the results

Having the above in mind, the next step is to choose a suitable workload. This choice though is fairly straight-forward; in production environments, the most troublesome workload is the burst of small random reads/writes and is usually the most common one that is benchmarked.

One may ponder however, how many requests can be considered as a "burst" or which block size is considered as "small". Of course, there is not only one answer to this question so, we will work with ranges. For our workload, we will use block sizes ranging from 4KB to 64KB and parallel requests ranging from 4 to 16.

The second category deals solely with the inner-workings of cached and its behavior on different operation modes or states. Its aim is not to capture the performance against a tough workload, but to explain **why** this performance is observed and how each of the options affect it. In this category, we measure how threads impact the performance of cached or what impact (if any) does our index mechanism have.

The third category positions cached and sosd in a real-world scenario; an actual VM that is configured to have Archipelago as its backing storage and on which we conduct the same benchmarks as in the previous categories. Our aim in this category is to:

1. Measure under real circumstances the performance of the cached configurations that have been discussed in the previous scenarios.
2. Compare their results with the respective results of the previous categories.

This way, we will be able to check if the reported performance gain of the previous categories can be applied in a real-world scenario.

Finally, in the following sections, for brevity reasons, we will talk about comparing cached and sosd. What the reader must keep in mind however is that cached is essentially the cache layer above sosd, which means that we actually test sosd vs cached over sosd.

7.2 Specifications of test-bed

The specifications of the server on which we conducted our benchmarks is the following.

Component	Description
CPU	2 x Intel(R) Xeon(R) CPU E5645 @ 2.40GHz [7] Each CPU has six cores with Hyper-Threading enabled, which equals to 24 threads.
RAM	2 banks x 6 DIMMs PC3-10600 Peak transfer rate: 10660 MB/s

Table 7.1: Test-bed hardware specs

Software	Version
OS	Debian Squeeze
Linux kernel	3.2.0-0 (backported)
GCC	Debian 4.4.5-8

Table 7.2: Test-bed software specs

7.3 Performance comparison between cached and sosd

As mentioned above, for our first test, we will evaluate the read and write performance of cached and sosd for a random workload with parallel requests of small size. In order to measure accurately their performance, we will use two different metrics:

Bandwidth:
Bandwidth measures the maximum throughput that the application can sustain. This metric is usually used to indicate how much I/O (from various inputs) can an application handle within a second.

Latency:
Latency is the converse of bandwidth. It is a measurement from the viewpoint of the issuer of requests and indicates the responsiveness of the tested application. It is commonly calculated as the average reply time for a series of requests.

On the following sections, we present the benchmarks that we conducted for the first category. The first benchmark, which is shown in Section 7.3.1, shows the behavior of cached during a peak of I/O requests. The second benchmark, which is shown in Section 7.3.2, illustrates the behavior of cached under continuous load. On these sections, cached is always multi-threaded and uses 4 threads.

7.3.1 Workload smaller than cache size - Peak behavior

We begin with the bandwidth performance of our peers. The write performance can be seen in Figure 7.1, while the read performance can be seen in Figure 7.2.

Before we proceed with the interpretation of the diagram results, we will briefly comment on the diagram structure. Due to the fact that the performance of the two peers differs in at least two orders of magnitude, the results would look too flat in a conventional diagram that would scale from 0 to 11000. To amend this, we have broken the y-axis of our diagrams in two parts with different scales and starting values, in order to make the comparison easier to the eye.

The initial results look very promising. For write requests, the speedup for very small block sizes (4KB - 16KB) is approximately **100x** whereas for larger ones (32KB - 64KB) it ranges from **50x to 200x**. For read requests, the speedup for very small block sizes is approximately **50x**, whereas for larger block sizes, it ranges between **20x - 75x**.

These results not only illustrate the performance gap between RAM and HDDs but also show that our implementation manages to keep its bandwidth consistently over 1 GB/s in stress scenarios. However, we must keep in mind that we see here is only a part of the full picture, since we do not know yet how cached behaves past its cache size.

Moreover, upon closer inspection the following questions arise:

1) Where is the speedup difference between reads and writes attributed to?

We have mentioned above that the speedup for writes is between 50x and 200x whereas for reads is between 20x and 75x. Although these speedups are very big, they also show some of the limitations of our implementation.

We attribute the speedup differences to three factors:

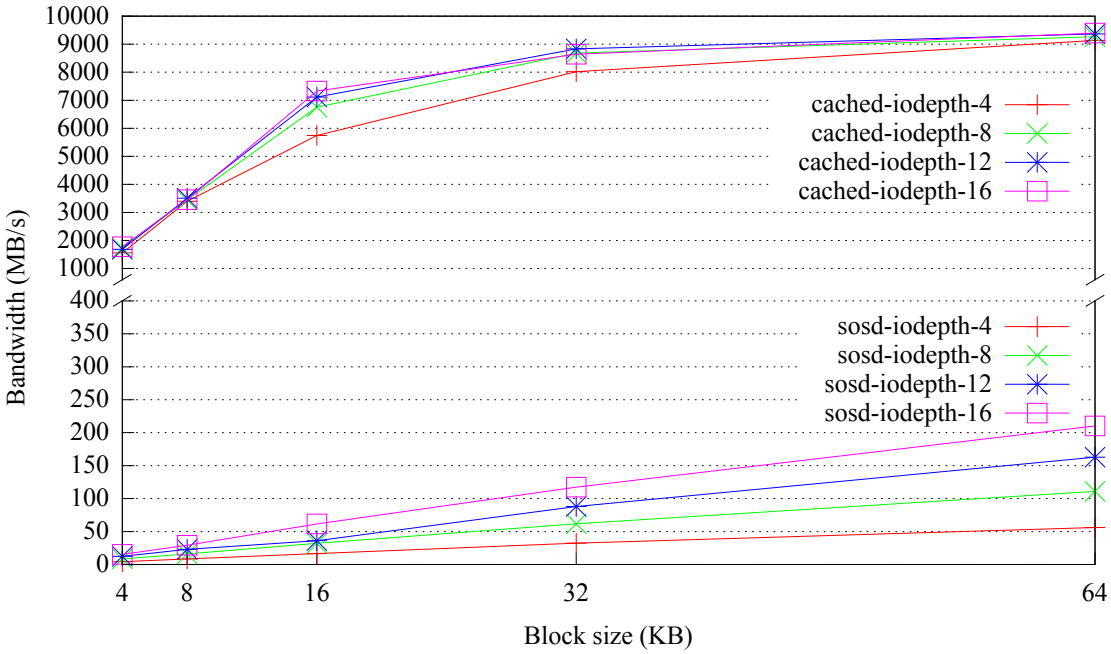

Figure 7.1: Comparison of bandwidth performance for write peaks

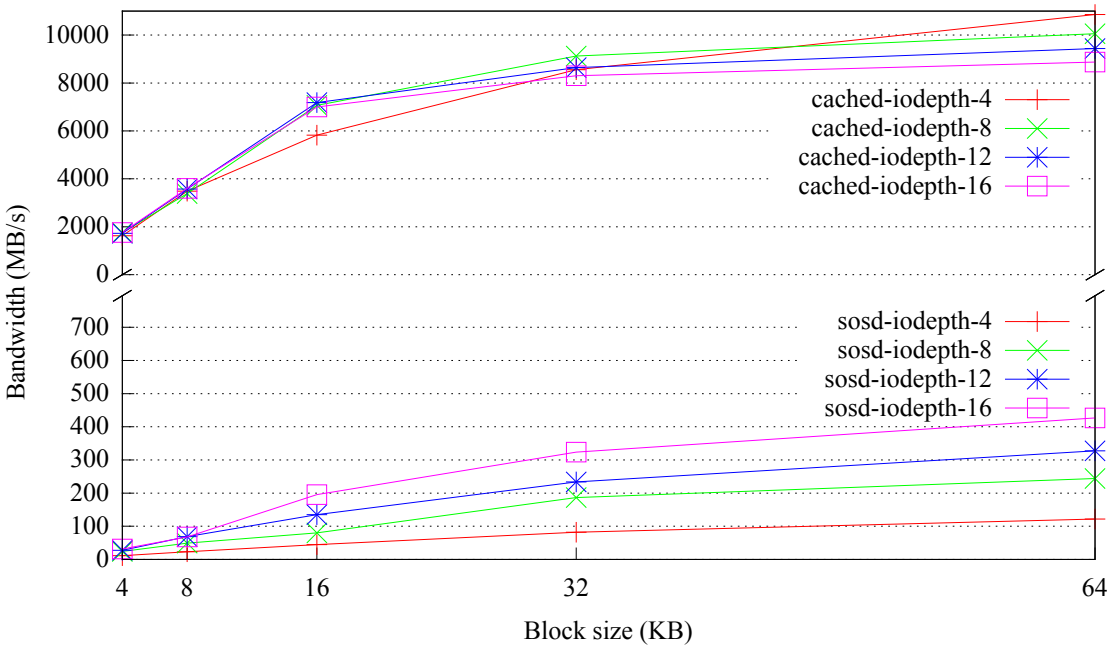

Figure 7.2: Comparison of bandwidth performance for read peaks

1. The performance of cached is almost the same for writes and reads. [1] This is expected behavior as the read and write paths for cached have many common parts, as seen in Section 5.4.

2. Cached doesn't scale much past the 16KB block size. This is an interesting observation with an unexpected answer. It may seem implausible at first, but what happens is that we are actually

[1] To be fair, reading seems a bit faster than writing, but that is probably attributed to CPU caches and the fact that reading from RAM is a non-destructive operation in contrast to writing.

hitting the bandwidth limit of the RAM modules. If we consult the Table 7.1, we can see that the bandwidth limit of our RAM is about ~10.7GB/s [2]. This limit is approached asymptotically as the block size increases and the index and concurrency overhead decreases

3. On reads, sosd is benefited from the existence of caches in various levels e.g. on OSD level, on RAID controller level.

To sum up, the cached's performance remains relatively the same in both reads and writes, it is merely the sosd that is getting faster in reads due to caches.

2) Why is cached's performance increased proportionally to the block size?

We expected that sosd's performance would increase proportionally to the block size, due to the rotational nature of hard disk drives, but why does this affect cached too? It is certainly not attributed to any `memcpy()` performance tricks, since we always write in bucket granularity, which means that a 16KB write is translated to 4 x 4KB writes.

From this observation, we extrapolate that the CPU operations such as indexing, job enqueuing, accepting and responding requests, that occur proportionally more times for smaller block sizes, dominate the cached's performance.

We can see this more clearly if we consult Table 7.3.

Bandwidth (MB/s)	IOPS	Latency (usec)	Iodepth	Block size (KB)
2652.327	678995.620	5.002	4	4
4427.89l	566769.888	6.176	4	8
5900.719	377646.042	9.672	4	16
6895.167	220645.338	17.179	4	32
7090.661	113450.584	34.237	4	64

Table 7.3: Write performance results for 2-threaded cached

On this table, we present the results for a benchmark that does writes, has four parallel write requests and its block size varies between 4KB and 64KB. Note that this benchmark is conducted on a cached that has **two** threads instead of four.

Comparing the results of this table with the results of Figure 7.1, and more specifically the "cached-iodepth4" trend line, we can observe two things:

1. The performance for small block sizes (4KB - 8KB) is better than the performance of cached with 4 threads.
2. The gap between the results for 4KB and 8KB is not 2x, as in Figure 7.1, but 1.5x.
3. The overall performance gap between the results for 4KB and 64KB is less than the gap in Figure 7.1.

To sum up, these results show that the lock contention in cached, due to the number of threads, can significantly dominate its performance.

3) Why cached's perforance does not improve proportionally to the parallel requests?

[2] A more careful look shows that we surpass this limit, which is expected given the fact that we have a multi channel RAM setup.

The reason why we see only a minor increase in the performance of cached, even though it is multi-threaded, is because our locking scheme is not fine-grained enough. We have a single lock for our request queue, a single lock for most of the hash table accesses and this inevitably causes a lot of threads to spin. This slight improvement we see is mainly due to the fact that requests are effectively being pipelined while waiting each other to release its lock.

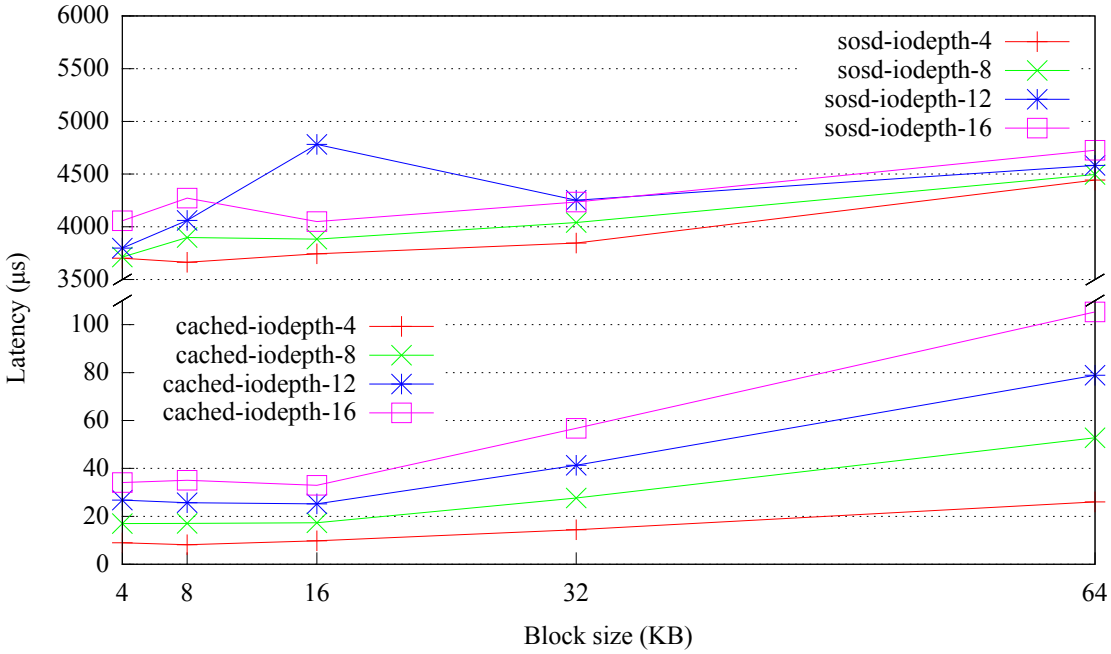

Figure 7.3: Comparison of latency performance for write peaks

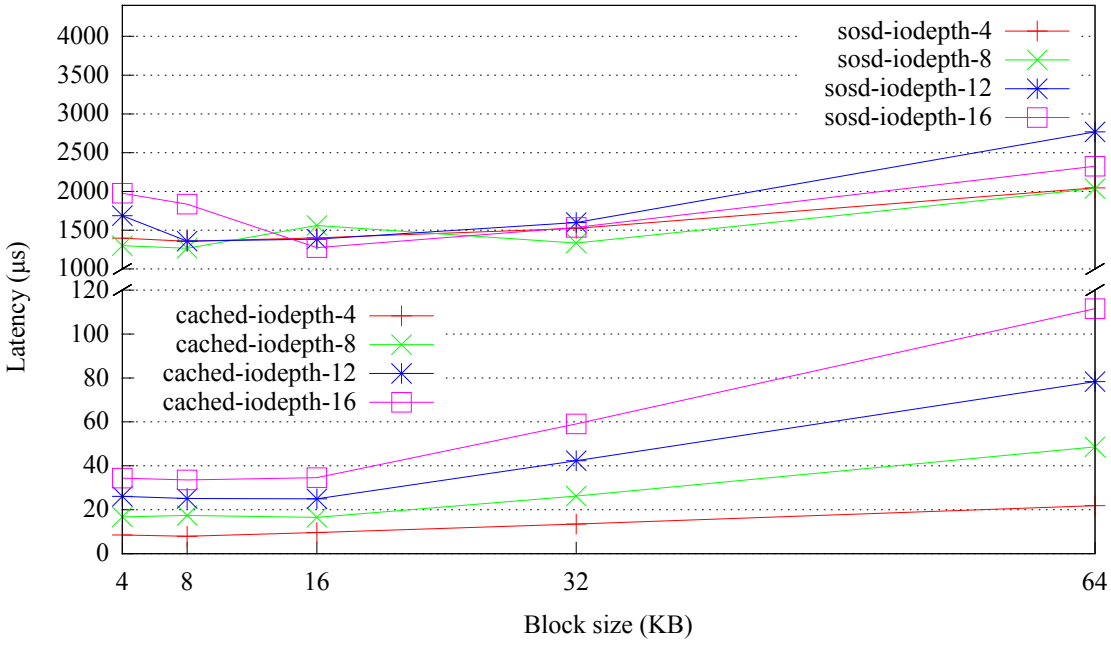

Figure 7.4: Comparison of latency performance for read peaks

We now proceed to the latency results. The write performance can be seen in Figure 7.3 while the read performance can be seen in Figure 7.4.

The latency results corroborate the observations that we made for the bandwidth results. As we can see, the latency of cached is proportional to the number of parallel requests, which reinforces the assumption that our locking scheme is not granular enough, whereas sosd has only a small variance.

7.3.2 Workload larger than cache size - Sustained behavior

We now proceed to the second part of the comparison between cached and sosd. On this part, we will once again evaluate their performance against a random workload with many parallel requests. Unlike the first part though, where the cache size was larger than the workload, on this part the cache size will only be half of it. Having a smaller cache than the workload is after all the most common scenario for production environments and the results should complete the picture of cached's behavior in a performance-intensive environment.

For this test, there are two main parameters we must take into account: the cache size and the maximum objects. These parameters have been decoupled in our implementation and we expect different results for each combination. More specifically, we have tested cached with half the workload size and:

1. half the objects of the workload (referenced as **cached-limited** from now on).
2. more than the objects of the workload (referenced as **cached-unlimited** from now on).

Also, we have tested cached in write-through mode, to see its behavior in sustained writes. For this mode, cached has been configured to store more than the objects of the workload (i.e. unlimited) but still has space for the half of it.

Furthermore, for this benchmark, we tested only the write performance of cached and sosd, since the evictions and subsequent flushes in reads would skew considerably the results. An exception to this is the write-through test, whose flushes do not produce extra writes to the storage backend. Also, we chose for our workload to use 16 parallel requests and 4KB block size, since this seemed the most troublesome workload in the previous benchmark.

Before we proceed to the results, we must explain first how the diagrams below are structured. Since cached's performance is fairly unstable in this scenario, we have chosen to illustrate it as the benchmark progresses. Thus, the x-axis shows what percent of the benchmark has been completed and for every 5%, we show the performance of cached for that part.

The bandwidth results can be seen in Figure 7.5, while the latency results can be seen in Figure 7.6. Due to the vast bandwidth drop of cached, the y-axis uses a logarithmic scale to show a more clear picture of what happens when the performance reaches the 10 MB/s mark.

These diagrams have several interesting points which we will address below.

1) Why does cached-unlimited's performance drop in the middle of the benchmark?

Since cached-unlimited can index more objects than the workload, there is no need to evict entries from the hash table to insert new ones. Cached-limited on the other hand, has at any point a 50% chance to receive a request for a target that does not index. Thus, when cached-unlimited starts to receive the first 50% of requests, it only needs to store them and, as we have seen in the first part of this section, it does so very fast.

However, at around 50% of the benchmark its space is depleted and thus, it has to manually evict entries. At this point, we see that its performance degenerates to the performance of cached-limited.

2) Why is cached-limited's performance significantly less than the sosd's performance?

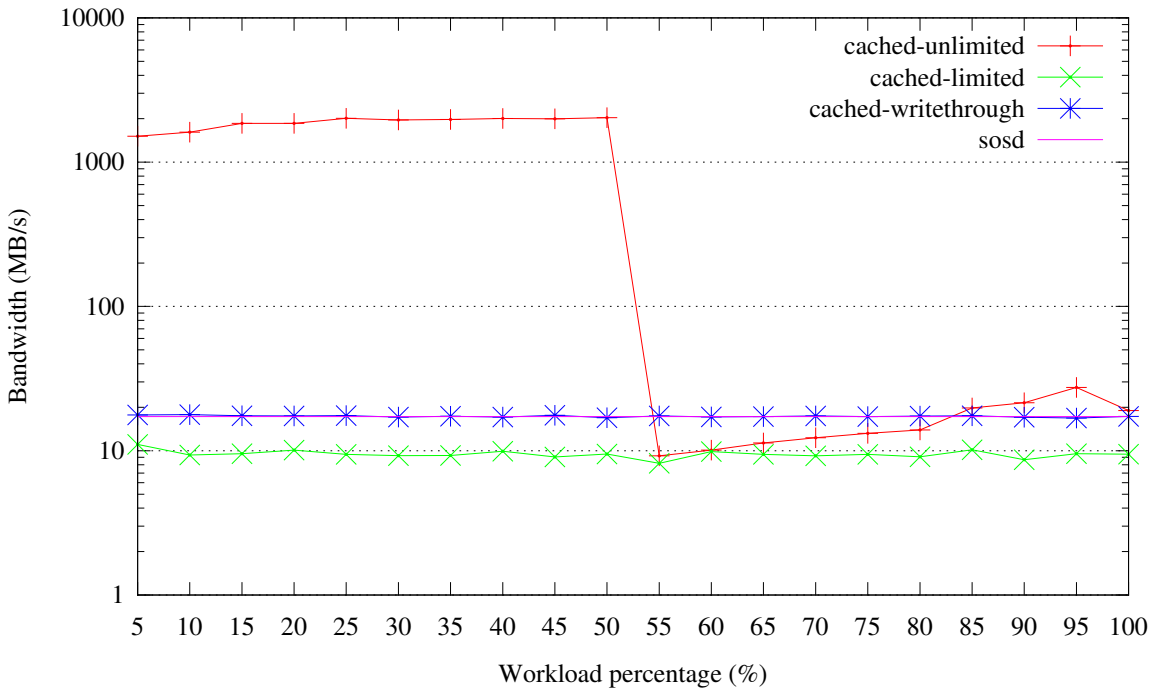

Figure 7.5: Comparison of bandwidth performance for sustained writes

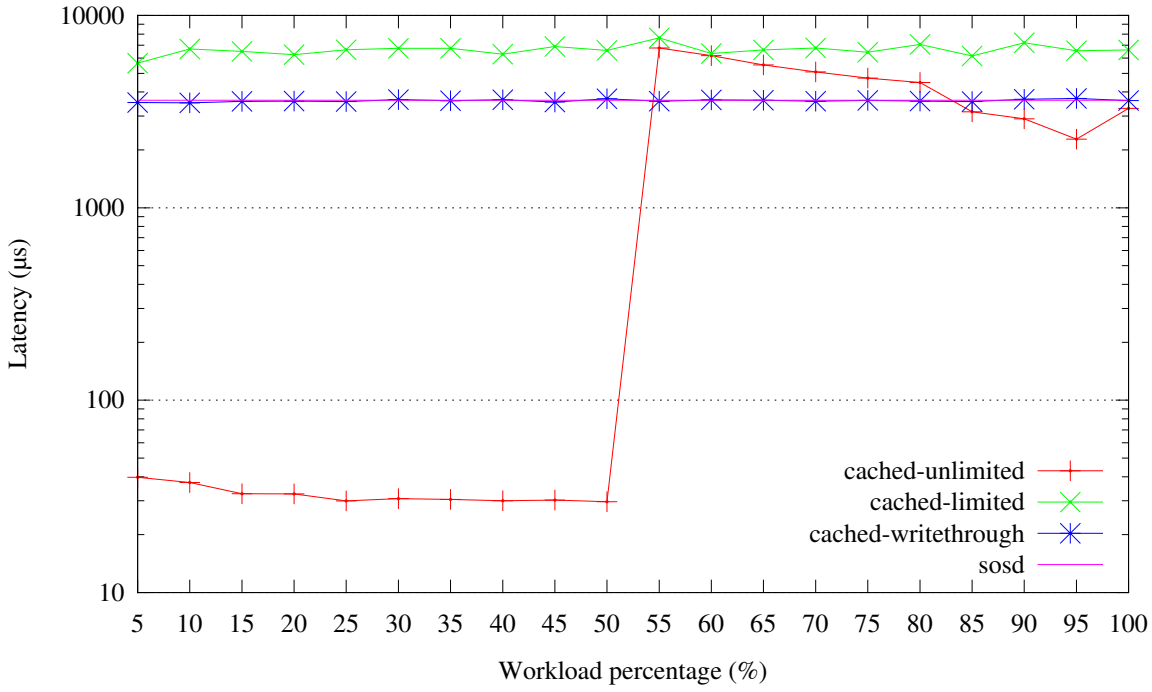

Figure 7.6: Comparison of latency performance for sustained writes

As we have explained above, cached-limited constantly evicts entries due to its object limitation. It may seem that the only cost of eviction is that the performance degenerates to the performance of sosd, but that is not entirely true.

When an entry is evicted, all of its contents must be flushed before the new entry can overwrite them. Moreover, the cost of concurrency control for hash table migrations and the cost of creating new requests and copying the data in them is not negligible. Finally, due to constant evictions, there are

only a few buckets cached for any entry, which in turn does not leave any margin for coalescence.

3) Why does cached-unlimited's performance increase consistently after the 50% mark ?

After the 50% mark, cached-unlimited will have to resort to evictions in order to store new data. Unlike cached-limited though, cached-unlimited has already cached half of the workload's data, which not only is favorable for coalescence, but also allows flushes to occur in a more sequential fashion. Moreover, cached evicts whole objects to get their buckets, which means that evictions will be less frequent that cached-limited, which constantly evicts entries to index other entries.

4) Why does cached in write-through mode perform the same as sosd?

The performance of cached in write-through mode is the same as sosd's since write requests are essentially forwarded to sosd and are cached when sosd responds. The caching has no significant overhead, compared to the waiting time for sosd, since there are no dirty entries and thus no costly evictions.

What is more interesting however is the comparison of read performance between cached and sosd, which can be seen in Table 7.4.

Peer	Bandwidth (MB/s)	IOPS	Latency (usec)	Iodepth	Block size (KB)
sosd	34.124	8735.699	1828.758	16	4
cached	44.897	11493.511	1389.511	16	4

Table 7.4: Read performance comparison of sosd and cached in write-through mode

The read performance of cached has a noticeable increase of 25%. This is expected, since cached serves read requests either from its cached data or from sosd. This means that it has the baseline read performance of sosd plus the extra performance gain from cache hits.

7.4 Performance evaluation of cached internals

To some extent, we have already discussed about the behavior of cached when we tweak various of its parameters, such as the over-subscription of its objects, the cache size or its write policy. In this category, we will further solidify our previous assumptions about the inner workings of cached by measuring:

1. how cached scales with the number of threads
2. the overhead of the index mechanism

Note that the tests will be run with the following parameters:

Mode Write-back

Block size 4KB

Cache size Always larger than benchmark size

Operation Writes

The above options have been chosen to isolate cached of any other factors that may alter its performance, such as evictions and delays due to sosd.

7.4.1 Cached performance per number of threads

Measuring the performance impact of multi-threading is nonsensical, if the application is not tested against many parallel requests. Hence, we will test each n-threaded cached configuration against a range of parallel requests and measure its performance.

The bandwidth results can be seen in Figure 7.7 whereas the latency results can be seen in Figure 7.8.

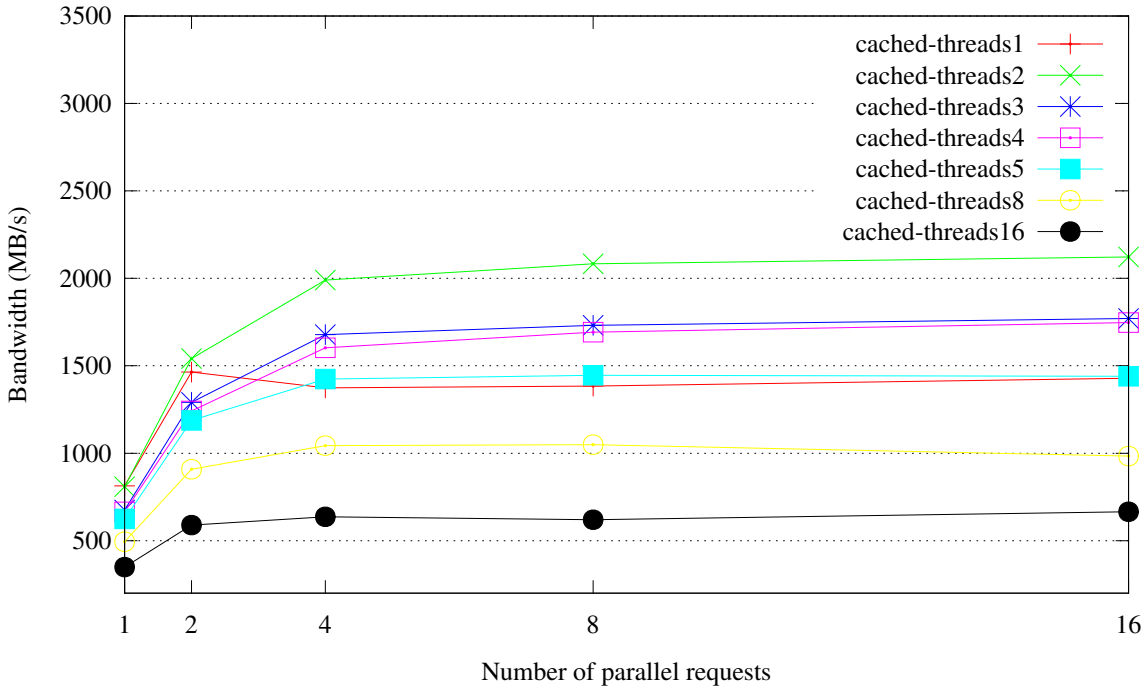

Figure 7.7: Bandwidth performance of cached per number of threads

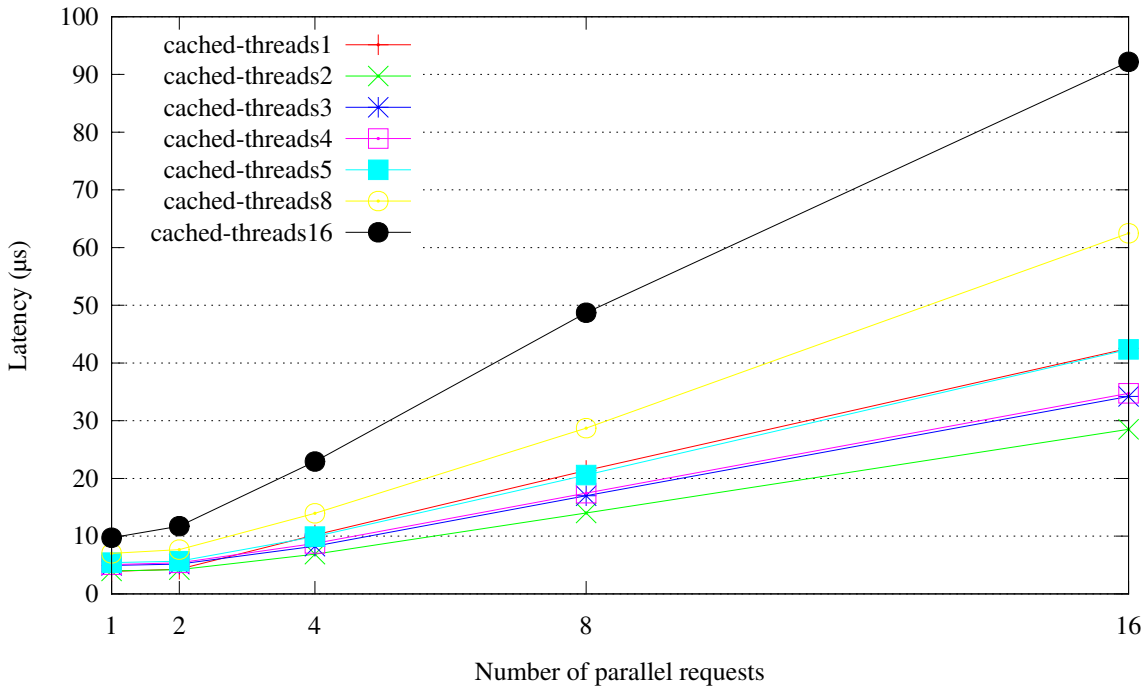

Figure 7.8: Latency performance of cached per number of threads

From these results, we derive the following conclusions:

1. Cached is generally benefited from multi-threading, when the amount of parallel requests is more than two. More specifically, we achieve a major performance improvement of up to 40% when using two threads in contrast to one, as well as a lower, but significant, performance improvement for up to four threads, as the number of parallel requests increases.
2. Cached does not scale past four parallel requests, regardless of the number of threads.
3. Adding more than two threads degrades significantly the performance of cached, when the number of parallel requests is small.

Finally, these results, along with the results of the first part, clearly show that we cannot fully utilize the multi-threading capabilities of cached unless we employ a more fine-grained locking scheme.

7.4.2 Cold cache vs Warm cache

This scenario will attempt to evaluate the overhead of cache misses for **write** operations. Theoretically, this should account to the overhead of adding new entries to cached and consecutively, an indication of the complexity of our index mechanism.

Our experimental procedure is the following:

1. We inserted 128K [3], 256K, 512K and 1M [4] objects in a cached peer configured to hold up to 2M objects and measured their average latency times (cold cache).
2. Subsequently, we attempted to insert these objects in cached again and measured their average latency times (warm cache).

The difference between the first and the second iteration is that in the second iteration, the objects are hot in the cache and thus cached will not need to insert them. This means that the first iteration has the extra actions of allocating the entries and inserting them in the cache, whereas the second iteration will only lookup the objects in the cache.

We expect that the extra indexing action will add a constant overhead to the measurements of the first iteration, regardless of the number of entries. Moreover, the latency times for the second iteration should remain the same, regardless of the number of entries.

Note: To get the most accurate results and since we want to test just the overhead of our indexing mechanism, cached has been configured to use one thread. Moreover, the I/O depth is also one, to make sure that latency times are not skewed by the time the request waits in the request queue.

We present the results in Figure 7.9.

The focal point in these results is that the write latency, either of cold or warm caches, remains practically the same as the number of objects increases. This reinforces our statement that cached has average case O(1) complexity.

Finally, an interesting finding from our results is that the latency of indexing an entry, even in a hash table that supports 2 million objects (and potentially 8TBs of data), is on average less than 1µs, which is a very strong feature of our implementation.

[3] where **K** is **x1024** objects
[4] where **M** is **x1024²** objects

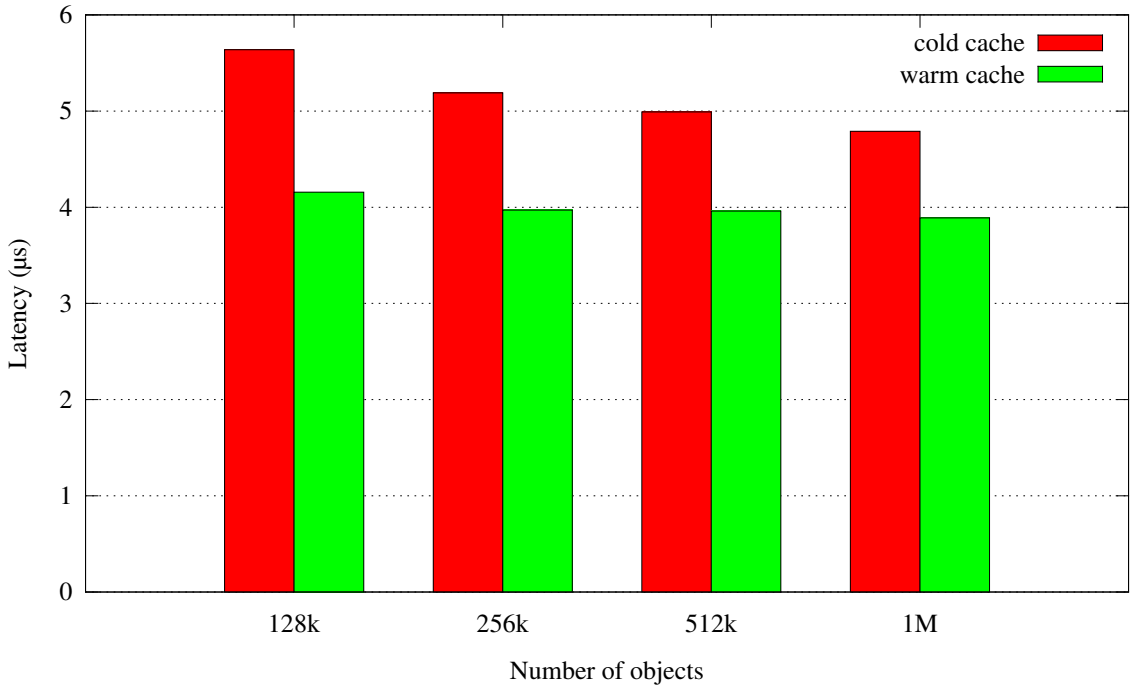

Figure 7.9: Latency performance of cold/warm cache per number of objects

7.5 Performance evaluation of a VM over Archipelago and cached

In this category, we will attempt to: i) create a VM using Archipelago as its storage backend with cached support and ii) use it to evaluate the real-life performance of various Archipelago configurations, with and without cached.

The benchmarks of this section have been conducted on a vanilla Ubuntu 13.04 VM. Once the VM was started successfully (which was an important feat in its own right), we installed FIO [5] , which is a fully-featured benchmark suite created by Jens Axboe, the maintainer of the Linux block layer.

Our aim was to simulate the workloads of the previous categories, in order to produce comparable results. The FIO job file that was created for this purpose is presented in Listing 7.1:

The workload that this job file describes has been uniformly issued for every Archipelago configuration that we have tested. We will illustrate this workload through an explanation of the job file parameters, which appear on the **[global]** section:

- The request block size is **4KB** and is aligned on a 4KB boundary,
- the total benchmark size is **1GB**,
- the number of parallel requests is **16**,
- the I/O requests bypass the VM's page cache, to ensure that the results are not skewed due to guest caching.

The following two sections, **[randwrite]** and **[randread]**, describe the two benchmark tests that were conducted. More specifically, in the first test we issue random writes that span the file and in the second test, we randomly read the written data.

[5] Flexible I/O tester, http://git.kernel.dk/?p=fio.git;a=summary

```
[global]
ioengine=libaio
bs=4k
ba=4k
direct=1
iodepth=16
size=1G
filename=job.tmp
filesize=2G

[randwrite]
rw=randwrite
stonewall

[randwrite]
rw=randread
stonewall
```

Listing 7.1: FIO job file

Moreover, the fact that the benchmark is executed on a file means the following two things:

1. That we are bound to include in our measurements the overhead of filesystem operations and I/O scheduler of the VM. These overheads however are essential for our benchmarks since they provide a more accurate measurement of the I/O speed that the common user is likely to experience.

2. That we are guaranteed that only the first write to the file will be affected by Copy-On-Write delays, while the subsequent writes will behave normally. Thus, we can discard the first write and use only the subsequent ones for our measurements.

Finally, in addition to the bypass of the VM's page cache, we have also ordered the hypervisor to not use the page cache of the host. The only exception to this is the page cache test that evaluates the performance of the VM using the host's page cache in writeback mode.

We can now present the bandwidth and latency results for our benchmarks, which are illustrated in Figures 7.10 and 7.11 respectively.

We will explain which scenario each label represents:

page-cache
: This scenario tests solely the page cache of the **host** machine. The rest of the scenarios bypass this page cache.

sosd
: This scenario tests solely the performance of sosd peer.

cached-vast
: This scenario uses cached with a large enough cache size, which means that no evictions occur.

cached-unlimited
: This scenario uses cached with half of the necessary space for the benchmark but with the ability to cache practically unlimited objects.

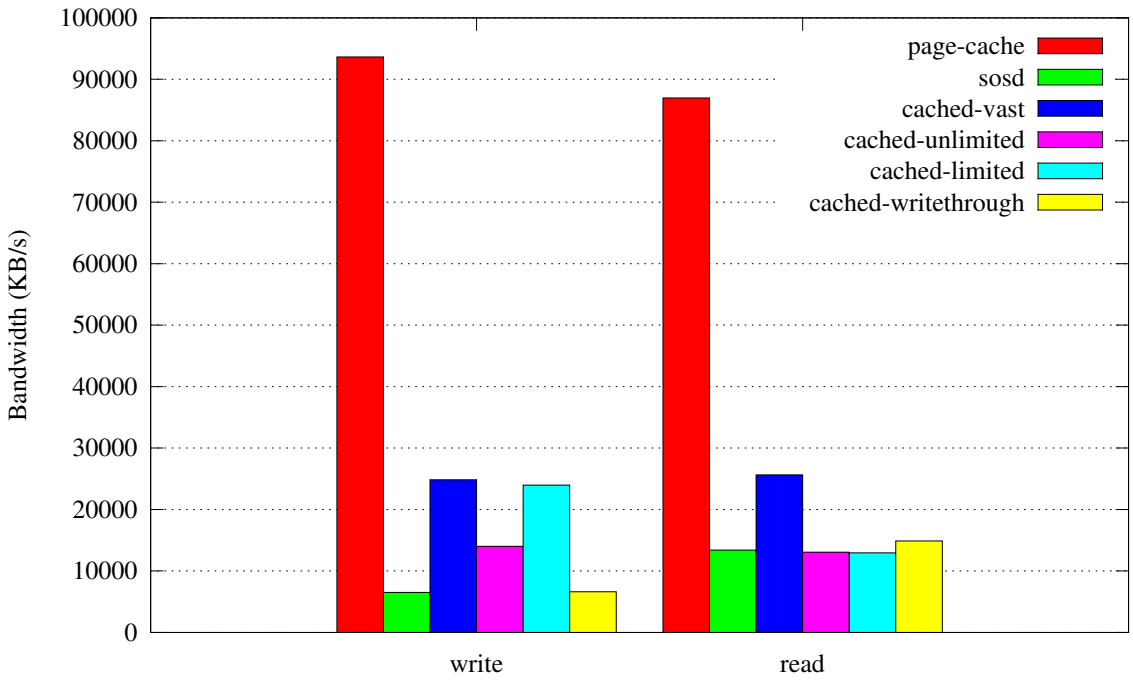

Figure 7.10: Bandwidth performance of a VM over Archipelago

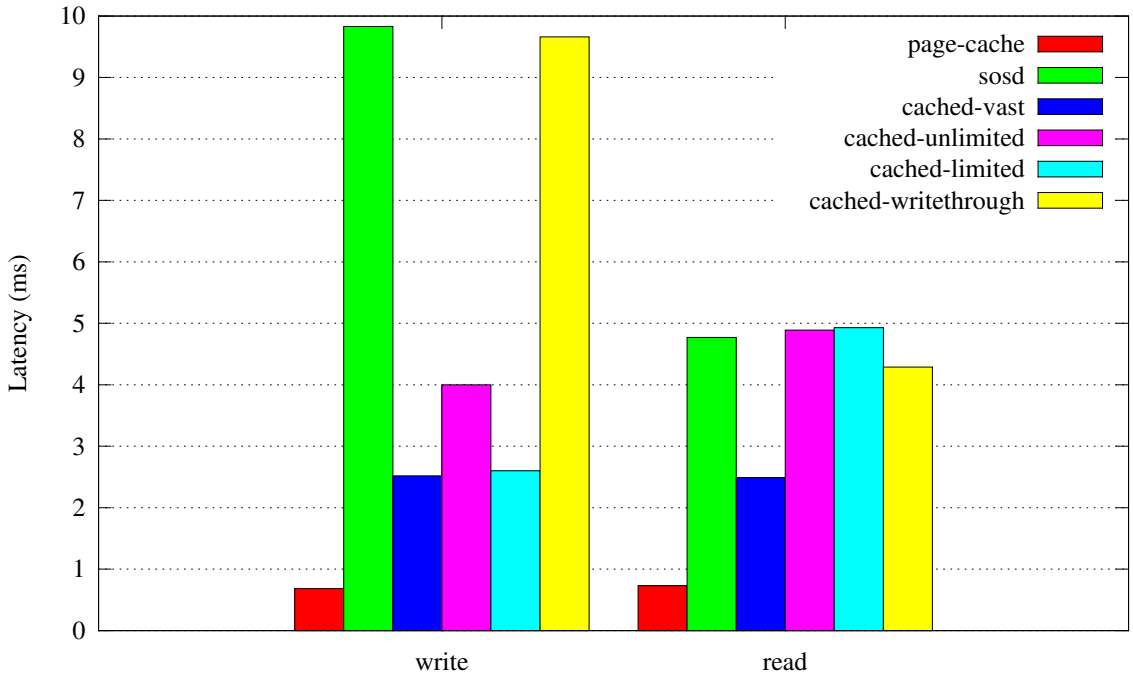

Figure 7.11: Latency performance of a VM over Archipelago

cached-limited
: This scenario uses cached with half of the necessary space as well as the ability to cache half of the necessary objects.

cached-writethrough
: This scenario uses cached with half of the necessary space, unlimited objects and in write-through mode.

Note that we have purposefully combined the write and read results on the same diagrams to make comparisons between them more evident. We will proceed with commenting on the performance results of each tested Archipelago configuration and we will invoke the previous synthetic benchmarks where necessary.

page-cache

Our measurements clearly indicate that the page cache outperforms every Archipelago configuration. It also sets the mark for the upper speed limit that the Archipelago must strive to achieve.

Since page cache stores data in DRAM, we would expect an even better performance, given that our implementation manages to achieve latencies smaller than 40μs, for the same number of parallel requests and block size, which can be seen in Figure 7.3. We attribute this speed difference of < 1ms to the overhead of the paravirtualized storage, as well as the overhead of the VM operations (e.g. the filesystem, scheduler and kernel operations).

sosd

The results for sosd are lower than the expected results. In fact, sosd has an additional latency of about 7ms for writes and 3ms for reads, which cannot be attributed to the < 1ms difference that we have observed in the page cache test. This means that besides the overhead that is incurred from the VM operations, there is also the latency of Archipelago, specifically of xsegbd, vlmcd and mapperd that has a performance penalty which is in the order of milliseconds.

cached-vast

The comparison between sosd and this scenario shows that the performance speedup is approximately 4x for writes and 2x for reads. This is a very reinforcing result for our implementation and it proves that it does make a difference in an actual VM.

Arguably, the performance speedup that we measured in the synthetic benchmarks of previous categories was much bigger, yet we must keep in mind that the latency of the levels above cached do not allow this performance difference to show. This statement is further reinforced by the comparison between these results and the results of the page-cache scenario; in both scenarios, data is cached in DRAM, which means that the main latency difference between them is the latency of Archipelago, which is approximately 2ms.

cached-unlimited

The results of this scenario were expected, given the results of the previous scenario (cached-vast) and the synthetic benchmarks of Figures 7.6 and 7.5.

For the first 50% of the workload, the performance of cached-unlimited was the same as the performance of cached-vast. Predictably, it dropped after the 50% mark but, interestingly, it was consistently better than the performance of sosd until the end, something that was not observed in the synthetic benchmarks we had conducted. This is attributed to the fact that background flushes to sosd are supposed to take on average 3-4ms (see Figure 7.6), which is close to the time that a VM request needs to reach cached. This means that evictions that happen in the background are essentially masked under the latency of the system.

We must also add that this scenario also achieves a **sustained** 2.5x performance speedup for writes, while the peak behavior of this configuration is expected to be the same as cached-vast's. This shows that, even with a small cache, we can achieve a major performance improvement over sosd.

Finally, we observe that the read performance is the same as sosd's but, as we have already explained, this happens due to background flushes that take place during the read operation, so the results are bound to be skewed.

cached-limited

The revelation of our benchmarks has been this scenario, which is just as fast as cached-vast for sustained writes. This is completely different from the results we observed in Figures 7.6 and 7.6. It seems that the system latency completely masks the background flushes that take place, which is probably due to the fact that evictions are more frequent but less batched.

cached-writethrough

This scenario had predictable results. The write speed in writethrough mode was the same as the write speed of sosd whereas the read speed was a bit better, due to the fact that the data were cached. Overall, there was no particular performance improvement in writethrough mode.

Chapter 8

The synapsed peer

On the previous sections, we have evaluated the design of cached and one of the conclusions that we have reached at is that it has heavy lock contention when more than one threads are used. The implications of this, however, are bigger, if we consider that Archipelago is running in the host machine, whose CPU's are already oversubscribed. This means that cached must compete for CPU time against the running VMs, essentially defeating the QoS purpose it serves.

On the other hand, on the RADOS nodes, the CPUs and RAM are not used to their full potential. If we could run cached or part of the Archipelago in these nodes, we would have the following benefits:

- We would have access to a much larger amount of RAM
- We would make a big step forward in terms of creating a distributed cache, since we would be able to replicate the data to two or more cached peers that would ran in different nodes, since these nodes are accessible from any host.

To this end, we have created a network peer called synapsed (from **synapse d**aemon) as a proof-of-concept, that should be able to accept read/write XSEG requests and send them through network to another Archipelago segment. We must note that this peer, given that it is proof-of-concept, has not been created with high-performance in mind but its main aim is to provide the functionality needed for our purposes. As a consequence, we have not used tools such as ZeroMQ or libevent that would boost the performance of the implementation.

The structure of this chapter is the following. Section 8.1 presents the design of the synapsed peer. Section 8.2 explains how we implemented synapsed as well as the challenges that we have faced. Finally, Section 8.3 provides the results of some preliminary benchmarks that we have conducted using synapsed.

8.1 Design of synapsed

Given that currently Archipelago is not network-aware, peers from one segment cannot know the ports of peers of another segment. Moreover, they cannot send a request to the synapsed peer and simultaneously target a peer in another segment. So, we are faced with the problem of defining the limitations of synapsed.

We propose the following design: Each of these two segments must have at least one synapsed peer. So, when one synapsed accepts an XSEG request, it will translate it to a network request, send it to the synapsed peer of the other segment, who in turn will finally translate it back to an XSEG request. Moreover, each synapsed peer must be attached to a peer of its segment, which will serve as the request target when synapsed accepts a request.

This means that synapsed does not actually connect two remote segments. More appropriately, it bridges two remote peers over network.

Another difficult task was to enable synapsed to listen from its port and its request queue simultaneously, using the request handling of Archipelago. We have tackled this problem in Section 8.2.2.

Moreover, synapsed has been designed to send requests using the standard TCP protocol. This means that it must handle the marshaling of buffers, send/receive errors as well as polling for new requests all by itself.

Finally, synapsed has been designed as a single-threaded peer.

8.2 Implementation of synapsed

In this section, we will explain how we implemented the main aspects of synapsed.

8.2.1 Synapsed initialization

Synapsed is a much simpler peer than cached and requires only the following arguments:

- The network port where it will listen for requests.
- The remote address of the segment. **NOTE:** it can point to the host address, which means that synapsed can also provide a generic way to bridge two peers who reside in the same host but in a different segment.
- The remote network port of the other synapsed peer.
- The XSEG port of the peer where synapsed will send the requests that it accepts.

Once synapsed has the above arguments, it can create a socket, bind it and connect to its port. After that, it can listen for new connections and forward the requests that it receives to the synapsed of the other segment. That synapsed can in turn forward the request to the peer that it has been configured to target to.

8.2.2 Request polling

Synapsed typically uses the common peer polling and Archipelago IPC methods (see Section 3.3.1) to check its XSEG ports for new requests. However, it simultaneously needs to listen for new requests on its socket. Yet, there is no way for a process to block on its socket **and** and synchronously wait for a queued signal.

The solution seems obvious; synapsed should instead block while polling for requests in its socket and, if a new request arrives at its XSEG port, its polling should be interrupted. This is a simple solution, but there is one detail we must take into account; sleeping like so is unsafe because peers currently block the SIGIO signal in order to wake up synchronously.

Let's elaborate on that a bit: When a peer is initialized, it blocks the SIGIO signal. Using sigtimedwait, it can check fast and without races if a signal has been enqueued, and then it can sleep. This is not the case with `poll()` though, since if the peer sleeps with the SIGIO signal blocked, it will not wake up. On the other hand, if we unblock the SIGIO signal, we have a new set of problems:

Consider the case where we receive a signal before we go to sleep. In order to be notified that a signal has been sent so as not to sleep, we would need to utilize a signal handler that would increment a global variable which would be checked right before we go to sleep. This approach is not only racy (what happens if SIGIO is received after this global variable is checked) but it is also very slow.

For this reason, we use the `ppoll()` alternative, which accepts a signal mask as an additional argument. `ppoll()` provides the guarantee that it will atomically: i) replace the old signal mask of the process with the one provided, ii) check if there are pending signals and if so return, iii) block until a request has been received or a signal has been queued.

Solving the issue of waiting for XSEG requests while blocking on a socket is only part of the problem. We also have to solve its counterpart, which means we need to find a quick way to iterate the XSEG ports of synapsed for new requests, while listening for new requests on its socket.

We have countered this problem by checking alternately the XSEG port of synapsed, with the existing methods, and the network port of synapsed, by doing a poll with zero timeout, which returns immediately.

8.2.3 Marshalling

The XSEG request cannot be sent as is, since its data and target name are stored in two different buffers. Moreover, since these buffers can have variable length, the synapsed peer that reads from its socket needs a way to know when it has finished reading all the necessary data.

This is a common problem in computer science and its solution is to serialize (or ”marshal”) the object that must be transfered. The marshaling method we have chosen is the following:

1. We send at the start of every transaction a fixed-length header, whose size is known to all synapsed peers, and which has the information about the length of the rest of the buffers. The header is presented on Listing 8.1

```
struct synapsed_header {
    struct original_request orig_req;   /* Address of original requests
   */
    uint32_t op;
    uint32_t state;
    uint32_t flags;
    uint32_t targetlen;
    uint64_t datalen;
    uint64_t offset;
    uint64_t serviced;
};
```

Listing 8.1: Synapsed header

   The header is simply the necessary fields of the original request that the remote synapsed peer needs to know. The most important fields are **datalen** and **targetlen** which indicate the length of the data and target buffers respectively.

2. Once the header has been sent and the remote peer has the information that it needs, it allocates a new XSEG request from its segment and fills it with the information provided by the header, thereby creating to buffers with the appropriate length.

3. Finally, the remote synapsed peer reads the rest two buffers and stores them in the XSEG request it has previously allocated.

Marshaling however is usually not so simple. There are three caveats that one must take into account before attempting to serialize manually an object. They derive from the often overlooked fact that the host and the remote machine are not architecturally the same, which can lead to:

1. **Different endianness**. This means that the two machines have completely different byte order. This is commonly solved by converting all of the data to network byte order (big endian) and then convert them to the native endianess of the machine.

2. **Different type representation**. Even if two architectures have the same endianness, it is possible that they represent the same types with different number of bytes. This is most common with 32-bit and 64-bit architectures which, for example, use 4 bytes and 8 bytes respectively to represent an int.

3. **Padding**. Due to data alignment issues, the compiler may need to pad the fields of a struct. The padding is depended on the processor that is used to compile the program. A common solution to sidestep padding issues from different processors is to ”pack” the structure, i.e. to enforce that the structure will have no padding.

For synapsed, the first two caveats do not affect us, since the machines that are used are both 64-bit, little endian machines. The third caveat may also not affect us, but just to make sure, we have packed our header structure using the gcc pragma directive:

```
1 #pragma pack(push, 1)
```

Listing 8.2: GCC pragma pack directive

8.2.4 Send/Receive

As we have mentioned in the previous chapter, in order to send the data from one synapsed peer to the other, we must marshal them first and then unmarshal them. This commonly requires to merge all buffers into one and send that buffer.

In synapsed however, we have chosen a different approach; we have employed the `readv()`/`writev()` functions, that allow us to do scatter/gather I/O.

This means that when we sent a request, we create an iovec vector that points to the data that we want to sent and their sizes (gather). Respectively, when we receive a request, we check its type and allocate the necessary XSEG buffers where the data can be copied (scatter).

8.3 Evaluation of synapsed

Currently, synapsed is in a functional but nascent state, meaning that important features such as mirroring of requests have not been implemented yet. Were these features implemented, they could also be evaluated and we would be able to quantify their performance cost.

This means that right now, synapsed’s main purpose is to enable cached and other Archipelago peers to work in a remote environment with more resources, without encumbering the host machine. This kind of flexibility cannot be evaluated of course, but it should be considered an important feat, since it opens numerous possibilities for a previously network-unaware Archipelago.

There is however an interesting way we can evaluate synapsed. More specifically, we can benchmark its performance for each of the Archipelago configurations that are tested in Section 7.5, minus the page cache test of course.

The setup for our benchmarks is identical to the setup that is used in Chapter 7. The only addition is that synapsed is in a different host, with similar specifications as our test-bed (see Section 7.2). Additionally, the two hosts are connected with a 1Gbit Ethernet cable. Note that neither this connection type nor synapsed are tuned for high performance. This is important in order to properly evaluate the following results.

We proceed with the performance results of our benchmarks. The bandwidth results are presented in Figure 8.1 and the latency results in Figure 8.2.

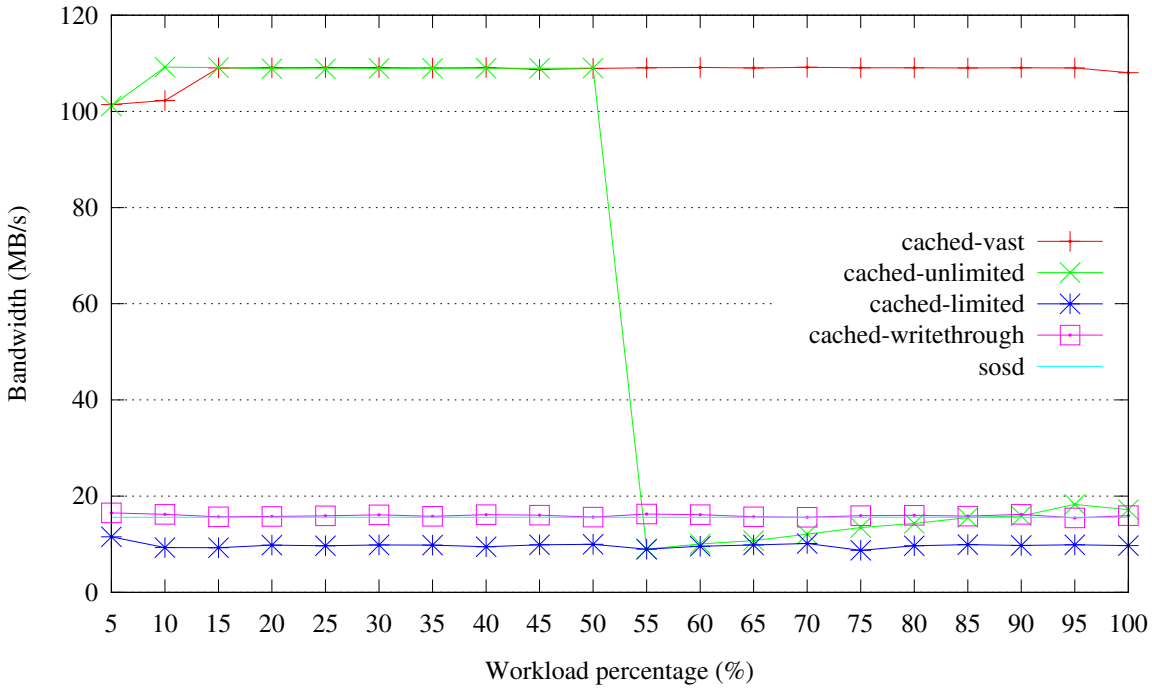

Figure 8.1: Bandwidth performance for writes through synapsed

For the first round of comparisons, will be invoke the results of Section 7.3.2 and more specifically Figures 7.5 and 7.6.

The first impression that one gathers by comparing these results is that they seem very similar but clipped. This is expected since the theoretical maximum bandwidth of a 1Gbit ethernet cable is 128MB/s. The fastest that we seem to reach is 110MB/s, which is very close, if we consider that on the same cable both peers send data simultaneously.

Most importantly however, besides the clipping in fast scenarios, there is practically little or no other effect on the rest of the scenarios, as they have approximately the same speed, with synapsed being only a little slower.

Moreover, we can see that cached-unlimited is performing similarly in both benchmarks and once again manages to outperform sosd, albeit being noticeably slower in the synapsed benchmark.

From the above comparisons we can observe that the network can be a bottleneck for an I/O intensive application, especially when it can achieve higher bandwidth than what the network can sustain.

We conclude this section with a comparison of our results with the results of Figures 7.10 and 7.11. As we can see in these results, the latency that is introduced by the VM operations, hypervisor and

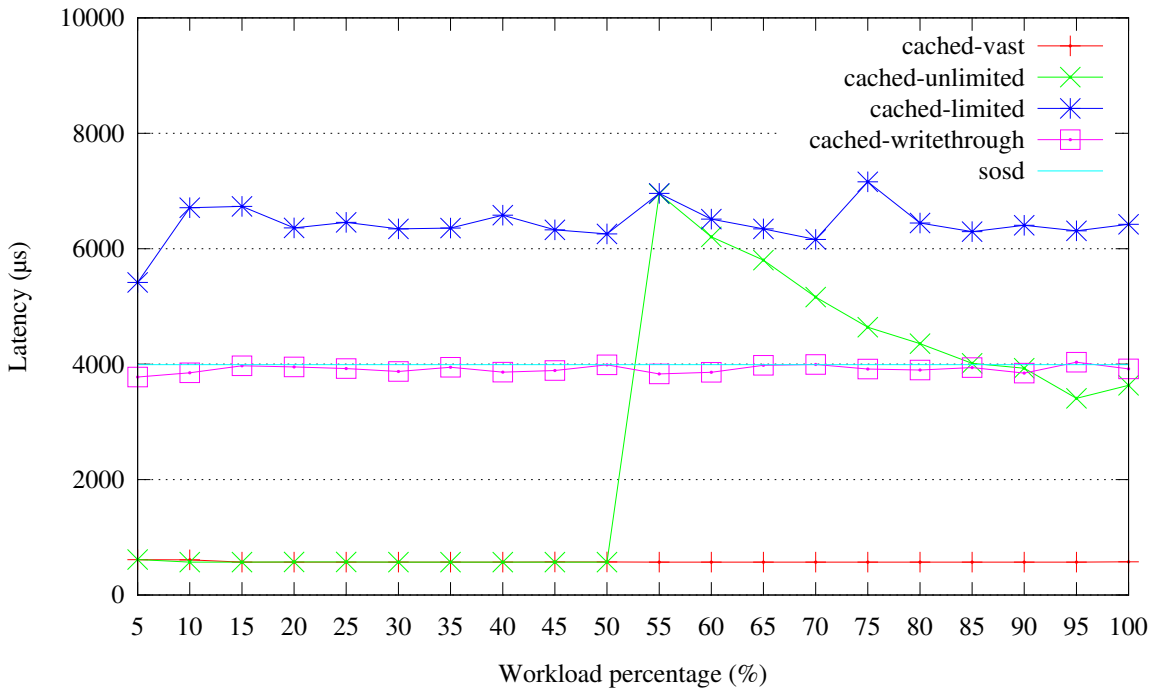

Figure 8.2: Latency performance for writes through synapsed

Archipelago is currently far greater than the < 1ms latency of the 1Gbit connection. Moreover, if the connection between the two servers were 10G, the latency that is imposed by the network would be negligible.

What we try to infer with the above observation is that in the current situation, the network latency of synapsed is a small price to pay if we can manage to run cached or even Archipelago in a remote environment with more RAM.

Chapter 9

Conclusion

9.1 Concluding remarks

This thesis is a record of over a year long process that has ultimately lead to the creation of two Archipelago peers, cached and synapsed. It documents the evaluation of third-party solutions, the reasoning behind our choice to create a custom cache mechanism, as well as the design decisions and technical issues that we have encountered.

Looking back at what we have created, we can safely state that this thesis has successfully covered two basic needs of Archipelago, caching and networking. The extend at which these needs have been satisfied can be considered as more than adequate, since cached has been successfully tested under an actual VM, while synapsed has managed to bridge two peers over network with minimum latency.

Most importantly however, cached provides a concrete solution to the task that was described in the very first chapter of this thesis; substantially improve the current performance of Archipelago. To be more specific, the results of synthetic benchmarks show that cached can improve up to 200x the current performance. Moreover, when tested with an actual VM, the performance speedup was able to reach 400%, even for small cache sizes.

Finally, the future looks very bright for our caching mechanism. The impending integration of cached in the demo environment[1] of Synnefo, will help it gain the necessary exposure that will provide us with the needed feedback for more improvements and the test-bed for new ideas.

9.2 Future work

The future work for cached is happening as of writing this very chapter. We are currently working to add the following:

- Full CoW support.
- Support for different namespaces (mappings, volumes, objects) so that cached can be used as a generic caching peer for all Archipelago needs.
- Support for different policies and limits per volume.

Moreover, the long term goal for cached is to be able to be used with synapsed effectively, in order to create a fast distributed and replicated cache.

[1] demo.synnefo.org

Bibliography

[1] Rfc 1122 by internet engineering task force - october 1989. http://tools.ietf.org/html/rfc1122.

[2] Thomas W. Barr, Alan L. Cox, and Scott Rixner. Translation caching: Skip, don't walk (the page table). SIGARCH Comput. Archit. Newsx, 38(3):48 – 59, 2010.

[3] L.A. Belady. A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2):78 – 101, 1966.

[4] Adrian M. Caulfield, Joel Coburn, Todor Mollov, Arup De, Ameen Akel, Jiahua He, Arun Jagatheesan, Rajesh K. Gupta, Allan Snavely, and Steven Swanson. Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.

[5] D. Dechev, P. Pirkelbauer, and B. Stroustrup. Understanding and effectively preventing the aba problem in descriptor-based lock-free designs. In Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2010 13th IEEE International Symposium on, pages 185–192, 2010.

[6] Imlementation of dictobject used by python. http://svn.python.org/view/python/trunk/Objects/dictobject.c?view=markup.

[7] Intel(r) xeon(r) cpu e5645 specifications. http://ark.intel.com/products/48768/Intel-Xeon-Processor-E5645-12M-Cache-2_40-GHz-5_86-GTs-Intel-QPI.

[8] 2011 thailand floods. http://en.wikipedia.org/wiki/2011_Thailand_floods.

[9] Gcc builtins for atomic memory access. http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/_005f_005fsync-Builtins.html.

[10] Cs 3110 lecture 22: Hash tables and amortized analysis. http://www.cs.cornell.edu/courses/cs3110/2008fa/lectures/lec22_amort.html.

[11] Would you pay $7,260 for a 3 tb drive? charting hdd and ssd prices over time. http://royal.pingdom.com/2011/12/19/would-you-pay-7260-for-a-3-tb-drive-charting-hdd-and-ssd-prices-over-time/.

[12] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. okeanos: Building a cloud, cluster by cluster. IEEE Internet Computing, 17(3):67–71, May 2013.

[13] Vangelis Koukis, Constantinos Venetsanopoulos, and Nectarios Koziris. Synnefo: A complete cloud stack over ganeti. ;login, 38(5):6–10, October 2013.

[14] Leslie Lamport. The part-time parliament. ACM Trans. Comput. Syst., 16(2):133–169, May 1998.

[15] Gordon E. Moore. Cramming more components onto integrated circuits. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, Readings in computer architecture, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[16] History of multithreading. http://people.cs.clemson.edu/~mark/multithreading.html.

[17] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling memcache at facebook. In Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation, nsdi'13, pages 385–398, Berkeley, CA, USA, 2013. USENIX Association.

[18] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in ramcloud. In Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, pages 29–41, New York, NY, USA, 2011. ACM.

[19] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Diego Ongaro, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for ramcloud. Commun. ACM, 54(7):121–130, July 2011.

[20] David S.H. Rosenthal, Daniel Rosenthal, Ethan L. Miller, Ian Adams, Mark W. Storer, and Erez Zadok. The economics of long-term digital storage. In The Memory of the World in the Digital Age: Digitization and Preservation, September 2012.

[21] Sugaya Seichi. Trends in enterprise hard disk drives. Fujitsu Sci Tech J, 42(1):61 – 71, 2006.

[22] The disk rotation speed bottleneck. http://www.dbms2.com/2010/01/31/the-disk-rotation-speed-bottleneck/.

[23] Ssd prices in steady, substantial decline. http://techreport.com/review/23149/ssd-prices-in-steady-substantial-decline.

[24] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: a scalable, high-performance distributed file system. In Proceedings of the 7th symposium on Operating systems design and implementation, OSDI '06, pages 307–320, Berkeley, CA, USA, 2006. USENIX Association.

[25] Sage A. Weil, Andrew W. Leung, Scott A. Brandt, and Carlos Maltzahn. Rados: a scalable, reliable storage service for petabyte-scale storage clusters. In Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing '07, PDSW '07, pages 35–44, New York, NY, USA, 2007. ACM.

[26] G. Wong. Ssd market overview. In Inside Solid State Drives (SSDs), volume 37 of Springer Series in Advanced Microelectronics, pages 1–17. Springer Netherlands, 2013.

[27] Φίλλιπος Γιαννάκος. Ανάπτυξη Οδηγού Στο Λ/σ linux Για Την Υποστήριξη Εικονικών Δίσκων Πάνω Από Κατανεμημένο Σύστημα Αποθήκευσης Αντικειμένων. Διπλωματική Εργασία, Εθνικό Μετσόβιο Πολυτεχνείο, 7 2012.