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

Thesis subject

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΟΝΟΜΑ ΦΟΙΤΗΤΗ

Επιβλέπων : Υπεύθυνος Διπλωματικής
Τίτλος Υπευθύνου

Αθήνα, Σεπτέμβριος 9999

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

Thesis subject

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΟΝΟΜΑ ΦΟΙΤΗΤΗ

Επιβλέπων : Υπεύθυνος Διπλωματικής
Τίτλος Υπευθύνου

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

Πρώτο μέλος επιτροπής Τίτλος μέλους	Δεύτερο μέλος επιτροπής Τίτλος μέλους	Τρίτο μέλος επιτροπής Τίτλος μέλους

Αθήνα, Σεπτέμβριος 9999

.....

Όνομα Φοιτητή

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

Copyright © Όνομα Φοιτητή, 9999.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

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

Περίληψη

Περίληψη της διπλωματικής.

Λέξεις κλειδιά

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

Abstract

Abstract of diploma thesis.

Key words

Key-word 1, Key-word 2, Key-word 3

Ευχαριστίες

Ευχαριστίες.

Όνομα Φοιτητή,

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

1.1 Introduction/Motivation

Bla-bla...

1.2 Thesis structure

Chapter 2: We define what ”cloud” means and mention some of the most notable examples. Then, we give a brief overview of the synnefo implementation, its key characteristics and why it can have a place in the current cloud world.

Chapter 3: We present the architecture of Archipelago and provide the necessary theoretical background (mmap, IPC) the reader needs to understand its basic concepts. Then, we thoroughly explain how Archipelago handles I/O requests. Finally, we mention what are the current storage mechanisms for Archipelago and evaluate their performance.

Chapter 4: We explain why tiering is important and what is the state of tiered storage at the moment (bcache, flashcache, memcached, ramcloud, couchbase). Then, we provide the related theoretical background for cached (hash-tables, LRUs). Finally, we defend why we chose to roll out our own implementation.

Chapter 5: We explain the design of cached, the building blocks that is consisted of (xcache, xworkq, xwaitq). Then, we give some examples that illustrate the operation under different scenarios

Chapter 6: We present the cached implementation, the structures that have been created and the functions that have been used.

Chapter 7: We explain how cached was evaluated and present benchmark results.

Chapter ??: It connects brain parts. And its tale must be told.

Chapter ??: We draw some concluding remarks and propose some future work.

Chapter 2

Chapter 2

2.1 Section 1

This section has an important citation[aeal99]

2.1.1 Subsection 1

This subsection has code in Haskell:

```
foo [] = []
foo h:t = 9: foo t
```

Listing 2.1: Sample code

It also has a list:

Item 1 First item

Item 2 Second item and a footnote[1].

Item 3 Third item and text in *italics*.

And an enumerated list:

1. First item.
2. Second item and text in **bold**

2.2 Section 2

2.2.1 Subsection 1

This subsection has a link to the block of code 2.1 in Section 1.

2.2.2 Subsection 2

This subsection has a FIXME comment, visible only to the author.

[1] Footnote description.

Chapter 3

Chapter 3

3.1 Necessary theoretical background

3.1.1 Multi-threaded programming

Multi-threading programming is good and is bad and here are some challenges:

1. Concurrency control
2. Challenge 2
3. Challenge 3

Concurrency control

Locking Three concepts for locking:

1. Lock overhead
2. Lock contention
3. Deadlocking

3.1.2 Typical IPC

Below we can see some IPC methods:

1. mmap()
2. Semaphores
3. Sockets

3.2 Archipelago

Archipelago consists of the following:

1. XSEG 2. 3.

3.3 XSEG

XSEG is the segment on which the IPC...

There are some XSEG stuff such as:

1. Drivers 2. Libraries 3. Xtypes 4. Peers

3.3.1 Drivers

3.3.2 Libraries

3.3.3 Xtypes

The rationale behind xtypes is:

- Abstraction(?) layers: Creating inner abstractions layers for software is not a new concept but it's very easy to miss, especially when you start small and end up big.

  In a nutshell, when writing code for a new software (in our case a peer for Archipelago but this can apply to most software that surpass the 1000 LOC[1] mark) it is wrong practice to create from scratch a monolithic implementation with indistinguishable parts. There is a main reason for this:

  Monolithic implementations usually derive from lack of code architecture and planning. Although it is feasible for a programmer to create fully-functional code that meets the necessary requirements, albeit with a lot more effort and concentration, this approach will backfire when the programmer needs to add new features. Since there is no explicit code architecture and the fragile inner correlations are between lines of code and not separate entities, stored precariously in the developer's mind, the result will eventually be constant code refactorization.

  One might think that new features happen once in a while in the development cycle but that would be wrong. This happens more often than you might think and is actually the common case in iteration and test-driven development.

  The right practice instead is to...

- Re-usability:...

- User-space / Kernel-space agnosticity: (I doubt that such a word even exists...)

3.3.4 Peers

Peers are Archipelago components that are responsible for accepting, processing and sending of the I/O requests. They are essential for the modular nature of Archipelago since each of them can be considered as a separate entity. They do their own logging, signal handling and processing.

The main Archipelago peers can be seen in Figure ?. As we can see from this figure, peers are processes that are attached to an XSEG segment. In the previous chaptr, we have mentioned that XSEG segments facilitate the IPC between different Archipelago components by offering a shared space where process can read and write to very fast. This however barely scratches the surface of IPC in Archipelago. In the following section, we will discuss more in-length the details behind Archipelago IPC

[1] Lines Of Code

3.3.5 Archipelago IPC

Fist of all, we must clarify that in Archipelago, IPC is done strictly between peers in the **same** memory segment. The reason is that we have crafted our own methods for IPC and the processes that need it must attain to a certain architecture, which is the peer architecture.

The entrance point for IPC is the peer port. When a peer is registered in the segment, it attaches itself to a port range. Peer ports are completely different to common ports (which are these ports?). When a peer wants to send a request to another peer, it must first ”get” the registered port on the segment. The xseg port is a structure that holds the necessary information as to where to send the request. Every port has three different queues; reply, request, free queue.

Request queues are typically a stack that can be addressed from different peers in the same segment. For this reason, they are designed as xtypes. For speed resons, they are pre-allocated to a certain length and re-allocated on-line, if there is need

Also, ports are designed to be considered as paths. That is, when a request is sent from one port to another...

3.4 Request flow example

We have bench xseg which works like so:

1. Get request
2. Prepare request
3. Create chunk
4. Allocate peer request
5. Set request (xhash)
6. Submit request

3.4.1 Get request

Explain here about xq or in xtypes?

Chapter 4

Tiering

4.1 Theoretical Background

4.1.1 Caching

In caching, there are usually the following two policies:

- Write-through: This policy bla bla bla
- Write-back: This policy blu blu blu

Eviction

Caching generally means that you project a large address space of a slow medium to the smaller address space of a faster medium. That means that not everything can be cached as there is no 1:1 mapping. So, when a cache reaches its maximum capacity, it must evict one of its entries

And the big question now arises: which entry?

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[Bela66]. However, the clairvoyance needed for this strategy was a little difficult to implement, so we had to resort to one of the following, well-known strategies:

- **Random:** Simply, a randomly chosen entry is evicted. This strategy, although it seems simplistic at first, is sometimes chosen due to the ease and speed of each. It is preferred in random workloads where getting fast free space for an entry is more important than the entry that will be evicted.
- **FIFO (First-In-First-Out):** The entry that was first inserted will also be the first to evict. 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 though, since it assumes that cache entries are used only once, which is not common in real-life situations.
- **LRU (Least-Recently-Used)**
- **LFU (Least-Frequently-Used)**

Choosing the LRU strategy is usually a no-brainer. Not only does it *seem* more optimal than the other algorithms, but it has also been proven, using a Bayesian statistic model, that no other algorithm that tracks the last K references to an entry can be more optimal.

4.2 Existing storage tiers

4.2.1 Bcache

4.2.2 Memcached

4.2.3 Blabla

4.2.4 Summary

Chapter 5

Design of cached

In the previous chapters, we have addressed the need for tiering in terms of scalability as well as performance.

We have also evaluated current caching solutions and described why they couldn't be used as a cache tier in Archipelago.

With the results of chapter ? in mind, we can provide some more strict requirements that our solution must have:

1. **Nativity:** Our solution must be native to Archipelago i.e. not need any translation layers to communicate with it.
2. **Pluggability:** Our solution 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 solution must cache requests in RAM, since the next fastest tier, SSDs, are already being used in RADOS as a journal.

For the following chapters, we will drop the *"solution"* moniker and we will use instead the proper name of our implementation, "cached", which simply means **cache d**aemon).

The following two chapters are the main bulk of this thesis and they present our own implementation 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. Section 5.3 presents the interaction between cached and its as well as the unique components that cached consists of. Finally, in Section ? we illustrate the flow of requests for cached.

5.1 Design rationale

One of the first architectural decisions was to implement cached as an Archipelago user-space peer (see Section 3.3.4 about Archipelago peers). 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 register on-line the cached peer between the vlmc and blocker and unregister 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 is possible because, as we have mentioned in Section 3.3.5, XSEG ports can be registered on-line. Thus, during normal operation, the administrator can add the cached port to the request path between vlmc and blocker, and all requests will seamlessly be intercepted by cached. This follows the same principle with bcache, which plugs its own request_fn() function to the virtual device it creates. Unlike bcache however, cached can be plugged on and off at any time.

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 vlmc and blocker, where the VM's requests have already been translated to object requests, the natural choice is to cache objects.

The decision to cache objects not only is the most natural one, but also is closer to the way our storage (RADOS) handles data. To understand the importance of it, consider the following:

Like bcache, our implementation 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 the VMs' volumes are 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, unlike bcache which expects that the backing device is also the physical device and coalesces date accordingly, our implementation is limited only in coalescing data in the object range (commonly 4MBs). If our implementation was caching in block or volume level, it would be unaware of that fact.

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

As for what we would index, it would be an overkill to further partition the objects and index regions within them. Moreover, this would make sense only if the objects where large (e.g. like volumes). So, our index mechanism should index object names solely. As for the index mechanism, we have chosen to use a very fast in-memory hash-table for this job. This covers the in-memory requirement we have set above. Also, this choice is one of the main reasons that our implementation is O(1).

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 blocks. So, in order to achieve a balance between safety and speed, we use a fine-grained locking scheme in critical sections that can be seen is discussed in detail in Section 5.2.4.

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.

[1] Quality of Service

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 (see Section **??** about xtypes).

The components of cached can be seen below:

- xcache, an xtype that provides indexing support, amongst many other things
- 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

Below, we can see a design graph of xcache:

FIXME: add Figure here

FIXME: add better design explanation
As we can see above, xcache utilizes two hash tables. One hash table is responsible for indexing entries (or more generally speaking ”cache entries”) that are active in cache. The other hash table is responsible for indexing evicted cache entries that have pending jobs. Again, more generally speaking, evicted cache entries are entries whose refcount has not dropped to zero yet.

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 pre-allocate the necessary space in advance and store them in a cache-node pool (note that this is different from the bucket pool).

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

Entry indexing

The index mechanism that xcache uses is a hash table named xhash, also an xtype. The reason why a hash table is used as an index mechanism is because:

1. Given that we index only a certain number of entries, we expect the that the insert, lookup and delete operations are in constant time (see below for an explanation why)
2. Hash tables can preallocate the space needed whereas tries/b-trees/bst will allocate nodes as new entries are inserted. Again, the fact that we index a certain number of entries means that we expect that we will have many evictions and insertions.
3. We don't need to do substring matches (advantage of tries)
4. We don't need to traverse the entries sequentially (advantage of B-trees and BSTs)

The hash table that is used is heavily based on dictobject[dict], the Python dictionary implementation in C. Distobject has been created to minimize the collisions and the hops (**FIXME:** Explain that better). Its only drawback is that it needs to resize when the table's entry history has reached the 2/3 of its capacity.

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) doesn't 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 wouldn't be the case if the user was responsible for the evictions.

As for the eviction strategy, we have utilized an LRU queue. Not only it's optimal (**FIXME:** verify it) for our purposes, but we have also mitigated the cost of keeping the last references for each entry by creating a simple LRU algorithm, which has O(1) complexity for all update actions. 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 chapter ?. 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. The updates at the LRU queue 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 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.

Re-insertion

We have previously mentioned that in xcache, there can be data migration between hash tables. Most commonly, an entry that is evicted from the active cache entries migrates to the hash table of the evicted cache entries, until its reference count falls to zero and can be freed.

However, what happens when xcache receives a request for an evicted entry which hasn't been freed yet?

In this case, the entry switches state again and is inserted back to the hash table of active entries. Also, its reference counter is incremented accordingly in order not to be freed amidst this process.

This way, we can avoid waiting for an entry that has just been evicted to flush its data.

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 xccahe 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 xcache flow

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

Insertion

FIXME: add figure and explanation

Lookup

FIXME: add figure and explanation

Put

FIXME: add figure and explanation

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 the following figure, we can see the design of xworkq:

FIXME: add figure

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 and will be able to execute the enqueued job. Else, it can safely leave and its job will be executed by the thread that has 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.

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 considered 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.

FIXME: add figure

Unlike xworkq, before a job is enqueued, 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 length the design of the cached components. Having the above sections in mind, we can proceed with presenting how cached has been designed.

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:

1. Request handling
2. Write policy enforcing
3. Cache coherence

Moreover, cached extends its repertoire using some unique components, namely:

1. Bucket pool
2. Cache-IO
3. Book-keeping

We will illustrate the design of cached from two different perspectives: the operational perspective, which can be seen in Figure 5.1, and the component perspective, which can be seen if Figure ?. Moreover, we will further explain how cached manages the above new components and tasks in the following sections.

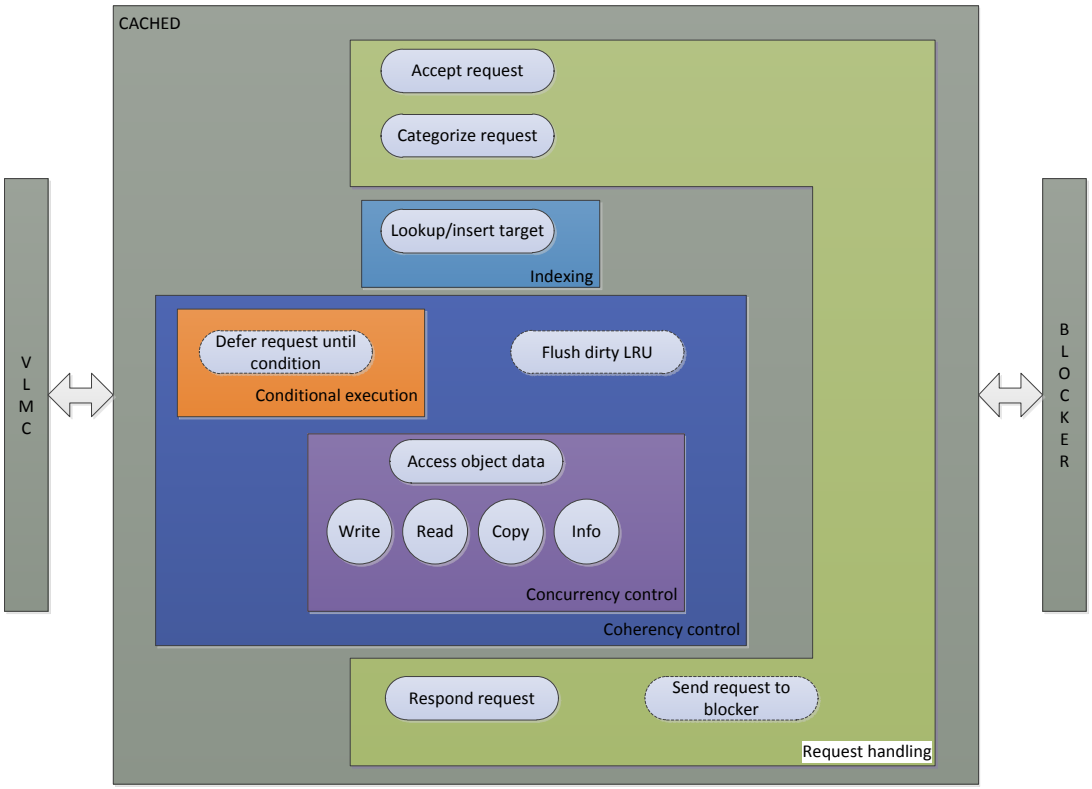

Figure 5.1: Cached design

FIXME: add component diagram for cached (xcache, xworkq, objects, buckets etc.)

5.3.1 Request handling

Cached operates as a peer that receives requests from the vlmc. 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 what will the write policy of cached be. There are two options: write-through and write-back. These policies aren't new and have already been discussed in chapter ?, 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 Cache coherence

Cache coherence is a concept that is closely connected with concurrency control, as is evident in Figure 5.1. It is commonly used as a term in the literature on the subject of caches for multiprocessor systems or distributed caches and it refers to:

1. the consistency of data that are stored in the cache and
2. how each change is propagated through the system, to the other storage tiers.

If we wanted to rephrase the above to match our case, we could say that it refers to the consistency of the data that are stored in cached and how each change is propagated to the blocker.

The consistency of the cached data is greatly secured using the xworkq as the guard of parallel accesses to the object's data. However, the guarantee that data updates are conducted serially does not suffice for the consistency requirement.

The problem is evident if you consider the eviction of an object with dirty data. xcache does not operate on object level and thus is unaware of the contents of the cache entry. Thus, cached must use the cache entry finalization hook to increment the refcount of the object so that it can not be removed

until all data have been flushed to the blocker. Essentially, cached overrides the xcache's book-keeping to maintain cache coherency.

The propagation of the data to the blocker is solely cached's job. One might consider that flushing the dirty data of an object would be sufficient to propagate the changes to the blocker. However, consider the scenario when an 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. The 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 following:

FIXME: add diagram

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 (xq). 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 Cache-IO

When a request is accepted, all Archipelago peers commonly embed it in a peer request (you can see more about peer requests here ?). The peer request always holds the original request and optionally, several other fields that are of importance to the peer.

In our case, we keep in cached's peer requests a new structure called Cache-IO, on which we store the xcache handler of the request handler as well as the state of the request. Moreover, since the request may break internally in many others, we keep track of how many pending requests remain until the original one can be completed.

5.3.6 Book-keeping

In order to know when an object is in flushing state, when a bucket has been allocated or when a bucket is being read, 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 correspond with the blockers data
   iii) loading, meaning that a read request has been sent to the blocker to be filled with data
   iv) dirty, meaning that it currently holds newer data that what the blocker has.

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) accpted, meaning that the request has just been accepted
   ii) reading, meaning that the request wants to read data
   iii) writing, meaning that the request wants to write data
   iv) served, meaning that the request has been served the data it needed
   v) failed, meaning that the request 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 Operation

5.4.1 Write

This is the flow for the write path:

FIXME: add diagram and explanation

5.4.2 Read

This is the flow for the read path:

FIXME: add diagram and explanation

5.4.3 Copy

This is the flow for the read path:

FIXME: add diagram and explanation

5.4.4 Info

This is the flow for the read path:

FIXME: add diagram and explanation

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 information of xcache, the main cached component. Next, section 6.2 presents the implementation of cached, showcasing the structures and functions 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: `xcache_init definition`

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;          /* Shadow entries */
    struct xq free_nodes;       /* Unclaimed (?) entries */
    xhash_t *entries;           /* Hash-table for valid entries */
    xhash_t *rm_entries;        /* Hash-table for evicted entries */
    struct xlock rm_lock;       /* Lock for rm_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 fields of the xcache structure for this purpose can be seen in Listing 6.3.

```
struct xcache {
    ...
    uint32_t size;              /* Upper limit of entries */
    ...
    struct xq free_nodes;       /* Unclaimed (?) entries */
    ...
    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 entries 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 cache 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 */
    xbinheap_handler h;                 /* Index in data segment */
    ...
    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 by our segment, typically 256 characters. During initialization, the entry name is cleared out of junk values.

h The entry's index.

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 the Section ?)

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

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

Listing 6.6: Cache entry allocation/initialization function

This function attempts to claim a cache entry from `free_nodes`. 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.

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

6.1.4 Cache entry indexing

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

```
struct xcache {
    struct xlock lock;          /* Main xcache lock */
    ...
    xhash_t *entries;           /* Hash-table for valid 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 6.7: xcache struct fields for entry indexing

As we have mentioned in Section 5.2.2, we utilize two hash tables, one for the cached entries and one for the evicted entries. These hash tables can be accessed from the `xcache struct` and are the `*entries` and `*rm_entries` 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[1]
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.

Moreover, we show in Listing 6.9 the cache entry struct fields 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 */
    ...
```

[1] #define NoEntry (xcache_handler)-1

```
    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

The commentary on the above fields follows:

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 an object is already ACTIVE for lookup operations (or else lookup will fail). For insertions and reinsertions, it is manually 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 code listing 6.10

```
struct xcache {
    ...
    struct xq free_nodes;        /* Unclaimed (?) entries */
    xhash_t *entries;            /* Hash-table for valid 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 **??**, 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.

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

6.1.6 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.

```
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

There are three main techniques xcache uses for concurrentcy control. The first one is the usage of locks, which is presented in Section 6.1.6. The second one is reference counting, which is presented in Section 6.1.6. Finally, the third one is a more esoteric method that counters the ABA problem and is the tracking of parallel puts to an entry, which is presented in Section **??**.

Locking

With xcache, we have tried not to use a BKL [2] 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] 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

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 by an external process and therefore not cache it.

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

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

Listing 6.13: Atomic operations of GCC

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

- When an entry is inserted in cache, the cache holds a reference for it (ref = 1).
- Whenever a new lookup for this cache entry succeeds, the reference is increased by 1 (ref++)
- When the request that has issued the lookup has finished with an which entry, the reference is decreased by 1. (ref–)
- When a cache entry is evicted by cache, the its ref is decreased by 1. (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

Entry put

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 void the usage of our two biggest lock: `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`.

FIXME: How much are we going to explain the ABA problem?

6.1.7 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. Information about the design of cached is provided in Section 5.3. Similarly to xcache, we will begin with the initialization process, we will continuee with the request handling of cached 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 command-line arguments can be seen in Table 6.1

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

```
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
   ) */
```

Switch	Info
-t	Number of threads
-mo	Max objects to cache
-ts	Total cache size
-os	Object size
-bs	Bucket size
-bp	Blocker port
-wcp	Write policy

Table 6.1: Command line arguments of cached

```
6     uint32_t object_size;           /* Max object size (bytes) */
7     uint32_t bucket_size;           /* Bucket size (bytes) */
8     uint32_t buckets_per_object;    /* Max buckets per object (
      object_size / bucket_size) */
9     xport bportno;                  /* Blocker port */
10    int write_policy;               /* Cache write policy */
11    struct xworkq workq;            /* xworkq for deferred jobs */
12    struct xwaitq pending_waitq;    /* xwaitq for when cache entry pool
      is empty */
13    struct xwaitq bucket_waitq;     /* xwaitq for when bucket pool is
      empty */
14    struct xwaitq req_waitq;        /* xwaitq for when we are out of
      requests */
15    unsigned char *bucket_data;     /* allocated space for buckets (
      bucket pool) */
16    struct xq bucket_indexes;       /* stack of bucket indexes (bucket
      pool) */
17    struct cached_stats stats;
18    //scheduler
19 };
```

Listing 6.15: Main cached struct

Moreover, on cached initialization we also initialize xcache as well as the general 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 (More on Section ?)

pending_waitq: A waitq for jobs that need to allocate a cache entry to continue

bucket_waitq: A waitq for jobs that need to allocate a bucket to continue

req_waitq: A waitq for jobs that need to allocate a request to continue

bucket_data: This is a malloced space whose size is the total_size of cache. This space is later split in buckets and its indexes are pushed on a stack

bucket_indexes: The stack where bucket indexes are pushed

Furthermore, during the xcache initialization that takes place inside the cached initialization, the cached node initialization hook is triggered and cached can create its own entries, which are as many as the max objects. The cache entry struct (or ”ce”) can be seen in Listing 6.16.

```
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

The explanation of the above fields follows:

status: The ce 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 buckets’ 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 ce 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 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 calloc 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 calloc two more arrays, the `bucket_alloc_status_counters` and the `bucket_data_status_counters`, 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, which is set to `buckets_per_object` (1024), and the data counter for INVALID buckets, which is similarly to the same number.

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

The way buckets are attached to the ce is analogous to the way a function maps to its address space a large memory chunk that has previously allocated; the chunk is internally divided to smaller chunks that are mapped to the function's address space only when the function "touches" them.

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 procedure is the following: we pop a bucket index from the `bucket_indexes` stack, translate it to the actual data pointer and store it to the `data` field of the struct bucket. Later on, when the bucket is released, we translate the data pointer back to the bucket index and we push it to the bucket pool.

6.2.3 Request handling

Cached uses a custom loop to poll for requests. This loop follows the same principles as the common peer's loop (see Section ?) with the addition of:

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 cio and we proceed according to its operation type. For received requests, the request's cio 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 on Section **??**.

6.2.4 "ENOSPC" scenarios

FIXME: add out of requests, out of buckets, out of cache entries

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’s actually is a valid one. In our case, we will try not to merely smear the next pages with diagrams but first explain the benchmarking methodology behind them.

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 7.2 presents the results of the benchmarks that we have done and provides in-depth explanations about each of them. Finally, Section ? is undefined.

7.1 Benchmark methodology

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

The first category is the comparison between using cached on top of the sosd (sosd has been discussed here ?) and using solely the sosd as the Archipelago storage. The category’s goal is to ”defend” one of the core thesis arguments, that tiering is a key element that will improve the performance of Archipelago.

In order to compare effectively 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 iii) they are out of the scope of this section as there can be many of these kinds of tests 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 the 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 stampede 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 ”stampede” 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. For example, we will measure things (blarg?) such as writethrough mode vs writeback mode, single-threaded vs multi-threaded etc.

Also, here is the following list is the options of cached that affect the measurements:

1. Bucket size
2. IOdepth
3. Cache size
4. Max cached objects
5. Write policy
6. ...

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. Thus 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 [e564] 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: dev100 specs

Also, mention that we evaluated both peers by sending requests directly at their request queues

7.2.1 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: We will measure the bandwidth for the bulk of our requests. This is a metric from the peer's perspective that reflects how much request size requests our peers can handle per second.

Latency: We will measure the average time the requests need to be served. Unlike bandwidth, this metric reflects the responsiveness of the implementation. For example...

IOPS: (Should we include them?)

Workload smaller than cache size - Peak behavior

Let's see now 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.

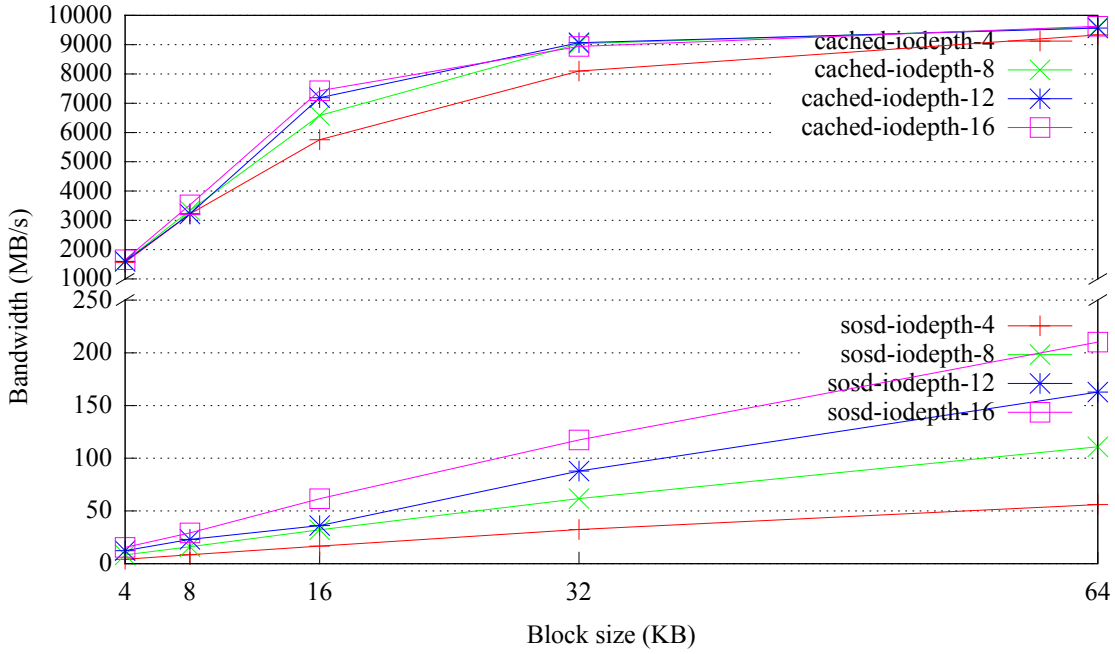

Figure 7.1: Comparison of bandwidth performance for writes

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.

We will begin the diagram interpretation with the bandwidth performance of the two peers. First, let's see the speedups of write and read requests with the addition of cached. 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.

From these first two diagrams we can extract the following points:

Why is there such a vast improvement?

(Explain that it's because we don't write past cache's size)

Where is the speedup difference between read and writes attributed to?

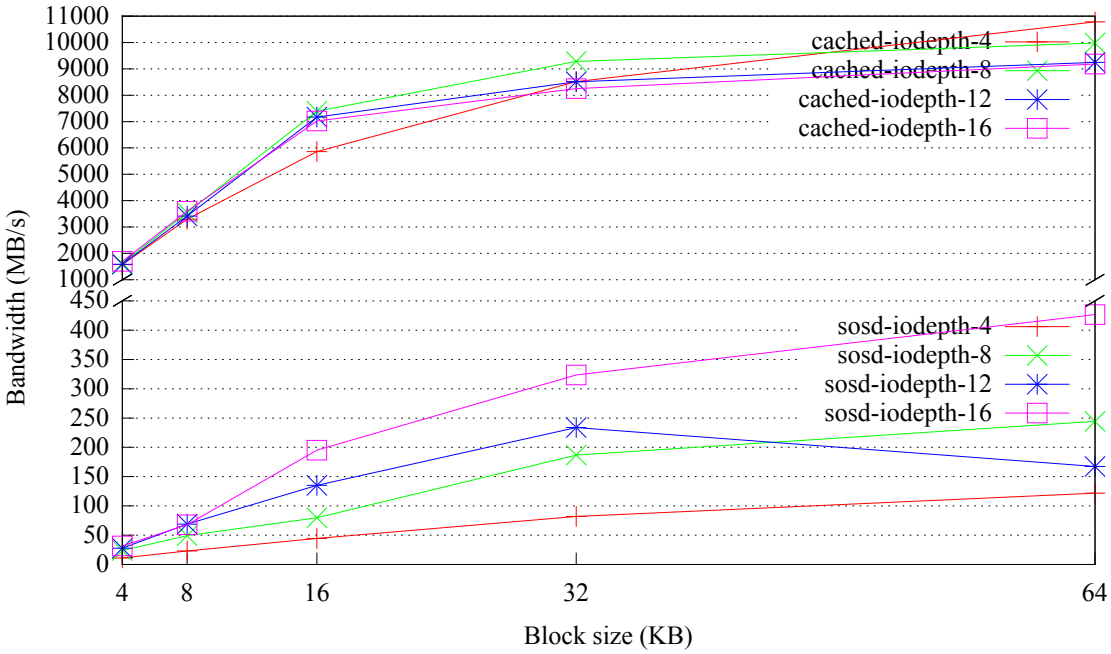

Figure 7.2: Comparison of bandwidth performance for reads

The speedup difference is attributed to three factors:

1. The performance of cached is almost the same for writes and reads. This is expected behavior as the read and write paths for cached have many common parts (see Figure ? and ?). However, if we go one step further we will see that under closer inspection, the reads seem consistently a bit faster than writes. So, how can these happen if both paths are the same?

   This is actually typical RAM behavior. Reads are reportedly faster than writes (find paper) due to the fact that the update of a bit of and SDRAM is slower than the read (arg, it's dumb).

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 hitting the bandwidth limit of the server's RAM. You can see in Table 7.1 that the bandwidth limit is 10.7GB/s. This limit is approached asymptotically as the block size increases and the CPU overhead decreases (more about the CPU overhead later on). Once more, the experienced eye may see that in reads we surpass this limit, which is logical given the fact that we have a multi channel RAM setup that reportedly increases marginally the RAM's performance (why reportedly and how marginally? Explain...)

3. On reads, sosd is benefited from the existence of caches in various levels: on OSD level, on RAID controller level and thus is faster.

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

Why is cached's performance increased along with block size?

This is another interesting observation but first, why don't we ask the same about sosd? This is because sosd's primary storage are hard disk drives, which have a major drawback; their seek time is not constant and is affected by the location of the contents in the disk platters. Cached however stores data

in RAM and we would expect that writing 16 x 4k blocks and 1 x 64k block to take approximately the same time.

From this observation, we can extract that the indexing-related stuff (job enqueuing, lock spinning, hash table indexing) dominate the cached's performance. We can make sure this is the case if the latency results are in microseconds instead of nanoseconds, which is typical for RAM.

Why isn't the performance of cached improved proportionally as the parallel requests increase?

The reason why we see a minor increase in the performance of cached, even though it's 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 locks. (explain better)

Let's proceed now 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.

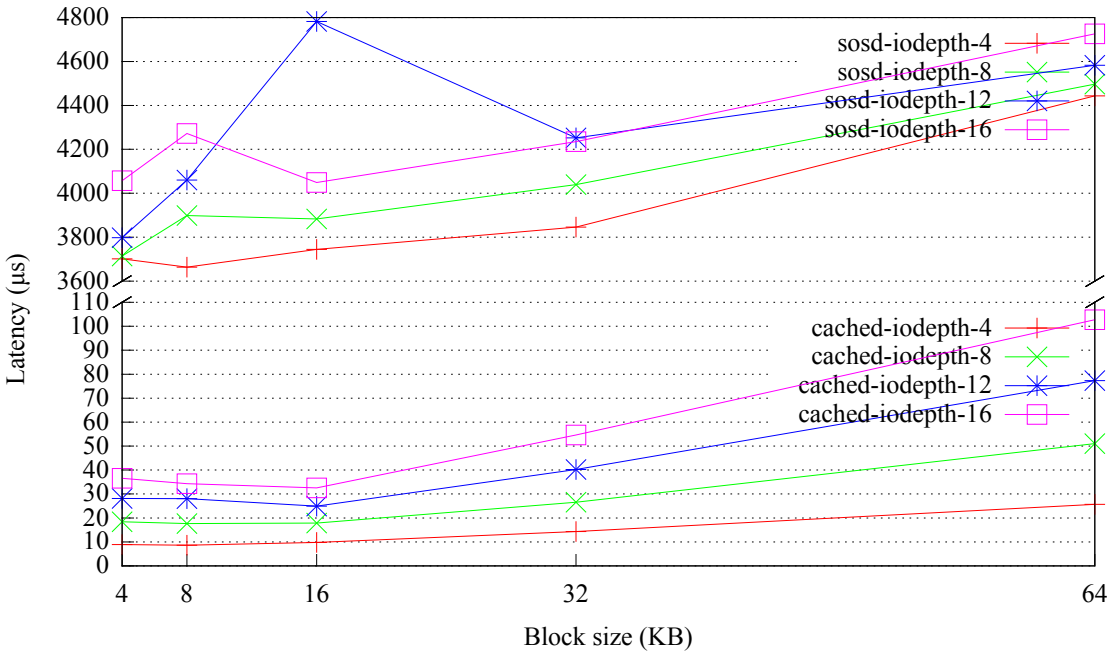

Figure 7.3: Comparison of latency performance for writes

We will measure the speedup of the write and read performance of cached. It is bla-bla for writes and bla-bla for reads.

We can also see that the latency results confirm our previous observations. The latency is increased proportionally with the iodepth, which indicates once more that we need a more fine-grained locking scheme. Also, latency results are in the order of microseconds instead of nanoseconds which further supports the assertion that the results are dominated by index-related stuff (argh, think of something prettier)

7.2.2 Workload larger than cache size - Consistent 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

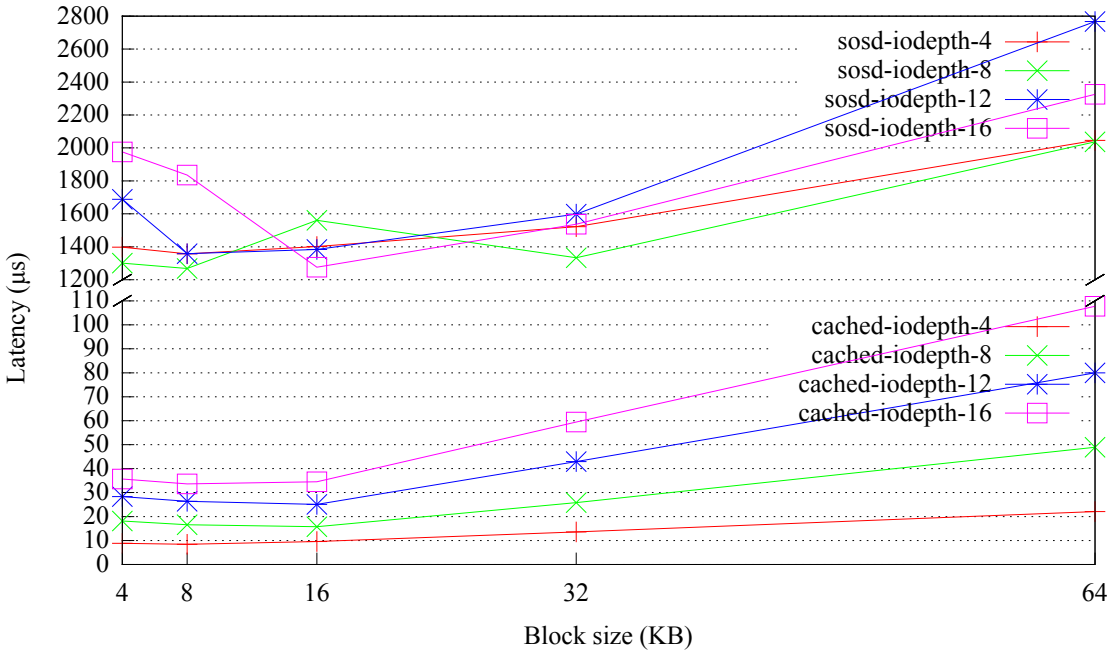

Figure 7.4: Comparison of latency performance for reads

the first part though, where the cache size was the same as the workload's, on this part the cache size will be only a fraction of it. This is after all the projected usage of cached in production environments.

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. We have chosen to measures cache sizes that start from the 1/64th of the workload's size and reach up to the 1/8th of the workload's size. The maximum objects are chosed differently (oversubscriptions on cache size etc.) they start from a 1/1 and reach up to 8/1.

In Figure 7.5 and Figure 7.6 we can see how cached performs for the above scenario. sosd's result are of-course un-affected from the cache size and maximum cache objects, and that's will be used as indicative lines for slowness, fastness, does this word even exist?

Comparing these results with the results Figure 7.1 and 7.2, there is a vast drop in the performance. Writes specifically cannot outperform the sosd. On the other hand, reads are generally faster than sosd, about 1.5x.

Why adding more objects makes us slower?

Why reads manage to remain faster?

We will accompany the above diagrams with latency results. You can see them below:

7.3 Performance evaluation of cached parameters

On this part, we will see what impact do different cached parameters have on its performance. We will test the following:

1. Impact of different number of threads

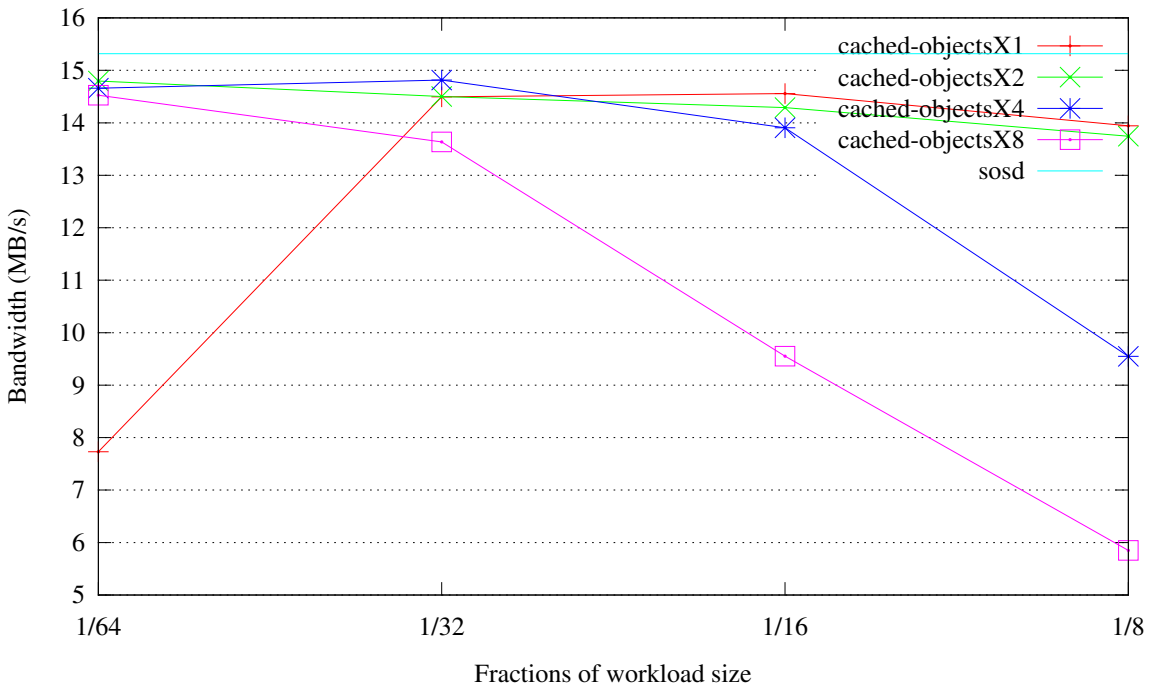

Figure 7.5: Comparison of bandwidth performance for writes

2. Impact of cold cache vs. hot cache
3. Impact of writeback vs. writethrough mode

Note that the tests above are run with the following parameters:

Mode Writeback

Block size 4k

Cache size Always larger than benchmark size

The above options have been chosen to isolate cached of any other factors that may alter it's performance. This way, we will be able to see more clearly the that in any other

Threads

Checking the performance impact of multi-tasking is meaningless without issuing parallel requests. Therefore, for each number of threads, we will use different IOdepth and measure its performance.

The bandwidth results can be seen in Figure 7.9 whereas the latency results can be seen in Figure 7.10.

From these results, we derive the following conclusions:

1. Our implementation is benefited from multi-threading. We achieve a major performance improvement of up to 75% when using two threads, as well as lower performance improvement for up to four threads, as the number of parallel requests increases.
2. We don't scale well past the two threads and four parallel requests.

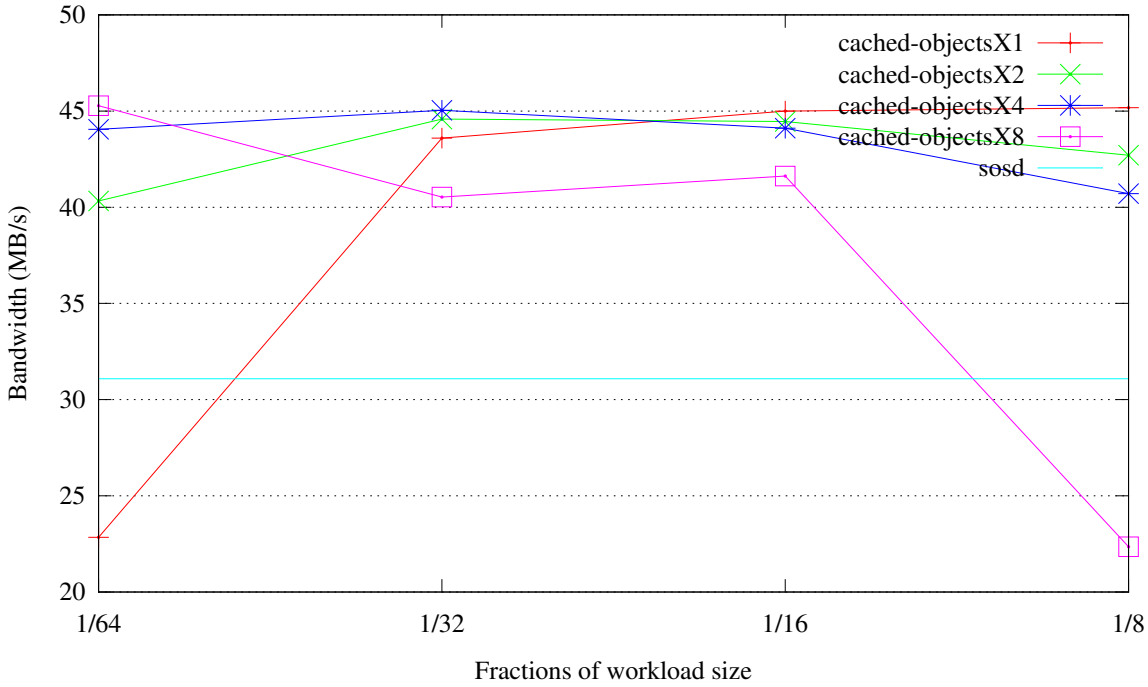

Figure 7.6: Comparison of bandwidth performance for reads

3. Adding more than two threads degenerates significantly the performance when the number of parallel requests is small.

Finally, these results, along with the results of the first part, clearly show the dark spot of our implementation; it needs a more fine-grained locking scheme else most of the thread's time will be spent spinning for a lock.

Cold cache vs Hot cache

This scenario will attempt to evaluate the overhead of cache misses in cached against cache hits 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.

For this reason, we have written 128K (where K is 1024 and M is 10242), 256K, 512K and 1M objects and have measured their latency performance. We expect that the experimental results will verify the claim that our implementation is O(1).

To get the most accurate results and since we want to test just the performance of our indexing mechanism, we have also used only 1 thread and only 1 IOdepth.

On Figure 7.11 we can see the results we were talking about. The major point in these results is that we can see that write latency, either of cold or warm cache, remains practically the same as the number of objects increases.

As a side note, we observe a constant decrease in latency as the number of objects increase this is not something that should be attributed to our implementation. (explain that we have used a hash table that holds 2million objects, so it is not mapped to our process's address space. When more objects are indexed, the hash table becomes fuller and the latency of mmap()s is equally distributed to the objects. Else, the hash table is more scarce but the same blocks are hit, albeit not fully written, and thus the mmap latency is the same but distributed to less objects.)

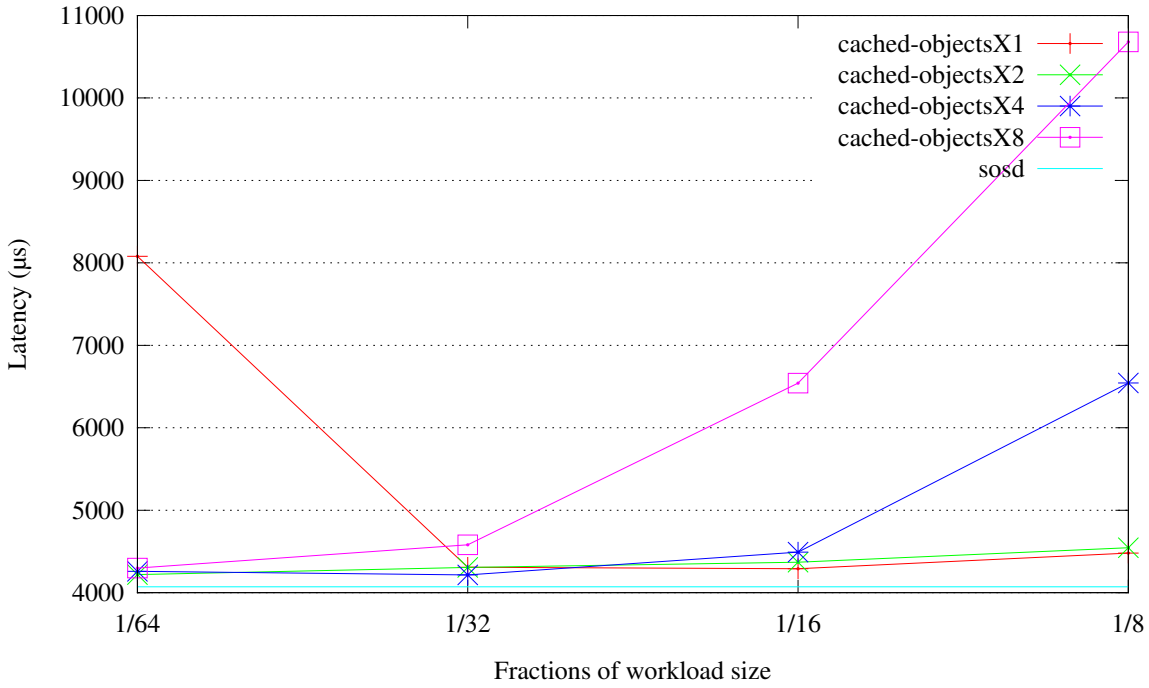

Figure 7.7: Comparison of latency performance for writes

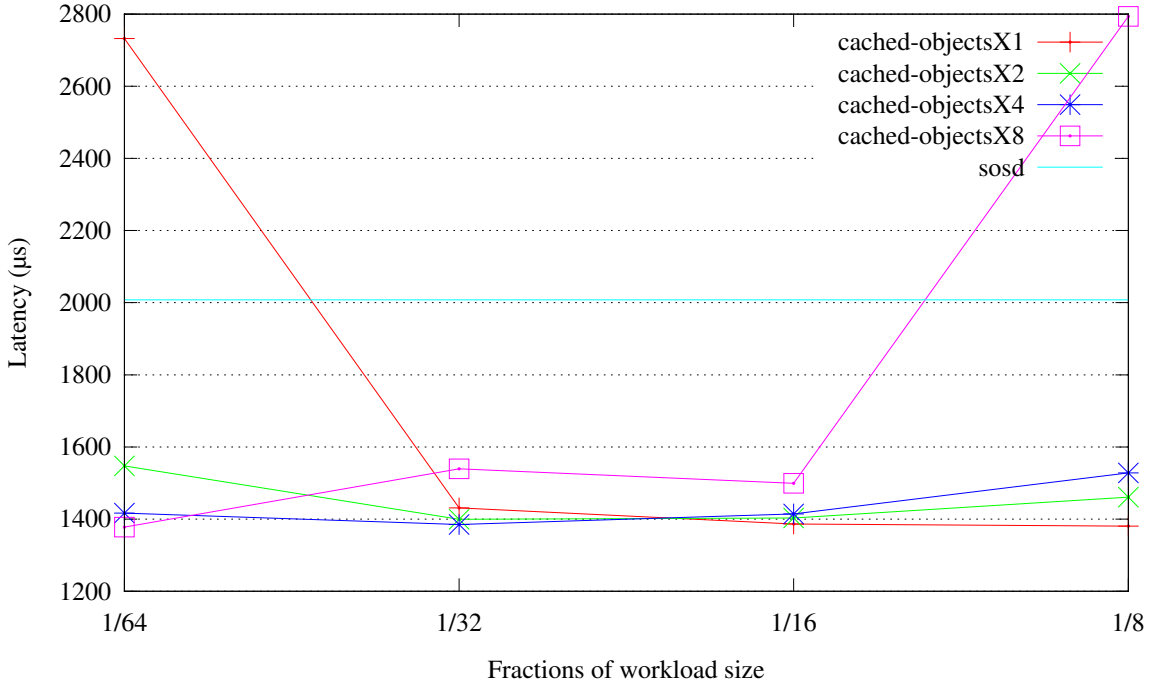

Figure 7.8: Comparison of latency performance for reads

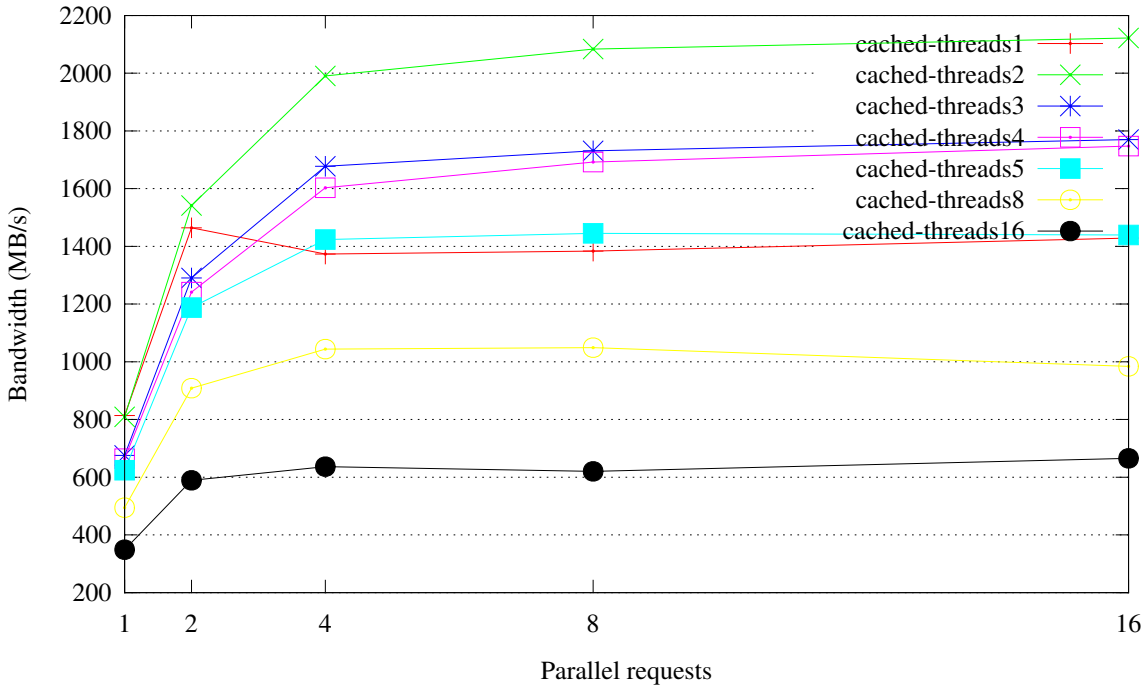

Figure 7.9: Bandwidth performance per number of threads

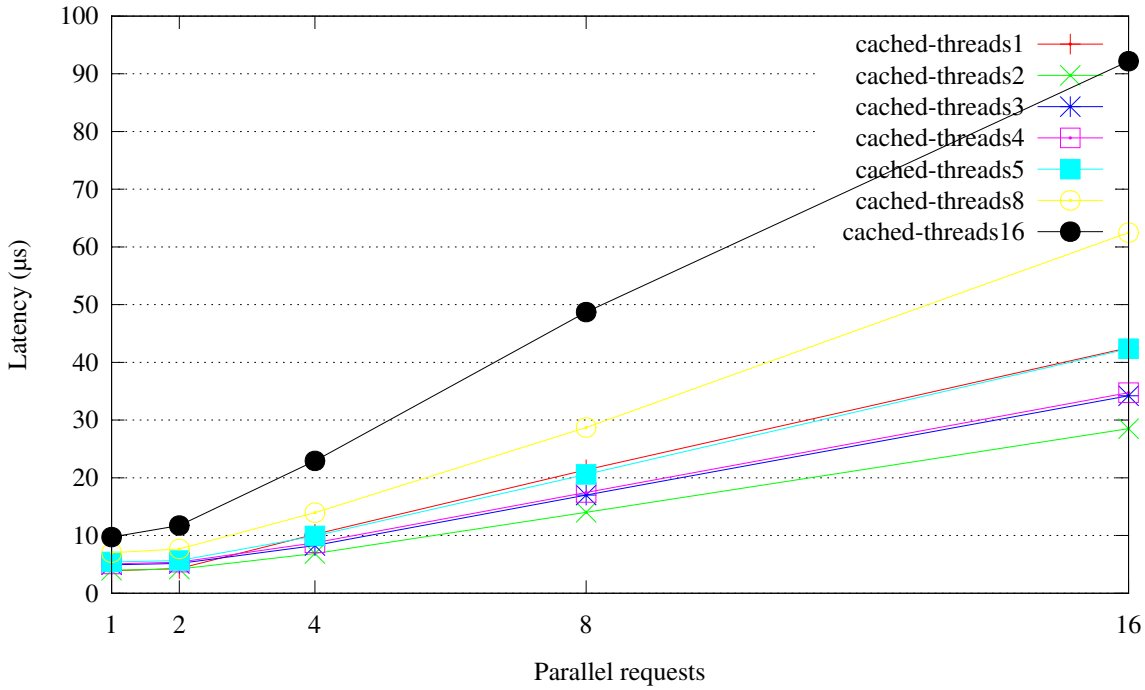

Figure 7.10: Latency performance per number of threads

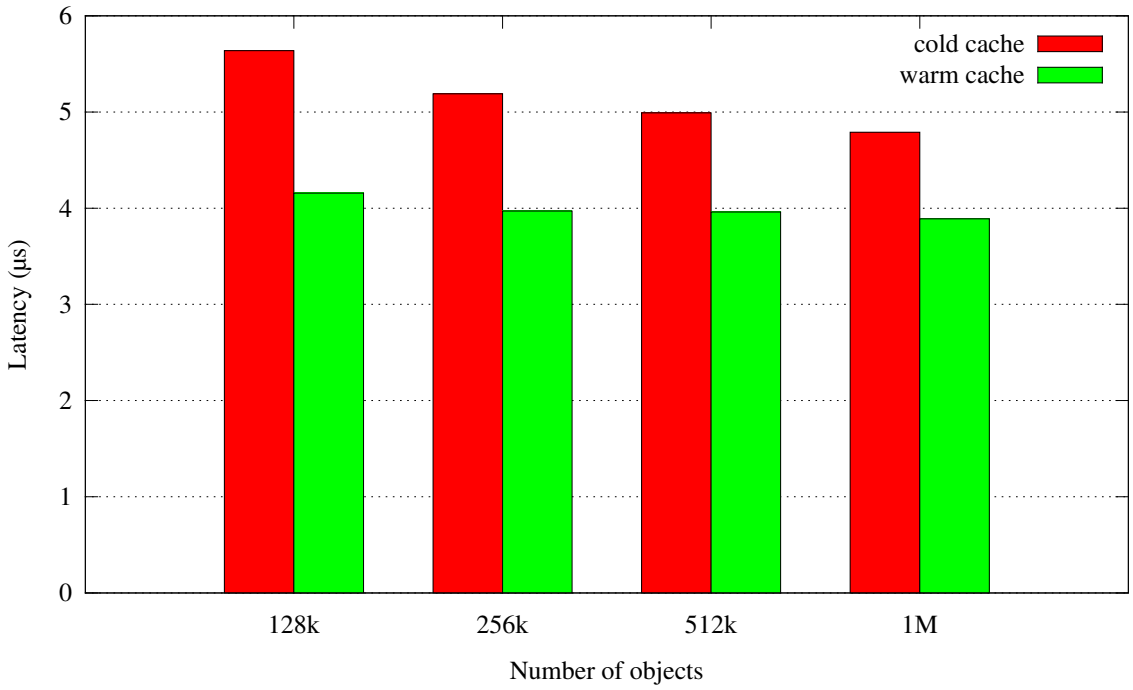

Figure 7.11: Latency performance of cold/warm cache for variable sizes

Bibliography

[aeal99] Some author et al., "Name of citation", in Proceedings of the 99th ACM Symposium on Something (POPL'99), pp. 999–999, Nine, 9999.

[Bela66] L.A. Belady, "A study of replacement algorithms for a virtual-storage computer", IBM Systems Journal, vol. 5, no. 2, pp. 78 – 101, 1966.

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

[e564] "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.