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.

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

Contents

List of Figures

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 provide extensive benchmark results and compare them to the ones of Chapter 3.

Chapter ??: TODO

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:

```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 Archipelago

Archipelago consists of the following:

1. XSEG 2. 3.

3.2 XSEG

XSEG is the segment on which the IPC...

There are some XSEG stuff such as:

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

3.2.1 Drivers

3.2.2 Libraries

3.2.3 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

[1] Lines Of Code

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...)

3.2.4 Peers

Chapter 4

Tiering

4.1 Theoretical Background

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. Requirement 1
2. Requirement 2
3. Requirement 3
4. Requirement 4

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 ? provides a general overview of cached. Sections ? - ? present the building blocks of cached and their design. Section ? presents the interaction of cached and its building blocks. Finally, in Section ? we illustrate the flow of requests for cached.

5.1 Design overview

Cached is a peer that can operate in two modes: writethrough and writeback.

In writeback mode, the data are written directly to cache... In writethrough mode, the data are written to the backing storage and then to cache...

Since Archipelago divides internally the volumes to objects (usually 4MB), the cached peer operates on object level. Also, in order to know which parts of the cached object are actually written, or are in the process of being read etc. cached further divides objects to the next logical entity, buckets (typically 4KB). Each bucket consists of each data and metadata and cannot be half-empty, or half-allocated. Thus, a bucket can be considered as the quantum of cached objects.

The cached peer consists of a number of building blocks. Per Archipelago policy, these building blocks have been written in the xtypes fashion. The reasons behind this decision have been discussed in chapter ? but for completeness’ sake, we will mention once more the merits of xtypes:

The main parts that cached consists of are the following:

- xcache, an xtype that mainly provides indexing support
- xworkq, an xtype that guarantees atomicity for execution of jobs on the same target
- xwaitq, an xtype that allows condition execution of jobs
- bucket pool, a pre-allocated memory pool for buckets and their metadata

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

5.2 The xcache xtype

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

- object indexing,
- eviction handling and
- coherency

Below we can see a design overview of xcache:

More specifically, xcache utilizes two hash tables. One hash table is responsible for indexing objects (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.

5.2.1 Indexing

In order to index the cached objects, xcache relies on another xtype, xhash, which is a hash table. What’s more, it’s actually the C implementation of the dictionary used in Python.

We have chosen to use a hash table as our index because:

Finally, the xhash xtype gives provides us with the basic hash table functions, namely:

- Insertion
- Look-up
- Deletion

5.2.2 Eviction

When xcache reaches its maximum capacity and is requested to index a new entry, we have to resort to the eviction of an older cached entry. But, which cache entry must we evict? This is a well documented problem that 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 object is more important than the object 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 object can be more optimal.

Also, an added bonus is that we won't need to sacrifice speed over optimality, since that, our hash table approach allows us to create an O(1) LRU algorithm which you can see in the following figure:

In a nutshell, our LRU implementation uses a doubly linked list blablabla. This design allows us to do all of the following action in constant time:

- Insert a new entry to the LRU list
- Evict the LRU entry
- Update an entry's access time (i.e. mark it as MRU)
- Remove an arbitrary entry

Another interesting feature of xcache is that evictions occur implicitly and not explicitly. The user doesn't need to interact with the LRU queue.

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. Also, the user will be notified via specific event hook that is triggered upon eviction that an entry has been evicted.

More about hooks can be seen in the following subsection.

5.2.3 Concurrency control

The concept of thread-safety has been discussed in chapter ?. The goal of xcache is to provide the necessary guarantees so that the code can be accessed from different threads at the same time.

In order to do so, we must first identify which are the critical sections of xcache, that is the sections where a thread can modify a shared structure. These sections are the following

- All xhash operations: Two of the three xhash operations (inserts and removals) can modify the hash table (e.g. they can resize it and reallocate space for it). This means that the third one (lookups) must not run concurrently with the other.
- Cache node claiming: Before an entry is inserted, it must acquire one of the pre-allocated nodes and we must ensure that this can happen 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. This a scenario that must be handled properly.
- Reference counting: Every entry must have a reference counter. Reference counters provide a simple way to determine when an entry can be safely removed. You can see more about reference counting in chapter ?
- LRU updates: Most actions that involve cache entries must subsequently update the LRU queue. Being a doubly linked list, if two threads update the LRU simultaneously, we can lead to segfaults.

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

xhash operations: We provide a lock for each hash table Cache node claiming: The free node queue is protected by a fast lock Entry migration: We always take fist the lock for entries and then for rm_entries LRU updates: Since all LRU operations take place for entries in ”entries” hash table and LRU updates are blazing fast we can secure our LRU with the cache->lock.

Another important guarantee is the reference counting of objects. xcache uses atomic gets and puts to update the reference count of an object.

5.2.4 Re-insertion

In xcache, there is a concept called ”re-insertion”. In order for an entry to be re-inserted to the primary hash table (which will be called ”entries” from now on) it must first reside in the hash table that indexes the evicted cache entries (which will be called ”rm_entries” from now on). As mentioned above, an entry that is in rm_entries has probably pending jobs that delay its removal.

So, what happens if a lookup arrives for that entry while on this stage? In this case, we re-insert it to entries and increase its refcount by 2, since there is one reference by the hash table and one reference by the one who requested the lookup.

5.2.5 xcache flow

Below we will see three important scenarios

Insertion

Figure

Lookup

Figure

Put

Figure

5.3 The xworkq xtype

The xworkq xtype is a useful (what?) for concurrency control on object level. It is important to distinguish between cache level operations and object level operations. Cache level operations include insertions, lookups, removals, allocations and refcount handling. On object level, there is a different set of operations that must be synchronized across threads. Namely, we have bucket claiming, read/write operations and object flushes.

The above distinction makes it easy to see that provided that operations on object level need not worry about interactions with other objects. Each object is ”sandboxed”, so to speak.

Let’s see the design of the xworkq xtype. It consist of a queue where jobs (e.g. read from block, write to block) are enqueued. The thread that enqueues a job can attempt to execute it to, by acquiring a lock for the workq. If the lock is free, the thread will be able to execute the enqueued job. Also, other threads can enqueue their jobs, so the thread that has the lock can do those too. There is an xworkq for every object.

Every object has a workq. Whenever a new request is accepted/received for an object, it is enqueued in the workq and we are sure that only one thread at a time can have access to the objects data and metadata.

For more information, see the xworkq.

5.4 The xwaitq xtype

When a thread tries to insert an object in cache but fails, due to the fact that cache is full, the request is enqueued in the xcache waitq, which is signaled every time an object is freed.

For more information, see the xwaitq.

5.5 Cached internals

5.5.1 Object states

Every object has a state, which is set atomically by threads. The state list is the following:

- READY: the object is ready to be used
- FLUSHING: the object is flushing its dirty buckets
- DELETING: there is a delete request that has been sent to the blocker for this object
- INVALIDATED: the object has been deleted
- FAILED: something went very wrong with this object

Also, object buckets have their own states too:

- INVALID: the same as empty
- LOADING: there is a pending read to blocker for this bucket
- VALID: the bucket is clean and can be read
- DIRTY: the bucket can be read but its contents have not been written to the underlying storage
- WRITING: there is a pending write to blocker for this bucket

Finally, for every object there are bucket state counters, which are increased/ decreased when a bucket state is changed. These counters give us an O(1) glimpse to the bucket states of an object.

5.5.2 Per-object peer requests

Reads and writes to objects are practically read/write request from other peers, for which a peer request has been allocated. There are cases though when an object has to allocate its own peer request e.g. due to a flushing of its dirty buckets. Since this must be fast, there are pre-allocated requests hard-coded in the struct of each object which can be used in such cases.

5.5.3 Write policy

The user must define beforehand what is the write policy of cache. There are two options: writethrough and writeback. On a side note, as far as reads and cache misses are concerned, cached operates under a write-allocate policy.

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.