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 : Υπέυθυνος 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

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 ??: 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.
3. **Object awareness:** Our solution must be able to operate on object level, which are the data entities Archipelago understands.
4. **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 ? 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

First of all, cached has been designed as an Archipelago user-space peer (see Section 3.3.4 about Archipleago peers). This design decision covers the nativity requirement we posed at the beginning of this chapter.

Requirements:
1. ~~Nativity~~
2. Pluggability
3. Object Awar
4. In-memory

Also, cached's purpose is to provide a caching layer between the vlmc and blocker. Due to the fact that cached is a peer, this is easily achievable even under normal operation because:

1. As we have mentioned in Section 3.3.5, XSEG ports can be registered on-line.
2. 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.

Thus, the pluggability requirement has also been covered.

Furthermore, one of the most important aspects of cached's design is the index mechanism. Specifically, we have utilized an in-memory hash table to index our cached objects.

Moreover, cached has been designed to be object aware. This means that we search in the hash table with the object name as key.

There is however a problem when operating solely on object level. Objects have typically 4MB of size. What would happen if a user requested e.g. a 16KB chunk of an object?

In this case, 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[1].

The solution to this 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.

The buckets are pre-allocated, which means two things:

1. We don't need to care about memory fragmentation and system call overhead
2. We cannot index single buckets. <FILLME>

Finally, an important aspect of cached is its multi-threading support. Specifically, cached can work with multiple threads that can 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 5.3.

5.1.1 Cached components

Let's see now the design of cached in detail. The cached peer consists of a number of building blocks. Per Archipelago policy, most of these building blocks 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

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

- xwaitq, an xtype that allows conditional execution of jobs
- bucket pool, a pre-allocated memory pool for buckets

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:

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

Below we can see a design overview of xcache:

As we can see above, xcache utilizes two hash tables. One hash table is responsible for indexing entriess (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 Entry Preallocation

Since xcache has a bounded number of entries that will allocate, 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, the best thing to do in our case would be to pre-allocate the necessary space.

5.2.2 Entry indexing

In order to index the cached entries, xcache relies on another xtype, xhash, which is a hash table. Moreover, 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

5.2.3 Entry eviction

As we can see in figure ?, xcache has been designed to index a pre-defined number of entries. That means that when xcache reaches its maximum capacity and is requested to index a new entry, it has to resort to the eviction of a previously cached entry. We have chosen the LRU strategy

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 and utilize the hash table to jump to the element (instead of traversing the list linearly). 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.

5.2.4 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 shared memory.

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, add more entries or delete existing ones). 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
- Reference counting: Another important guarantee is the reference counting of entrys. xcache uses atomic gets and puts to update the reference count of an entry.
- 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.

5.2.5 Re-insertion

We have previously mentioned that in xcache, there can be data migration between hash tables. This is easy to see why in case of evictions: an entry that previously was in "entries" must now be migrated to "rm_entries" until its reference count falls to zero and can be freed.

However, what happens when xcache receives a request for an evicted entry?

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.

5.2.6 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: write-through and write-back. These policies aren't new and have been discussed extensively 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.

5.6 Cached Operation

5.6.1 Write-through mode

Here we will see how cached operates in write-through mode.

Write

This is the flow for the write path:

Read

This is the flow for the read path:

5.6.2 Write-back mode

Here we will see how cached operates in write-back mode.

Write

This is the flow for the write path:

Read

This is the flow for the read path:

Chapter 6

Implementation of cached

In the previous chapter, we presented a design overview for cached and its components. In this chapter we will blabla how the above design has been implemented and explain in depth the structures and functions that have been created for this purpose.

More specifically, sections ? - ? provide implementation information for the components of cached, as described in Chapter ?. Next, section ? presents the actual initialization and blabla operations using excerpts from the code.

6.1 Implementation of xcache

In this section, we describe how we implemented the design concept of section 5.2. The main `xcache` structure is the following:

```
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 6.1: Main `xcache` struct

Each of the above `xcache` struct fields serves a design purpose. Let's see which fields help in what:

6.1.1 Entry Preallocation

The relevant fields for this purpose can be seen in the following code listing:

```
struct xcache {
    ...
    uint32_t size;              /* Upper limit of entries */
    uint32_t nr_nodes;          /* Shadow entries */
```

```c
    struct xq free_nodes;        /* Unclaimed (?) entries */
    ...
    struct xcache_entry *nodes; /* Data segment */
    ...
};
```

Listing 6.2: `xcache` struct fields for preallocated entries

and the definition of the `xcache entry` struct which shows up in `xcache` struct can be seen below:

```c
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 6.3: `xcache entry` struct

Let's start by listing what `xcache entry` consists of. First of all, it must have a name. Since we preallocate the entries and cannot know in runtime their length, we must allocate as much space as possible. The `char name[XSEG_MAX_TARGETLEN + 1]` field, which is 256 characters long, is long enough to hold the target's name. Also, as we have mentioned in Section 5.2.1, xcache must be agnostic of the cache contents. To this end, we use the generic `void *priv` field as a pointer to the actual entry content. The rest of the fields will be explained in the following chapters.

Let's continue now with the fields of Listing 6.2. Since we preallocate the entries using `malloc`, they take up a contiguous space in memory. The start of this space is the where the `*nodes` field points to. The `free_nodes` field works similarly to the free_entries field in Section 3.4.1 i.e. it is a stack where indexes to unused nodes are pushed. These indexes will be seen in various code excerpts in this chapter and have a specific name, `xcache_handler`[1].

6.1.2 Entry Indexing

The relevant fields for this purpose can be seen in the following code listing:

```c
struct xcache {
    ...
    xhash_t *entries;           /* Hash-table for valid entries */
    xhash_t *rm_entries;        /* Hash-table for evicted entries */
    ...
};
```

Listing 6.4: 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 former cached entries (or evicted entries or removed entries). These hash tables can be accessed from the `xcache struct` and are `*entries` and `*rm_entries` respectively.

[1] #define xcache_handler uint64_t

These are the functions which are related to indexing and `xcache` exposes to the peer function:

```
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);
int xcache_invalidate(struct xcache *cache, char *name);
```

Listing 6.5: Indexing functions

All of these function 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[2]
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: α□) the same handler or β□) another one, if this entry already exists in cache. **Note:** It looks up first if the entry exist in `entries` or `rm_entries`. The later case can lead to re-insertions.

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

xcache_invalidate: An `xcache_remove` spin-off. Takes the name of the entry as an argument, looks it up and then removes it Returns on failure: -1. Returns on success: 0. **Note:** Unlike remove, entries can either be on `entries` or `rm_entries` hash table.

6.1.3 Entry eviction

The relevant fields for this purpose can be seen in the following code listing:

```
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 6.6: `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. This entry is the Least Recently Used entry. The doubly-linked list we maintain for this end can be seen below:

[2] #define NoEntry (xcache_handler)-1

```
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 6.7: Doubly-linked LRU list

Lets explain these fields a bit:

lru: Obviously, it's the least recently used entry. It can be considered as the one end of the doubly linked list.

mru: The entry that has just been used. It can be considered as the other end of the doubly-linked list

younger: This entry-specific field points to an entry used right after our entry was used.

older: Same as "younger", it points to the entry that has been used right before our entry was used.

Finally, as we have explained in Section **??**, the eviction internals should normally not bother the user. However, if the user wants to, `xcache` provides the exposes the following functions:

xcache_evict_lru: The name says it all, it evicts the recently used item.

xcache_peek_and_get_lru: This function allows the user to atomically take a peek on the Least Recently Used entry and also update its refcount.

6.1.4 Concurrency control

Locking

Reference counting 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 entry, the reference is decreased by 1. (ref–)
- When a cache entry is evicted by cache, the its ref is decreased by 1. (ref–)

Some common refcount cases are:

Case	Refcount
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

Table 6.1: Reference counting of xcache

- 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)

and, as always, the entry is freed only when its ref = 0.

6.2 Implementation of cached

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.