NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

ANDRÉ JOÃO CÉSAR COSTA

Bachelor in Computer Science

FLEEC: A FAST AND LOCK-FREE APPLICATIONAL CACHE

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

FLEEC: A FAST AND LOCK-FREE APPLICATIONAL CACHE

ANDRÉ JOÃO CÉSAR COSTA

Bachelor in Computer Science

Adviser: João Manuel dos Santos Lourenço
Associate Professor, FCT - NOVA University Lisbon

Co-adviser: Nuno Manuel Ribeiro Preguiça
Full Professor, FCT - NOVA University Lisbon

Examination Committee

Chair: João Ricardo Viegas da Costa Seco
Associate Professor, FCT - NOVA University Lisbon

Rapporteur: Ricardo Jorge Gomes Lopes da Rocha
Associate Professor, Faculdade de Ciências da Universidade do Porto

Adviser: João Manuel dos Santos Lourenço
Associate Professor, FCT - NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2023

FLeeC: A Fast and Lock-Free Applicational Cache

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.11) [35].

To all my teachers.

Acknowledgements

At this moment, I am overwhelmed with feelings of relief and happiness. Although I have enjoyed the process of writing this dissertation, I am glad it is finally over. I deeply value the people in my life and their contributions to this work.

I want to express my gratitude to my adviser, Professor João Lourenço, for his dedication and enthusiasm that pushed me to perform at my best. I also want to thank my co-adviser, Professor Nuno Preguiça, for his valuable insights and pertinent critiques. Moreover, I want to express my appreciation to Professor João Leitão, who, although not directly related to this work, assisted me in reviewing some of it.

Additionally, I want thank the Department of Informatics of the NOVA University of Lisbon and the NOVA LINCS research centre for providing a strong academic foundation that has helped me grow both academically and as a person.

This work would not have been possible without Inês, whose love, kindness and patience were critical in helping me stay motivated. I am grateful for my family's continued support throughout my life; without it, I would not have come this far. I extend my thanks to Rodrigo, Henrique and Tomás, with whom I have had the pleasure of working alongside.

Finally, I want to express my appreciation to dormando, the main Memcached contributor, for his willingness to engage in discussions related to Memcached.

”

“In theory, theory and practice are the same;
in practice, they are not.”

— Yogi Berra

Abstract

Access to data in applications that make use of external storage systems (e.g., Databases) can be a major performance bottleneck. A common solution is to first query a cache application to reduce data access overhead. These cache applications leverage fast main memory access rates and the parallelism capabilities of hardware to provide high performance. To this end, a cache application needs a concurrency control mechanism to maintain correctness under conflicting concurrent accesses, of which mutual-exclusion locks (blocking concurrency control) are the most common mechanism used. Blocking concurrency control strategies fail to provide a high level of performance when under medium to high contention, as the pessimistic nature of blocking concurrency can not completely avoid needlessly synchronizing operations that do not conflict. Conversely, non-blocking (or lock-free) concurrency control strategies can provide high degrees of performance in parallel shared memory contexts, especially in high contention scenarios.

This work proposes FL^{EE}C, an application-level cache system based on Memcached, which leverages non-blocking re-designed data structures concurrency to improve performance by allowing any number of concurrent writes and reads to its main data structure. FL^{EE}C features a hash table with embedded eviction policy to allow the correct functioning of its non-blocking algorithms; a memory reclamation scheme adapted to cache semantics; a non-blocking hash table expansion mechanism to preserve the strong progress guarantees that non-blocking concurrency provides; and dedicated replacement algorithm to minimize false misses that can occur in non-mutually exclusive environments. We have extensively evaluated FL^{EE}C under varied scenarios and workloads and found that, when compared to Memcached, it is capable of achieving 6× higher performance when under high contention scenarios, 1.2× higher performance when under low contention scenarios and equivalent performance in scenarios with no contention. FL^{EE}C can thus be used to further improve the performance of any application that currently uses Memcached.

Keywords: Caching, Concurrency, Non-Blocking, Lock-Free

Resumo

O acesso a dados pode ser uma limitação de desempenho em aplicações que utilizam sistemas de armazenamento externos (e.g. Bases de dados). Uma solução comum consiste em primeiro fazer pedidos a uma cache aplicacional par reduzir o custo associado ao acesso a dados. As caches aplicacionais aproveitam a velocidade de acesso à memória principal assim como as capacidades de paralelismo presentes no hardware para proporcionar alto desempenho. Para este fim, uma cache aplicacional precisa de mecanismos de controlo de concorrência para garantir correção na presença de acessos concorrentes conflituosos. O mecanismo de controlo de concorrência mais comum baseia-se em garantir exclusão mútua em acessos a estruturas de dados. Estes mecanismos (bloqueantes) não conseguem fornecer um alto teor de desempenho quando os níveis de contenção são médio-altos, pois a sua natureza pessimista não consegue evitar totalmente sincronização entre duas operações não conflituosas. Por outro lado, mecanismos de controlo de concorrência não bloqueantes permitem alto desempenho, especialmente em cenários de alta contenção.

Este trabalho propõe o FLEEC, uma cache aplicacional baseada no Memcached, que aproveita estruturas de dados redesenhadas não bloqueantes paa melhorar o desempenho ao não restringir o número de escritas e leituras feitas concorrentemente na sua estrutura de dados principal. O FLEEC tem uma hash table com uma política de eviction embebida com o fim de garantir correção na utilização de algoritmos não bloqueantes; um esquema de reutilização de memória adaptado às semânticas de uma cache; um algoritmo não bloqueante de expansão da sua hash table para manter garantias de progresso fortes provenientes da filosofia não bloqueante; e um algoritmo de substituição que minimiza cache misses falsos que pode acontecer na ausência de exclusão mútua. Avaliámos extensivamente o FLEEC sob vários cenários e descobrimos que, quando comparado com o Memcached, o FLEEC é capaz de ter 6× mais desempenho em cenários de alta contenção, 1.2× mais desempenho em cenários de baixa contenção e desempenho equivalente na ausência de contenção. O FLEEC pode então ser utilizado para melhorar o desempenho de qualquer aplicação que atualmente utilize o Memcached.

Palavras-chave: Caching, Concorrência, Livre de Locks, Não-Bloqueante,

Contents

List of Figures

List of Tables

Acronyms

CAS Compare-and-Swap *(pp. 4, 12–14, 22–24, 29, 40, 41, 43–46, 52, 55, 57, 63)*
CPU Central Processing Unit *(pp. 1, 6, 19, 29, 32, 68, 73, 76, 80, 85, 103, 104)*
DCAS Double-Compare-and-Swap *(p. 24)*

DEBRA Distributed Epoch Based Reclamation *(pp. 28–30, 45, 46, 91)*
EBR Epoch Based Reclamation *(pp. 16, 25–28, 30, 45)*

FRS FleeC Reclamation Scheme *(pp. 45, 46)*

GC Garbage Collection *(p. 14)*

KVS Key-Value Store *(pp. 17–22)*

LRU Least Recently Used *(pp. 2–5, 7, 8, 16, 19, 40–42, 55, 85, 91)*

MTU Maximum Transmission Unit *(pp. 80, 81)*

NIC Network Interface Controller *(pp. 18, 21, 68, 80)*

NUMA Non-Uniform Memory Access *(p. 45)*
OS Operating System *(pp. 8, 18, 21, 29, 48, 76)*

QSBR Quiescent-State Based Reclamation *(pp. 28, 30, 45)*

RAM Random Access Memory *(p. 31)*

RDMA Remote Direct Memory Access *(p. 18)*
SCU Single Compare-and-swap Universal *(p. 23)*

TTL Time to Live *(pp. 8, 47, 72)*

1 Introduction

1.1 Context and Motivation

Caches are systems whose purpose is to help other systems achieve better performance [53]. The increase in performance is accomplished either by placing data closer to where it is required, by storing the data in a faster medium, or by a combination of both. The former aims to achieve lower latency, while the latter's intent is both to reduce latency and to increase throughput. Faster storage mediums are significantly more expensive than slower ones, meaning that it is usually not feasible to have a system's entire data stored in said faster medium. To effectively benefit from faster storage mediums, caches store only a small portion of data, substituting old data entries with newer ones when the storage's maximum capacity is reached. This way systems can have fast data access most of the time, incurring the overhead of fetching data from slow storage only when the data is not present in the cache.

What a cache actually consists of depends on the scale at which it is intended to operate at and with which entities it will serve as a cache to. For example, in the context of a Central Processing Unit (CPU), caches are hardware components that store data (memory contents) that is being manipulated by some core – for a disk, the main memory can be used as a cache.

In the context of this work, a cache is a piece of software that serves applications that also communicate with a database. Access to data that is stored separately from the system can be a major bottleneck for applications. These types of caching systems are useful because database systems store data in slow persistent storage, so having a caching layer can significantly increase application performance as well as help minimize database congestion.

The particular caching system that will be of focus is Memcached [42], an open-source cache application that has seen widespread use. The most relevant requests Memcached performs are the GET and SET operations, similarly to any other caching system. A GET request checks if a key is cached — if it is successful we say that a HIT occurred and the corresponding value is returned; if it is unsuccessful we say that a MISS occurred, in this

case either the client or the cache itself is tasked with placing the value into the cache. A SET request asks the server to either add a new key-value pair to the cache or to update an existing one if an item with the same key already exists. A SET request can have two different outcomes: either the cache is not full and the key is simply inserted, or the cache is full and an "old" entry is removed before the new one is added. To decide which "old" entry shall be discarded, caches employ an eviction policy (e.g. Least Recently Used (LRU)).

Optimizing a Caching system is crucial in order to achieve the end goal of improving the performance of the systems that use it. Memcached and similar caching systems usually employ a scale-out scaling strategy, as opposed to a scale-up one. This way client's requests are spread among multiple caching servers, effectively distributing and diminishing the load each server experiences, potentiating faster response times for clients. The scale-out strategy adapts very well to uniform workloads, as every server will receive approximately the same amount of requests per time unit, preventing scenarios where only a few servers are saturated and the overall system performance is not optimal. In reality, most workloads are highly skewed [5], which means that the load will not be evenly distributed across the set of caching servers. For this reason, it is also important to improve the performance of each server individually, not solely relying on a scale-out strategy. This way each server can sustain higher loads, effectively improving the performance of the overall system both under uniform and skewed workloads.

1.2 Problem Statement

If there are multiple threads concurrently accessing and modifying shared data, some form of concurrency control is required to prevent inconsistent states. The most popular concurrency control mechanism for multithreaded systems is the mutual exclusion policy, which limits concurrency in well-defined (critical) sections, which are usually protected by locks [15, 1]. The simplest synchronization strategy to implement is Coarse-Grained Synchronization [25], which is also the slowest as it is essentially equivalent to sequential accesses to objects. More complex locking strategies exist, such as ones that allow for finer grain, namely "hand-over-hand locking" [25], and ones that allow for multiple simultaneous readers, such as "optimistic locking" [25]. Although improving locking strategies allows for a performance increase, one must deal with the fact that the use of locks is accompanied by considerable disadvantages [18, 22, 30, 4, 21]:

- Each access to a resource will have significant overhead, even when there is little to no contention.
- When there is contention some processes will idle, resulting in performance loss.
- Locks are not composable, meaning that it if there is a correct system that uses locks, it is difficult to build another system on top of the first one without knowing how

it is implemented w.r.t which locks are used and how (it is difficult to apply the *modular programming* software design technique).

- Even if a program is proven to be deadlock and starvation-free, it is possible that a process/thread crashes while holding a lock, resulting in resource starvation.

- Locks are pessimistic, as they assume conflicts happen at every access, and because of this prevent concurrency from ever occurring in certain sections of a program[1].

Some of the problems that result from the use of locks can be solved by non-blocking algorithms [46, 48, 21], which utilize atomic read-modify-write primitives to deal with concurrency. These primitives are implemented in hardware and they enable atomic modifications to small amounts of memory. Having "smaller" atomic modifications, i.e., smaller critical sections, allows for better scalability. When comparing non-blocking to Coarse and Fine-Grained lock-based synchronization, there is considerably less synchronization between two operations that do not conflict, meaning that there is less overall probability that processes slow each other down [25]. When comparing non-blocking to Optimistic or Lazy synchronization the benefits are less notable but are largely due to the fact that non-blocking algorithms allow for better progress conditions, as processes that do not block can still make progress when arbitrary delays occur [25]. Furthermore, as there are no locks being used in non-blocking synchronization, there is no possibility of deadlocks [46].

1.3 Objective

Memcached utilizes a lock-based Fine and Coarse-Grained concurrency control strategy [32] for its core data structures: the hash table, the LRU linked list and a slab allocator. The main objective of this work was to improve the individual performance of Memcached by substituting the concurrency control mechanism of its main data structures, replacing a blocking concurrency control strategy with a non-blocking one. This way, we reduced the time that Memcached's threads spend waiting for mutual exclusive accesses which decreases the total processing time of each request, resulting in lower latency and an increase in the maximum throughput an individual server can sustain. These performance improvements are proportional to the amount of contention the server experiences, as is expected when comparing blocking to non-blocking concurrency control [18]. An additional positive side-effect of replacing Memcached's blocking concurrency strategy are the stronger progress guarantees that non-blocking synchronization offers, allowing the system to proceed under partial (thread) crashes.

[1] As opposed to an optimistic approach where conflicts are "repaired" when they happen.

1.4 Contributions

The main contribution of this work is our new non-blocking (Memcached-based) applicational cache: FLEEC, which can be further subdivided into 4 contributions, necessary to make it correct and performant under a wide range of scenarios:

- We merged Memcached's lookup (hash table) and eviction policy (LRU) structures into one single structure that performs both functions: a hash table with embedded CLOCK-based [9] eviction policy. This way we can guarantee correct serialization under non-blocking synchronization.
- Memcached's hash table buckets implemented as singly linked lists were substituted by Harris' non-blocking linked lists [22]. So that memory could be safely re-utilized under an environment without mutual exclusion, we implemented a memory reclamation scheme based on DEBRA [8] that took advantage of cache semantics to minimize memory reclamation overhead.
- We substituted Memcached's stop-the-world style hash table expansion with a non-blocking expansion mechanism, enabling FLEEC to preserve the strong progress guarantees inherent to its non-blocking nature.
- We devised two dedicated non-blocking linked-list replacement algorithms to minimize performance-degrading false misses that can occur in non-mutually exclusive environments.

These new data structures can provide fast wait-free lookup operations and implement an eviction policy, only ever synchronizing through the use of the Compare-and-Swap (CAS) atomic primitive, allowing for any number of concurrent writes and reads without process stalling. These changes result in FLEEC's increased capability to efficiently handle high contention scenarios. When compared to Memcached, FLEEC provides 6× higher performance when under high contention, 1.2× higher performance when under low degrees of contention and equivalent performance when contention is practically non-existent. Additionally, our contributions also include a detailed profiling and experimental evaluation of Memcached and FLEEC.

1.5 Document Organization

The rest of this dissertation is structured as follows: in Chapter 2 we introduce the reader to important concepts such as Memcached and blocking and non-blocking concurrency control; in Chapter 3 we introduce the state-of-the-art caching systems and non-blocking concurrency strategies; in Chapter 4 we cover our new system's design; in Chapter 5 we talk about advanced details; in Chapter 6 we evaluate our new solution and its intermediate steps and compare them to Memcached; and in Chapter 7 we give our concluding remarks and hint at possible future work.

2

Background

In this Chapter, we go over the inner functioning of Memcached (Section 2.1) and examine blocking and non-blocking concurrency management (Section 2.2), focusing on non-blocking concurrency control and its challenges.

2.1 Memcached

Memcached has three main structures: a hash-table to keep track of which items are in the cache; an LRU to implement an eviction policy; and a slab allocator [6] to efficiently use memory by minimizing memory fragmentation and reducing memory management overhead. Memcached's hash table is implemented as an array of singly linked lists to act as collision buckets. Memcached's LRU is implemented as a doubly-linked list. Current Memcached has three LRUs, instead of just one, to minimize contention. We will discuss the multi-LRU model, but for the time being, we will assume that only one LRU exists for the sake of simplicity. Memcached's slab allocator is a structure that keeps track of which portions of a set of fixed size memory blocks are in use, essentially partitioning memory into indivisible slots in which items will be inserted.

2.1.1 Hash Table

Memcached's **hash table** is implemented as a set of singly linked lists acting as hash buckets. A typical Memcached procedure starts by accessing the hash table to check whether the item is present or not. Any access to the hash table, be it a read or a write, must first acquire a lock to guarantee mutually exclusive accesses. The lock to be acquired depends on which hash bucket the process wants to access: each lock is mapped to a subset of buckets, so that some degree of concurrent reads and writes can be executed in parallel. The number of hash table buckets each lock protects depends on the number of threads running in the system and the initial hash table size. This dependency aims to balance the amount of memory used by each lock with the amount of contention mitigation a higher quantity of locks provide: if there are few threads, then there will likely be less contention

meaning the system can afford to have a lesser amount of locks without compromising on performance.

To traverse the hash table's buckets (i.e. linked lists), we follow a chain of references that point to non-contiguous memory locations. This access pattern exhibits low cache locality, which makes bucket traversal expensive as each traversal step likely has to fetch memory from main memory. Furthermore, as Memcached stores item data and meta-data contiguously in the same memory block, a lot of the data loaded into the CPU cache is not useful for a hash bucket traversal, which further degrades the performance of the hash table search operation.

To minimize the impact of traversing hash table buckets Memcached performs a hash table expansion, i.e., it doubles the number of buckets when the number of items exceeds 1.5 times the number of buckets. This means that the average number of dereferences per traversal is at most 1.5, limiting the negative impact of the cache unfriendly linked lists. The number of locks that protect hash table buckets does not grow when a hash expansion occurs. This means that, as the hash table expands, the granularity of concurrent hash table accesses remains the same as each lock always protects the same percentage of buckets throughout execution. Since the amount of locks depends on the initial hash table size, these problems can be solved with good system configuration. Memcached's hash table expansion is rather simple. When Memcached detects that an expansion should occur it stops all threads that do modifications to its data structures, creates a new hash table, and resumes all threads. Subsequently, a helper thread (the *hash table expansion thread*) starts iterating through the old hash table, moving items from the old hash table's buckets into the correct bucket of the new hash table. During expansion, other threads can also modify and read from the hash table, as they know which buckets the expansion thread has already updated and therefore know in which bucket the item they are accessing is in. It is worth noting that configuring Memcached so that it has a high number of buckets from the beginning would waste a considerable amount of memory, which is why it is best to allow the system to adapt the number of buckets during execution.

2.1.2 Slab Allocator

Memcached keeps all its data in memory so that the cost of fetching and modifying that data is reduced. Since Memcached aims to provide excellent performance, it should manage its data efficiently and with low processing overhead. To solve this problem Memcached uses a slab allocator [6], an algorithm initially used for kernel memory allocation. Memcached's slab allocator uses coarse-grained synchronization as it is protected by a single lock.

The slab allocator is essentially a resource pool where each resource is a memory region. A given amount of memory is divided into multiple *slab classes*. Each slab class contains many *slabs*, that are memory blocks of a fixed pre-determined size. When in need of a block of memory of size S, a request is sent to a slab class whose slabs have a size of at least S. The slab class then returns a slab to be used to store an item and removes that

slab from its list of available slots. When an item is evicted, and we want to reclaim its slab, we simply give it back to its slab class, which in turn adds it to the list of available slots so that it can be reused in the future.

The slab allocator provides low management overhead [6] because acquiring and releasing a slab only requires a simple removal/insertion of slots in a list. The slab allocator also provides high memory efficiency [6] since it reduces memory fragmentation when compared to an ad hoc memory manager or other memory allocation schemes [6]. Furthermore, as the number of slab classes and their slabs' size is easily configurable, the slab allocator can adapt to various workloads.

2.1.3 LRU

Memcached needs to implement an eviction policy so that when there is not enough free memory to store a new item, Memcached can decide which item to delete to make space for a newer entry. Memcached uses the LRU eviction policy: it evicts the item that was least recently used, i.e., the item that was not accessed for the longest time. Naturally, the LRU policy captures a useful temporal access pattern, as items that were used most recently are more likely to be re-accessed than items that were used least recently. This way, evictions remove items that are less likely to be accessed in the future, minimizing the probability of subsequent client requests resulting in a cache miss, and maximizing Memcached's hit ratio.

Memcached's LRU eviction policy is implemented as a doubly-linked list. We refer to this doubly-linked list as the LRU, while we refer to the eviction policy as the LRU eviction policy. The item at the LRU's head is the item that was most recently accessed. Inversely, the item at the LRU's tail is the least recently accessed item. New items are inserted into the LRU's head, while old items (that we want to evict) are removed from the tail. When items are accessed they are *bumped*, i.e., they are removed from their current place in the LRU and re-inserted into the LRU's head, so that their recency is updated. The LRU is implemented as a doubly-linked list so that there is no need for its traversal: when evicting or inserting we can look directly into the LRU's tail or head, respectively; when bumping items we can remove them directly from their current place in the LRU as we have already found the item by traversing the hash table[1].

To allow multiple threads to access the LRU concurrently, the LRU is protected by a Coarse-Grained lock. Older versions of Memcached had performance issues due to high contention on the LRU lock, which led newer versions to employ a more granular solution by having multiple smaller LRUs rather than one large LRU. Multiple LRUs reduce the amount of conflicting accesses and therefore reduce the amount of time each thread spends waiting to acquire a LRU lock.

[1] All item data and meta-data (which includes its hash table and LRU pointers) is stored inside of the slab assigned to that item. This way, when an item is found through the hash table its place in the LRU is also located.

Current Memcached divides its LRU in two ways: by slab class and by recency/popularity. There is one LRU for each slab class, where each LRU only contains items from the corresponding slab class (i.e., items of the same size). When we want to evict an item, we know exactly how much memory the new item will require, meaning we can target evictions on items of the same size by evicting from the corresponding LRU.

The other way in which Memcached divides its LRUs is by item recency. Per slab class, there are three LRUs: the HOT, WARM and COLD LRUs. The HOT LRU contains new items. Items in the HOT LRU can not be bumped nor evicted, as most workloads present strong temporal locality and items are very likely to be accessed again shortly after insertion. The WARM LRU contains items that were accessed at least twice since the last time their place in the LRU was updated. The WARM LRU serves as protection from scan access patterns that can flush the cache's entire data [29, 28]. Finally, the COLD LRU contains inactive items (ones that have been accessed 0 or 1 times since their last LRU update). Only items in the COLD LRU are candidates for eviction. Memcached has an extra thread that is tasked with moving items from one LRU to the other[2].

So, Memcached has $3 \times \#slabclasses$ LRUs, each protected by its own lock, effectively improving the granularity of concurrent accesses. Furthermore, Memcached tries to minimize the amount of accesses to each LRU by not performing more than one bump in a fixed time interval (e.g. not performing a bump if the item was bumped in the last second).

2.2 Concurrency Management

A sequential program consists of a series of instructions that are executed one after another, in an orderly fashion. Sequential programs are easier to reason about as they are deterministic and produce the same result over and over, unless randomness is purposefully added [38]. In contrast, parallel programs are naturally non-deterministic as their result usually depends on the unpredictable timing of tasks.

Concurrency and Parallelism are two different but interconnected concepts. Concurrency occurs when tasks are being executed in overlapping time, meaning that two tasks that have some overlap between their start and end time are said to be concurrent [38]. Concurrency is possible with a single processor, e.g. if an Operating System (OS) interrupts a process to give processor time to another process, the two processes are running concurrently. Parallelism refers to work being carried out simultaneously. For an execution to be parallel, it requires more than one execution unit to execute instructions at the same time.

[2]Additionally, the LRU thread also checks if items have outlived their Time to Live (TTL), and removes them if so.

2.2.1 Shared Memory

There are two main ways for processes to communicate: Message Passing and Shared Memory. In the Message Passing Model processes communicate by sending messages to each other; it can be used for communication between processes in the same machine or on different machines. In the Shared Memory Model processes interact by accessing the same address space through *shared variables* [38]. When a process modifies a shared variable by writing to it, that modification is visible to other processes. Processes also possess their own *local variables*, which only they can read and write to. In the context of this work, we focus on the Shared Memory Model, as it more accurately represents the underlying hardware architecture [10] and does not require an additional abstraction layer for communication. This enables a finer control of resources, as well as a lower processing cost, which contributes to better performance.

The issue of a *race condition* arises when two or more processes try to access the same shared variables simultaneously without any synchronization, and at least one of those accesses is a write [38]. A computation suffering from a race condition might still produce correct behaviours some of the time, but may also produce incorrect behaviours, depending on the *interleaving* of the tasks involved in the race condition. An interleaving is a specific execution ordering of the operations of multiple processes.

For example, consider two processes trying to add one to a shared counter at the same time. It is expected that the resulting counter contains the previous value of the counter plus two, but without synchronization that might not be true.

The example in Figure 2.1 presents a specific interleaving that constitutes a race condition:

Thread 1		Thread 2
`x = counter;`	1	
`x = x + 1;`	2	
	3	`y = counter;`
	4	`y = y + 1;`
`counter = x;`	5	
	6	`counter = y;`

Figure 2.1: An interleaving of two threads with a race condition.

In the above example, two processes read the value of a COUNTER (lines 1 and 3), they make modifications to their local variables that contain the counter's value read (lines 2 and 4) and subsequently both processes try to write the new value to the counter (lines 5 and 6). The write of Thread 1 (line 5) is lost, as it is overwritten by Thread 2's write (line 6). Thread 1's efforts should have impacted the counter's final value, but they did not. This occurrence constitutes a race condition, as the semantics of the increment operations are corrupted by the concurrent accesses.

For a parallel program to be correct, all its possible interleavings must be correct [7]. The program represented above is not correct, as the presented interleaving does not result in correct behaviour.

Concurrency control encompasses a class of mechanisms and methodologies that serve as ways to synchronize processes, ensuring that no incorrect interleavings occur.

2.2.2 Blocking Concurrency Control

One way of going about Concurrency Control is by defining sections of a program that are only accessed by at most one process at a time. These regions are referred to as *critical sections*. Critical sections are said to be accessed under *mutual exclusion*, i.e., they can only be executed by one process at a time [25].

A common way to ensure mutual exclusion is through the use of *locks*. A process must first gain access to a resource by *acquiring* its lock. After acquiring access to a resource and utilizing that resource, a process will let go of the access to the resource by *releasing* its lock. Releasing a lock allows other processes to acquire the right to the corresponding resource. Lock implementations must ensure that a maximum of one process can acquire the lock at any point in time, enabling mutual exclusion.

Let us revisit the previous example where a counter was being concurrently incremented. We can synchronize accesses to that counter with locks so that the semantics of the increment are correctly followed:

In the example of Figure 2.2, we have successfully eliminated the race condition illustrated in Figure 2.1. Now, processes can not access the counter simultaneously, as they first have to acquire its lock and only one process can have the lock at any given time.

Thread 1

```
acquire_lock(); //Acquires lock

x = counter;
x = x + 1;
counter = x;
release_lock();
```

Thread 2

```
1
2  acquire_lock(); //Has to wait
3
4
5
6  //Acquires lock
7  y = counter;
8  y = y + 1;
9  counter = y;
10 release_lock();
```

Figure 2.2: An interleaving of two threads with a race condition fixed.

Mutual exclusion is not the only desirable property. Consider a lock implementation that never allows a process to access the critical section. That lock would still guarantee that critical sections were accessed under mutual exclusion because no process could ever access it! Mutual exclusion is a *safety* property [25], to ensure progress we need a *liveness* property. Intuitively, safety properties ensure that *nothing bad ever happens*, while liveness properties ensure that *something good will eventually happen*.

Deadlock-freedom is a liveness property that says "if some process attempts to acquire a lock, some process will eventually succeed in acquiring that lock" [25]. By requiring deadlock-freedom, we no longer allow the lock implementation where no process ever acquires a lock. Deadlock-freedom ensures that some process will eventually acquire the lock and make progress, meaning that the system will also make progress. Since deadlock-freedom only requires *some* process to make progress, it still allows other processes to be *starved*. Imagine that two processes, P_1 and P_2, are trying to acquire a lock. If we implement that lock such that it only allows P_1 to acquire the lock, then that lock guarantees both mutual exclusion and deadlock-freedom, but it will *starve* P_2.

Note that we also want to ensure deadlock-freedom at the algorithm level. Imagine a possible *deadlock* scenario caused not by the lock implementation but by the algorithm: P_1 acquires $lock_A$, P_2 acquires $lock_B$, then P_1 tries to acquire $lock_B$ and P_2 tries to acquire $lock_A$; both P_1 and P_2 are waiting for each other, resulting in a deadlock.

Starvation-freedom is a liveness property that ensures "if some process attempts to acquire a lock, it will eventually succeed" [25]. Starvation-freedom is stronger than deadlock-freedom since algorithms that ensure starvation-freedom also ensure deadlock-freedom. If an algorithm is starvation-free then no processes will be starved, and therefore all processes will eventually make progress.

It is possible to go a step further and not only guarantee that no process will be starved but also that the order in which processes acquire locks is *fair*, for some definition of fair, such as *first-come-first-served* accesses. To be able to define fairness in this way, we need to first define what *first-come* means [25]. We can divide the procedure of acquiring a lock into two sections:

- A *doorway* section, whose execution will take a bounded number of steps; followed by
- A *waiting* section, whose execution will take an unbounded number of steps.

A lock allows for first-come-first-served fairness if it guarantees that if some process P_1 finishes its doorway section before some other process P_2, P_1 will also finish its waiting section before P_2 [25].

2.2.3 Non-Blocking Concurrency Control

Non-blocking concurrency control mechanisms are mechanisms where processes are never idle waiting for other processes. Non-blocking concurrency is an *optimistic* approach to concurrency as it resolves *conflicts* instead of preventing them. To this end, Non-Blocking algorithms utilize *atomic primitives* that allow operations to be performed atomically. An atomic operation will appear as if it was executed instantaneously from the perspective of other processes, which means that a process can not interfere with other processes' atomic operations.

2.2.3.1 Atomic Primitives: Compare and Swap

The CAS is an atomic instruction that is implemented in hardware in most modern multiprocessor architectures. The CAS operation, whose pseudo-code is presented in Algorithm 1, takes three arguments: a POINTER to a memory location; an EXPECTED value of that memory location; and a DESIRED value to be put into that memory location. If the memory address pointed to by POINTER contains the EXPECTED value, then the DESIRED value is written into that memory location and TRUE is returned. If the value in memory differs from the EXPECTED value, then no operation is performed and FALSE is returned.

Algorithm 1 The Compare and Swap primitive: pseudo-code

```
1:  function CAS(a: address, e: expected value, d: desired value)
2:      BEGIN ATOMIC
3:      if *a = e then                    ▷ Check if address contains expected value
4:          *a ← d
5:          return true
6:      else
7:          return false
8:      end if
9:      END ATOMIC
10: end function
```

The difference between locks and CAS is that locks allow for a user-defined critical section, while CAS limits the critical section to the execution of the compare and swap operation. The lock operation is more costly than performing a CAS as it requires two atomic operations on the lock data structure: one for lock acquisition and one for lock release [39].

2.2.3.2 Progress Conditions

The nature of Non-Blocking algorithms is different from Blocking algorithms, motivating the need to define different progress conditions. For example, since Non-Blocking algorithms do not use locks they will never deadlock. Therefore it is not very descriptive to say that a Non-Blocking method is deadlock-free.

Progress conditions say something about the progress of the system as a whole, as well as the progress of individual processes. Progress is defined on a per-method basis since different methods might have different progress conditions.

A method is **obstruction-free** if a process executing in isolation finishes in a finite number of steps [25]. A process is executing in isolation if no other processes take a step.

A method is **lock-free** if some process finishes in a finite number of steps [25]. Lock-free methods guarantee that at least one process will make progress (but allow other processes to starve), therefore ensuring the whole system will also make progress.

A method is **wait-free** if all processes finish in a finite number of steps [25]. Methods that are wait-free guarantee that both the system and all individual processes will make progress (ensuring that no processes will starve).

Note that a lock-free method is also obstruction-free, and a wait-free method is lock-free and obstruction-free.

2.2.3.3 The ABA Problem

The ABA problem is a type of problem most Non-Blocking algorithms are subject to. The ABA problem manifests itself when:

- process T_1 reads some variable x, observes A and suspends arbitrarily;
- process T_i, $i \neq 1$, writes B to x;
- process T_j, $j \neq 1$, writes A to x;
- process T_1 performs a successful CAS on x, since it expected and observed A.

This general execution describes a potential false positive if CAS is used under the assumption that x had not been modified since being read. The occurrence of ABA constitutes an issue if the semantics of the problem disallow it. Generally, programs where the variables that are being CAS'ed contain values where different instances of the same value are interchangeable, do not to need prevent the ABA problem. For example, consider a simple atomic increment and decrement of a shared integer variable: if the variable initially contains A, is then changed to B and back to A' (such that A' is a different instance of the A and represents the same value), the occurrence of ABA does not constitute a problem as the values A and A' are interchangeable as different instances of an integer are equivalent.

Usually, the behaviour that stems from having an ABA prone algorithm is hazardous if the variable that is being CAS'ed is a reference to a memory location. This is due to the fact that the **same** memory location can contain different values at two different points in time. Therefore, if a program assumes that a successful CAS indicates that the value contained in a memory location has not changed, then an ABA occurrence can lead the program to "confuse" the values it is manipulating, corrupting the intended functioning of the program.

For example, assume an object O that is pointed to by a shared pointer P_S. A general ABA prone algorithm would have processes first store P_S in some local variable P_L, compute a desired change to the object and attempt to serialize that change by performing a CAS. The procedure of attempting to change P_S if it still matches P_L ignores the possibility of object O having been freed and some other object O' being allocated in the same memory location that previously contained O. If this happens object O' will be treated as if it were object O, possibly enabling inconsistencies in the algorithm. For example, if process T_i initially wanted to remove O, but O was removed concurrently by

some other process, it might end up removing O′ instead, as can be seen in Figure 2.3. If the algorithm does not account for the ABA problem, it may allow undesired modifications to occur.

Figure 2.3 shows concurrent interactions between three Threads; it depicts an occurrence of ABA in a linked list where Thread 1 wanted to remove an element with key "2" but ended up removing an object with key "5" due to other threads concurrently removing and inserting objects. The problem occurred because Thread 1, through a successful CAS, assumed the element with key "2" had not yet been removed.

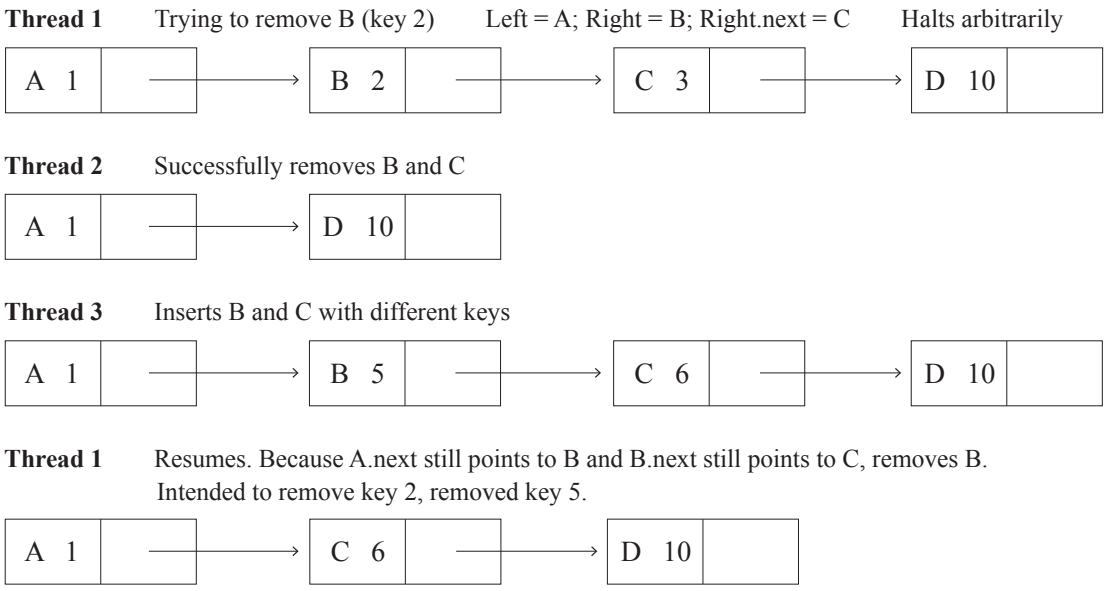


Figure 2.3: Example of ABA in a Non-Blocking Linked-List. Letters represent memory locations. Numbers represent keys stored in each node.

2.2.4 Memory Reclamation Problem

The Memory Reclamation problem originates from the presence of Non-Blocking concurrency in programming environments that lack a Garbage Collector. Memory Reclamation is a relevant problem because many systems are built with languages that require manual memory management (languages such as C and C++).

Memory Reclamation is a weaker problem than Garbage Collection (GC) [3]. Techniques applied in GC are usually too strong to be applied to Memory Reclamation as they significantly hinder performance. Since the need for Memory Reclamation solutions stems from the presence of concurrency in Non-Blocking algorithms, and the end goal of Non-Blocking algorithms is performance improvement, efficiency is a critical property in Memory Reclamation solutions.

Memory Reclamation is trivial in algorithms that employ a mutual exclusion policy, since after acquiring a lock any memory reference protected by that critical section is guaranteed to only be accessible by the process holding the lock. Reclaiming Memory

in Non-Blocking algorithms is much more difficult because of the optimistic nature of Non-Blocking algorithms, which allows concurrent accesses to be performed without any synchronization between processes (e.g. data structure traversals). There must exist a mechanism that allows the system to recognize when it is safe to reclaim[3] a memory location.

Intuitively, Memory Reclamation techniques aim to be able to detect when there are no private variables that are instances of "deleted" shared variables (a reference that points to an object that was removed from the data structure). The correct application of a Memory Reclamation solution prevents problems that occur when it is not clear if objects have been deleted. For example, Memory Reclamation prevents *dangling pointers* by not allowing objects that **may** be accessible to be deallocated.

Due to the fact that Memory Reclamation impedes the deallocation of memory locations that are pointed to by some private variable, it often also solves ABA as a side-effect [46]. This is because for ABA to occur there has to be more than one "instance" of A in the system at a given point in time. An instance can be seen as what processes "think" a memory location contains. For example, if process P_1 acquired a pointer to A, then A was deallocated and later reallocated, then P_2 acquired a pointer to A, we say that processes P_1 and P_2 point to different instances of A since P_1 will treat the memory location referenced by its pointer as if it contained the object before deallocation and P_2 will treat the it as the object after deallocation.

It is important to understand the life-cycle of a record (as described in [8]) to be able to fully comprehend both the need for specialized Memory Reclamation as well as its solutions. A RECORD is an amalgamation of one or more primitive objects (commonly called a STRUCT). In the context of data structures, records contain at least one POINTER to another record, so that all records in the data structure are accessible by following some path from an initial record called an ENTRY POINT. A record's life cycle has five distinct states:

- UNALLOCATED: all records begin as unallocated. An unallocated record means that no memory has been reserved for this record's use. An unallocated record transitions to the UNINITIALIZED state after being ALLOCATED.

- UNINITIALIZED: an uninitialized record is a record that has had memory reserved for its use. An uninitialized record must *not* be accessible through other INSERTED records, i.e., it should not be in the data structure. An uninitialized record becomes inserted after it has been initialized and successfully inserted into the data structure.

- INSERTED: an inserted record is a record that is accessible from following other inserted record's pointers starting from the entry point.

[3]The terms "reclaim", "deallocate", and "free" are used interchangeably to refer to the process of releasing memory.

- RETIRED: a retired record is one that is not accessible by traversing the data structure starting from the entry point record. A retired record may or may not be accessible by processes that were traversing the data structure when the record transitioned from the inserted to the retired state.

- SAFE TO FREE: a safe to free record is a record that can not be accessed by any process through the data structure or through private references.

Non-Blocking algorithms have the responsibility of transitioning records between the UNINITIALIZED and INSERTED states, and between the INSERTED and RETIRED states. It is the responsibility of the Memory Reclamation scheme to transition records between the RETIRED and SAFE TO FREE states.

The quote "We call a memory reclamation scheme automatic if it performs all of the work of reclaiming a record, and non-automatic if it requires a data structure operation to invoke a procedure whenever it removes a record from the data structure." [8] succinctly explains the definition of automatic Memory reclamation schemes. Furthermore, non-automatic methods may require complex analysis of the already designed algorithm to ensure that the applied Memory Reclamation solution functions correctly. For example, Hazard Pointers [46] require the identification of *ABA hazards* in all operations, as well as determining where certain procedures should be executed. In contrast, automatic methods, such as Reference Counting require no modifications to the algorithm in the sense that the behaviour needed for proper memory management can be automatically added by a compiler.

A Memory Reclamation solution is said to be fault tolerant if it defines a finite upper bound on the amount of records that can not be safely reclaimed in the case of process crashes. For example, Epoch Based Reclamation (EBR) [18] does not allow the safe reclamation [8] of any records after one process crashed, meaning that it is not fault tolerant as a process crash means that the number of unreclaimed records may grow infinitely.

2.3 Summary

In this Chapter, we presented Memcached and concurrency management strategies. We covered Memcached's hash table used for indexing, its LRUs used to manage its eviction policy and its slab allocator for fast and efficient memory usage. We also examined blocking and non-blocking concurrency, going over their fundamental idea and progress conditions. In the case of non-blocking concurrency, we also addressed its main difficulties, such as the ABA and memory reclamation problems.

3

Related Work

In this Chapter, we will examine previous application caching systems by analysing their constituent components, and discuss solutions that must be solved in any system that makes use of non-blocking concurrency control: the ABA and memory reclamation problems.

3.1 Caching Systems

A Key-Value Store (KVS) is a data storage system with a simple data model consisting of key-value pairs and a simple GET/PUT interface. KVSs are commonly used to store data in memory, to aid applications in having fast response times. KVSs can either be persistent (e.g., RAMCloud [50]) if they persist stored data across failures and reboots, or non-persistent (e.g., Memcached [42]) if data is lost in the case of crashes. Additionally, KVSs can present either *Store* or *Cache* semantics [34]. *Stores* only delete elements when the client explicitly requests them to, whereas *Caches* can delete elements if there is no available storage when attempting to insert new elements. In the context of this work, we are more interested in *Key Value Caches*, although we only make this distinction when necessary.

Designing a KVS can take multiple directions. A common objective is to improve its performance by maximizing throughput and minimizing latency [26, 36, 17, 34]. Some works also focus on other aspects, such as minimizing the amount of space required to manage each stored element [17] or providing range queries [36].

Despite presenting simple semantics, many combinations of design choices can be made when conceptualizing a KVS. The KVS stack can be broken down into four essential components [32]:

Network Stack how multiple client connections and messages are managed.

Indexing and Eviction policy the data structures used to store keys and aid in item eviction policy.

Concurrency Control Mechanism how data consistency is maintained in multiprocessor environments.

Memory Allocation how memory for the values kept are managed.

3.1.1 Network Stack

At their core KVSs are network applications, as clients and servers utilize the network to communicate. The server-side process of receiving requests and sending responses to clients can take a significant amount of time out of the entire computation cycle of a KVSs [44]. There are two distinct approaches to deal with Network I/O: the **socket I/O** abstraction (Kernel level) and **direct Network Interface Controller (NIC) access** (User level).

The socket I/O is commonly provided by the underlying OS, meaning that it is almost universal and simpler to develop applications that use it. The downside is that the socket abstraction yields less peak throughput because of a higher per-read overhead [34]. Given that network I/O is quite literally the entry point of a KVS, it can easily bottleneck its performance. Some solutions try to utilize client-side batching by sending multiple operations in a single request, as a way to diminish the overhead of reading client packets; the problem is that this increases latency and is not realistic when distributing requests across multiple servers through *sharding* [34].

The alternative to the OS socket abstraction is to have direct access to the NIC. By bypassing the kernel, the system can effectively diminish communication overhead, at the cost of losing access to useful TCP [16] features. Systems such as MICA [34] use direct NIC access to achieve line-rate packet processing. The purpose of direct NIC is similar to Remote Direct Memory Access (RDMA)'s [51] since we are trying to avoid costly computation to allow for high-throughput, low-latency networking. The main difference is that KVSs use direct NIC access as a way for servers to be able to access packets with less overhead, with no client-side knowledge of this mechanism; RDMA is more cooperative in the sense that both the client and the server are aware of and following the RDMA protocol.

3.1.2 Indexing and Eviction Policy

The indexing component of a KVS is how the system keeps track of cached objects; the eviction policy is how the system chooses which items get evicted[1]. The indexing component goes hand in hand with the eviction policy. Some indexing and eviction policy structures are more general, meaning they are more easily combined in other configurations (such as Memcached's Hash-Table and LRU), while others are tightly

[1]The eviction policy only applies to *Key Value Caches* as *Key Value Stores* do not delete items without an explicit client request

bound together and difficult to re-utilize in other settings (such as MemC3's [17] Cuckoo Hashing and CLOCK like eviction).

Indexes and Eviction policy's data structures should aim to be space efficient and take advantage of space locality. Space efficient solutions have less metadata per stored item, therefore more space for actual data and a higher hit ratio. Solutions that take into account space locality are faster, as there are less CPU cache misses while traversing data structures.

It is the Indexing component's responsibility to associate the keys of key-value items to their value. More accurately, the Indexing component is used to find the memory location where an item's value is stored. Indexing can either be Read-oriented or Write-friendly [34]. Read-oriented indexing means that the data structure is optimized for read operations, but has costly write operations. A Write-friendly index means that reads are not optimized, but writes are not as expensive. The choice between a Read-oriented vs. a Write-friendly index depends on the workload. As workloads can reach a 30:1 GET/SET ratio [5], many solutions prefer to have Read-oriented indexing.

Eviction Policies are used in Caches to be able to discern which elements should be evicted when required. A common eviction policy is the LRU policy, which maintains an ordering of items based on their most recent access, and chooses to evict the *least recently used* item. A strict LRU is expensive as it uses *bookkeeping* to update items as they are accessed [34]. Because of this, a typical approach is to use approximate LRUs to be able to cut corners and gain on performance. Approximate LRUs mimic the behaviour of a *strict* LRU, while they are not so rigid in their functioning, e.g. while a strict LRU updates item's "used status" (re-inserts them at the head of the queue) every time an item is accessed, we could approximate an LRU if we only updated items every other time. Approximate eviction policies fit particularly well in the semantics of a cache, as there are no guarantees that items will remain in storage. The downside of using approximate eviction policies is that they might diminish the hit ratio [2].

Some solutions leverage cache semantics by using a lossy index. For example, MICA [34] uses hash-table collisions to evict items (with an extra mechanism to prevent recently used items from being removed), leading to fewer resources and space being required for the eviction policy portion.

3.1.3 Concurrency Control

Modern systems leverage multi-core architectures for performance gains and KVSs are no exception. Concurrency control is the methodology the system uses to efficiently benefit from parallelism while ensuring the consistency of its data structures in case of concurrent accesses. Concurrency Control can refer both to ensuring data structure's consistency and higher-level semantics (such as implementing transactions). In the context of this

[2]The lower hit ratio depends on more factors. For example, MemC3 has LRU-approximating eviction algorithm based upon CLOCK, that in turn uses significantly less space, meaning that the system can allocate more memory for key-value items and achieve a higher hit ratio because of it.

work, Concurrency Control refers to the process of avoiding concurrent accesses from corrupting data structures. Conflicting concurrent accesses happen when two operations access the same data, and at least one of them is a write operation. There are two types of coordination: writer-writer coordination and writer-reader coordination. Previous KVSs have mainly used three concurrency control mechanisms to guarantee the consistency of their data structures: coarse and fine-grained locking [42], versioning-based optimistic locking [17, 36, 34] and partitioning [34].

A Coarse and/or fine-grained locking mechanism essentially means that both writer-writer and writer-reader conflicts are resolved in the same way, i.e., both write and read operations require the use of locks. Both coarse and fine-grained synchronization imply that there is implicit (unnecessary) reader-reader synchronization, and since many workloads contain an overwhelming majority of read operations, coarse and fine-grained locking will result in a huge quantity of unnecessary coordination, resulting in a performance deficit.

Versioning-based optimistic locking solutions [17, 34, 36] stem from identifying read-heavy workloads. In this concurrency control strategy, writer-writer conflicts are solved by the conventional use of locks. Reader-writer conflicts are solved through versioning:

- each item has a version number;
- there are "clean" and "dirty" version representations (e.g. odd numbers represent clean versions; even represent dirty);
- readers observe the version of an item before reading its contents;
- if at any point a reader detects a version change or observes a dirty version, it retries the operation; and
- writers have to atomically increase the version of an item before starting and after finishing each modification.

Version-based optimistic locking negates the need for reader-reader synchronization, minimizing synchronization and improving performance. As having a version number for each item can be spatially expensive, some solutions apply the methodology to sets of items instead [17]. This optimization trades space efficiency for marginally worse performance, as false positives can occur when a reader notices a version change caused by a modification of a different item of the same set.

Finally, there are partitioning-based solutions [34, 32], which completely eliminate conflicts, therefore also negating the need for concurrency control. Partitioning solutions divide the data space by the number of cores (or a number higher than the number of cores), meaning that each request can only be served by one core. By doing so, these solutions increase performance as they are not susceptible to synchronization overheads. These solutions are optimal when the workload distribution is uniform, as each core will process approximately the same number of requests at any given time. A problem arises

with real workloads, as they are inherently skewed, meaning that some items are much more likely to be requested than others. This imbalance in access item frequency means that the cores that have to serve popular items will become "hot", and have to do most of the work, while other cores become idle. MICA's authors [34] argue that the trade-off of partitioning is beneficial since cores that are not "hot" will take on other work, such as processing network packets, and because "hot" cores will benefit from having fewer cache misses resulting from data locality. Having more partitions than the number of cores may be favourable as a way of distributing popular items and reducing the impact of "hot" cores.

3.1.4 Memory Allocation

In a KVS system, memory allocation is the component responsible for managing the storage of items in memory. Real workloads have variable value sizes, meaning that the system should support this variability while maintaining space efficiency. One of the common problems of memory allocation is *memory fragmentation*, where the memory management of objects of variable size becomes expensive and/or is not space efficient.

A common approach is the *Slab Allocator* [6], which separates memory into a certain number of slab classes, where each slab class contains slabs (memory regions) of fixed size. The slab allocator approach is simple, cheap to manage and good at minimizing fragmentation [6]. The problem with the slab approach is that if some item sizes are more frequent than others, there will always be slabs that are constantly full and slabs that are empty [34], resulting in low space efficiency. Some solutions try to dynamically change each slab class's size, which helps with space efficiency but requires additional overhead.

Another approach is to have a log-like structure, such that new items are appended at the end of the log and old items are garbage collected. MICA's circular log goes a step further and enables the log to wrap around, allowing for a lighter garbage collector by taking advantage of cache semantics [34].

3.1.5 Key Takeaways

There is a vast design space for KVSs. There are four main components that must be cleverly combined to form a fast and efficient KVS: the Network Stack, the Indexing and Eviction policy, the Concurrency Control Mechanism and the Memory Allocation component.

The Network Stack is tasked with managing user requests as well as communicating the results back to the user. A Network Stack of a KVS can either use the OS's socket abstraction or have direct NIC access. It is the Indexing component's duty to maintain information on which items are in the KVS and where they are stored in memory. The Eviction Policy should work closely with the Indexing component to enable the eviction of items when necessary. The Concurrency Control mechanism enables multiple processors to work simultaneously, helping the KVS provide a higher throughput and lower latency.

Finally, the Memory Allocator maintains the values of key-value items; it must efficiently manage values of variable size. Each component must complement the others in a balanced manner so that one component does not bottleneck the entire system.

Table 3.1 summarizes the design choices of each of the KVS mentioned in this section. It shows that there is a tendency for Optimistic Locking mechanisms, for Read-oriented indexing and for solutions to target commodity hardware.

Table 3.1: Properties of each Key-Value Store.

	Memcached [42]	Masstree [36]	MemcachedGPU [26]	MemC3 [17]	MICA [34] (EREW/CREW)
Concurrency Control	Coarse and Fine-Grained Locking	Optimistic (Fine-Grained) Locking	Timestamp for CPU-GPU ordering	Optimistic Locking	Partitioning/ Optimistic Locking
Indexing Data Structure	Hash Table (with Linked-Lists)	B+- tree	Set-associative Hash Table	Cuckoo Hash Table	Lossy Concurrent Hash Indexes
Read-oriented or Write-friendly	Write-friendly	Read-oriented	Read-oriented	Read-oriented	Write-friendly
Focus on Space Efficiency	No	No	No	Yes	Yes
Commodity Hardware	Yes	Yes	No(GPU)	Yes	Yes
Batching	Yes(Reads)	No	Yes(Server-Side)	Yes(Client-Side)	No
Network Stack	Kernel	Kernel	Direct NIC Access	Kernel	Direct NIC Access
Approximate Eviction Policy	No	n/a	No	Yes	Yes
Embedded Eviction Policy	No	n/a	No	Yes	No
Persistent Storage	No	Yes	No	No	No
Exploit Locality	No	Yes	No	Yes	Yes
Maintained? (Last commit)	jun. 2023 [42]	sep. 2023 [37]	sep. 2017 [43]	jun. 2014 [41]	jan. 2016 [45]

3.2 Non-Blocking Concurrency

Non-blocking concurrency encompasses optimistic concurrency control mechanisms that make use of atomic primitives, such as CAS.

Most Non-Blocking algorithms that utilize the CAS atomic primitive are subject to the ABA problem. The ABA problem manifests itself when a variable has the value A and it is concurrently changed to B and then back to A'. A process can not distinguish between A and A', which means it might perform a successful CAS and cause data structure inconsistencies. There are few ways to deal with the ABA problem directly, most of which utilize a VERSION COUNTER to help distinguish between A and A'.

Non-blocking algorithms must be accompanied by a Safe Memory Reclamation scheme so that memory can be reused. Memory Reclamation schemes may not be automatic, meaning that they require the Non-Blocking algorithm to invoke extra operations, and they might not be fault tolerant, meaning that process crashes will stop memory from being reclaimed. In some contexts, Memory Reclamation solutions solve ABA as a side-effect,

eliminating the need to have mechanisms that deal with both Memory Reclamation and the ABA problem.

In addition to addressing the ABA problem and Memory Reclamation, Non-Blocking algorithms are highly complex as the interaction of multiple processes without mutual exclusion leads to a very large space of possibilities. Designing a Non-Blocking algorithm that does not allow for unsafe interleavings through simple and restrictive atomic primitives is a challenge in itself.

3.2.1 Non-Blocking Algorithms

A common structure of Non-Blocking algorithms is the Single Compare-and-swap Universal (SCU) [2]. An algorithm in the SCU class is divided into a *preamble* and a *scan-and-validate* phase. In the preamble, the algorithm prepares the changes that will be done to an object. In the scan-and-validate phase, the algorithm tries to serialize its changes by performing a CAS operation. If the CAS operation succeeds, the procedure terminates, if it fails the procedure must be retried.

Many of the operations of Non-Blocking algorithms may perform more than one CAS. In those cases, it is usually possible to identify a primary and a secondary CAS. The secondary CAS's purpose is usually to fix intermediary state that previous conflicts might have caused. For example, Michael's Queue [48] allows interleavings that cause the Tail pointer of the queue to point to the second to last node, instead of the last node. When processes detect that the Tail does not point to the last node, they try to change the Tail by performing an extra CAS operation.

Each operation of the Non-Blocking algorithm must take into account possible interactions with other operations. For example, the insertion of a new node into a singly linked list must take into account other processes concurrently traversing the list: Imagine the list $A \rightarrow B$, in which we are trying to insert C. If we change the next field (the field that points to the next node) in node A first, then the list would temporarily look like $A \rightarrow C$, meaning that a concurrent traversal might try to access the next field of C and erroneously detect the end of the list.

Finally, Non-Blocking algorithms must take performance into account. Non-blocking concurrency takes an optimistic approach to concurrency, meaning that conflicts are resolved instead of prevented. The resolution of a conflict can have considerable overhead. An algorithm that does not take into account this overhead might not scale well with an increasing degree of contention, where conflicts are more likely. One approach is to minimize the work required in the preamble phase [18], so that in the case of a conflict further retries are not as expensive.

3.2.2 Solving ABA

There are multiple ways in which ABA can be solved. Some are too restrictive, others severely impact performance.

It is important to note that solutions to the ABA (and Memory Reclamation) Problem should be lock-free. If an algorithm uses an ABA solution with progress guarantees weaker than lock-freedom, then the algorithm itself can not be classified as lock-free.

One solution is to guarantee that the values that are subject to being CAS'ed are unique. This way ABA is impossible because there can only be one instance of A as all values are unique. Therefore, if some process T_1 compares a variable to see if it still holds A and that comparison succeeds, it is guaranteed that A has not been modified since T_1 first read it because any modification would forbid A from ever "re-appearing" after said modification.

Ensuring the uniqueness of all values that are subject to being CAS'ed presents many obvious disadvantages: ensuring that all values are unique might not be a feasible restriction; if the algorithm requires memory references to be CAS'ed, it has to guarantee that all references are unique, i.e., no memory location is reused, which severely hinders the memory footprint of a program.

Utilizing version counters [25, 54] (also called tags) is a different way to ensure that values are unique: each shared object has a version counter associated with it; when writing to that object, its counter is incremented. With this solution every time an object is written the combination of object and version counter changes so that the two instances (the object before and after the write) are distinguishable from each other, and therefore not ABA prone. The write operation and version counter increment have to be done atomically. There is the possibility that a process increments the counter enough times so that it overflows and reverts back to its first instance, meaning that values are not TRULY unique, the key is to have counters big enough so that it is extremely unlikely for this scenario to occur.

A possible implementation of the described version counters would be to utilize the atomic Double-Compare-and-Swap (DCAS) operation. The DCAS is similar to CAS except it takes two memory locations (that may or may not be contiguous) and performs two CAS atomically. To implement version counters with DCAS, one would use it to modify both the object and version counter atomically, making sure that each write increments the object's version counter. DCAS is not available on any modern multi-processor system [11, 12], making this solution not usable in practice.

A different implementation of version counters is to limit the addressing space so that a single memory location can contain both a reference and a version counter. With this method, the CAS could be used to atomically change both values, similarly to the method that uses DCAS. This solution is very limiting, as reducing the addressing space might not be feasible. Furthermore, this solution impacts different architectures disproportionately as some architectures have a larger addressing space and/or maximum word size.

3.2.2.1 Solving Memory Reclamation

The choice of Memory Reclamation scheme is an important one as different Memory Reclamation solutions have different costs, meaning that the Non-Blocking algorithm's performance is directly related to the Memory Reclamation solution the algorithm uses. Since performance is usually the main motivation behind Non-Blocking concurrency, a Memory Reclamation Scheme that bottlenecks the performance improvements of Non-Blocking algorithms is unusable in practice.

Traversals account for a considerable portion of a data structure's operations, therefore having unsynchronized traversals can significantly improve performance [3]. Memory Reclamation schemes typically require *per-element* fences, *per-operation* fences, or both. A per-element fence is usually done each time a new shared reference is read. An example of a per-element fence is the need to increment a reference count each time a new reference is read (in the Reference Counting solution [13]). A per-operation fence is done per data structure operation, e.g. announcing an epoch for each operation (in EBR [18]).

Naturally, in most algorithms, a per-element fence is more costly than a per-operation fence. This is because per-element fences are executed much more frequently than per-operation fences. Traversal length influences the performance of different Memory Reclamation schemes differently — algorithms that require per-operation fences allowed for better performance and scalability, when compared to algorithms that require per-element fences [23].

Object Pools. A way of solving Memory Reclamation by never actually deallocating memory [8]. For this solution to be feasible, it has to allow retired records to be re-used, so that memory footprint is reduced. Object Pools suffer from the disadvantage that once memory is used it can never be reclaimed, rendering it unusable in many applications.

Reference Counting. Records are accompanied by a (reference) counter that represents how many pointers point to the record. When a process reads a new Record it atomically increments its reference counter, when it will no longer use the record it atomically decrements the counter. A record will be freed once its reference counter reaches zero.

Reference Counting has many downsides:

- It requires additional memory per record;
- It prohibits records whose pointers form a cycle, as their reference count will never drop to zero; and
- It has high overhead since it requires per-element costly atomic operations.

Reference Counting has proven to be less performant than other schemes [23, 46].

Hazard Pointers. Processes explicitly declare what shared data structure references they are currently utilizing so that the memory locations they hold to are not freed [47, 46].

Hazard Pointers are essentially wrappers to shared references that symbolize to other processes that the references inside a Hazard Pointer are protected and can not be reclaimed. Each process keeps a list of its Hazard Pointers (the list size depends on the algorithm, and may or may not be of constant size); every time a process needs to access a shared reference it first declares that it will use that reference by putting it inside one of its Hazard Pointers.

When a process wants to reclaim a retired node N, it first scans all other processes' Hazard Pointers to see if they hold a reference to N. If there are no Hazard Pointers that point to N, the retired record N is safe to be reclaimed; if there are Hazard Pointers pointing to N, then its reclamation must be postponed. Because the scanning of every Hazard Pointer is costly, processes only execute the Scan procedure when they have at least R records to reclaim (R can be arbitrary, but is recommended to be $R = H + \Omega(H)$, H being the number of Hazard Pointers per process, Ω being the asymptotic lower bound [46]).

Algorithm 2 Hazard Pointer example in Non-Blocking List.

```
1:  function NonBlockingList
2:      ...
3:      retry:
4:      a ← Head                ▷ Store shared reference to Head of list in private variable
5:      *hp ← a                 ▷ Store private variable in Hazard Pointer
6:      if a ≠ Head then        ▷ Check if a still contains Head
7:          goto retry
8:      end if
9:      ...
10: end function
```

Algorithm 2 presents a simple example of the use of a Hazard Pointer. Lines 6 and 7 might not be intuitive; they serve as a way to detect conflicts: Imagine that thread T_1 is executing the code in algorithm 2, it gets to line 4 and suspends. Then thread T_2 wants to remove the head of the list, checks if there is any Hazard Pointer pointing to it, and removes it as there is none. Thread T_1 resumes execution, and executes line 5, protecting Head in a Hazard Pointer. If thread T_1 does not execute the check on line 6, then it will access a reference that might of been reclaimed, which is unsafe behaviour.

The key observation is that Head in line 4 was unprotected and could be reclaimed. After line 5, Head is protected, so we just need to make sure that it hasn't been modified during the transition from unprotected to protected.

Epoch Based Reclamation. EBR [18] is a reclamation method that works by coordinating processes in epochs, and only freeing records associated with a certain epoch such that it is impossible that a stale reference to those records exists.

In EBR each epoch has an associated *limbo bag*. Processes insert retired records (see Section 2.2.4) in the limbo bag of epoch ε (the "current" epoch). As processes execute operations they advance the epoch. Records in the limbo bag of epoch $\varepsilon - 2$ (the epoch that was the "current" epoch two epochs ago) can be safely reclaimed.

A process is in a *quiescent* (dormant) state whenever it does not hold a reference to an object in the shared data structure. To simplify, we consider that a process stops being quiescent if it starts a data structure operation, and resumes being quiescent when it finishes said operation (assuming it does not hold references to objects in the data structure between operations).

A grace period is a period of time in which every process has been in a quiescent state at least once [8], i.e., an interval in which no process was continually performing a data structure operation. Since processes do not hold references to the shared data structure's records while in a quiescent state, records that were retired before the last grace period are not accessible by traversing the data structure and can be safely reclaimed.

To identify grace periods, processes manipulate a global counter, that represents the current epoch, and an announce array. Before starting any operation processes observe the current epoch ε, announce it and check if all other processes have also announced ε (processes only announce any given epoch once). A process that detects that an epoch was announced by every process tries to atomically increment the current epoch.

Records in limbo bag of epoch ε are not safe to reclaim, as processes that are traversing the data structure can still hold local references to those records because they may not have been in a quiescent state. To understand why records in epoch's $\varepsilon - 1$ limbo bag cannot be reclaimed, there is this illustrative example:

- There are two processes, T_1 and T_2;
- T_1 starts an operation and announces ε. It does not try to advance the epoch as T_2 has not yet announced ε;
- T_1 holds a private reference to ε's limbo bag, as it plans on adding a retired record R to it;
- T_2 starts an operation, announces ε and subsequently increments the current epoch;
- T_1 has an outdated private reference, as it thinks that $\varepsilon - 1$ is the current epoch. Because of this, it inserts R into $\varepsilon - 1$, and not ε.

The described execution demonstrates how a process can insert a record into limbo bag $\varepsilon - 1$, meaning that objects in that limbo bag can not be safely reclaimed. The same can not happen for epoch's $\varepsilon - 2$ limbo bag as other processes would need to advance the epoch without the delayed process having announced it. Therefore, it is safe to reclaim all records in limbo bag $\varepsilon - 2$. In fact, "the period of time starting from when the epoch was changed from $\varepsilon - 2$ to $\varepsilon - 1$ until it was changed from $\varepsilon - 1$ to ε is a grace period" [8],

meaning that a grace period has occurred since epoch $\varepsilon - 2$ and for that reason its records can be reclaimed as there can no longer exist private references to records in its limbo bag.

Quiescent State Based Reclamation. QSBR [40] is a generalization of EBR [19], that identifies grace periods by processes explicitly declaring when they leave and enter a quiescent state. This way, if a process observes that all processes have been in a quiescent state at least once since the last grace period, then it knows that another grace period has occurred and previously retired records can be reclaimed. A key difference between QSBR and EBR is that EBR assumes that processes are in a quiescent state between data structure operations (i.e. they do not hold references to objects in the data structure), whereas QSBR does not. Instead, QSBR allows processes to hold references to data structure objects as long as they do not become quiescent.

The main problem with EBR and QSBR is that they are not fault tolerant, since a slow or crashed process implies that grace periods may be impossible to identify and the number of unreclaimed records can grow to infinity [18, 8]. In EBR, if any process crashes no records can be reclaimed as all processes must announce an epoch for the epochs to advance. If epochs don't advance then no grace periods are detected and no records are reclaimed. In QSBR, process crashes only prohibit records from being reclaimed if a process crashes while in a non-quiescent state, as a process that crashes while in a quiescent state still allows other processes to identify grace periods correctly. This means that, although QSBR is not fault tolerant, it is "partially fault tolerant" (coined by [8]) as a crashed process *may or may not* mean the amount of unreclaimed memory grows to infinity. Other distinctions between QSBR and EBR are that: QSBR is application-dependent in the sense that it is up to the application to manually detect quiescent states, while EBR is application-independendent [24]; and that QSBR presents less overhead than EBR, allowing for better performance [24]. The benefits of EBR and QSBR is that they are much more efficient than other methods since they only require overhead per operation.

Distributed Epoch Based Reclamation. DEBRA [8] works by identifying if a process was in a quiescent state in two ways that resemble EBR and QSBR directly. The two ways that DEBRA identifies that a process was in a quiescent state since the last grace period is if that process announced the current epoch, as in EBR; and through per process quiescent flags, as in QSBR. Through these two grace period identification methods, DEBRA combines advantages from both reclamation schemes: QSBR's low overhead and EBR's application independence. DEBRA hides QSBR's need for explicit declaration of quiescent states through the assumption that no shared data structure reference is held in between data structure operations (as in EBR). DEBRA minimizes EBR's overhead of announcing and incrementing epochs by only *partially* reading announcements every time a data structure operation is performed as well as only announcing the current epoch every x operations. Additionally, DEBRA also inherits QSBR's "partial fault tolerance"

in the sense that if a process crashes while in a quiescent state, then the system can still reclaim records.

DEBRA also includes other performance optimizations. Firstly, DEBRA minimizes the overhead that comes with manipulating shared limbo bags by having three private limbo bags per process instead. This way there is no synchronization required when modifying limbo bags. Secondly, DEBRA avoids freeing records as much as possible, as allocation and deallocation has significant overhead. It tries to reuse records instead, having processes keep local *object pools*, as well as shared object pools for when local pools become full. These object pools are managed in blocks, to reduce overhead.

DEBRA+ [8] goes a step further and adds fault tolerance. DEBRA can not reclaim records if a slow or crashed process is stuck in a non-quiescent state. DEBRA+ takes advantage of the naturally fault tolerant design of non-blocking data structures [8], and utilizes OS *signaling* so that processes execute a *signal handler* to try and repair the data structure after a process has crashed. This way DEBRA+ can recover from process crashes, continue advancing the epochs and reclaiming memory.

3.2.3 Contention Management

The presence of contention may severely impact the performance of a concurrent algorithm. For example, in the case of Blocking algorithms that use locks, contention implies that some process will remain idle for a period of time, diminishing the total useful CPU time and incurring a performance deficit.

For Non-Blocking algorithms contention results in significant performance decay, even though CAS-based algorithms that are designed to do the least possible work to resolve conflicts perform well under high contention [18].

A CAS-based algorithm suffers from contention because when multiple processes try to apply changes to the same shared variable only one of the processes will succeed. When shared variables manipulated through CAS become "hot spots" the interconnect and memory devices become congested, slowing down successful CAS operations, thus hindering performance [14].

Applying a simple contention management algorithm, such as Exponential Backoff, can improve the performance of algorithms that utilize the CAS primitive under medium and high contention while presenting small overhead in the case of low contention [14]. Simpler contention management algorithms outperformed more complex algorithms as they present less overhead. These algorithms contain parameters that can be tuned *per architecture*, as opposed to *per implementation*.

3.3 Summary

In this Chapter, we discussed the components of a cache application and the problems inherent to any non-blocking system.

We saw that, in a cache application, each of its components aims to solve a specific problem present in any cache application:

- The network stack establishes how network communication is managed (by the cache server);
- The indexing and eviction policy provide fast data retrieval and manage the items that are most likely to be of use to the cache, respectively;
- The concurrency control establishes how concurrent processes synchronize with each other to guarantee correctness in accesses to the cache's shared data structures; and
- The memory allocation component, that ensures that memory fragmentation is low so that the cache can efficiently utilize its memory resources.

Furthermore, we examined multiple ways to solve the ABA and memory reclamation problem and analysed a simple way to manage contention in non-blocking contexts. In particular, we concentrated on the DEBRA memory reclamation scheme, including its inspirations: EBR and QSBR, due to its flexibility and low overhead.

4

FLEEC Design

In this Chapter, we recapitulate the problem statement and its requirements, analyse Memcached under varying workloads and go into detail regarding the design choices of FLEEC.

4.1 Objectives and requirements

In-memory caches, and more specifically in-memory key-value caches, are systems whose purpose is to leverage faster (and smaller) storage mediums to lower application run times by minimizing the amount of time said applications need to wait for data stored externally.

In the context of in-memory key-value caches, the "in-memory" portion refers to the fact that data is stored in main memory (i.e., Random Access Memory (RAM)) so that accessing that data is done in a timely manner. The "key-value" portion refers to the simple data model being used: entries contain only a key and a value; meaning that the system need not maintain information regarding relationships between data entries. A simple data storage paradigm means that processing requests is also simple, allowing for higher performance boundaries as the work the system has to carry out is considerably diminished.

Due to the purposefully simplistic nature of an in-memory cache, the objectives and requirements of such a system are also very straightforward. A cache aims to provide high throughput, low latency and high hit ratio. High throughput means that the system can respond to many requests per time unit, resulting in a system that can sustain a much higher load. Low latency means that the amount of time an application has to wait for the cache's reply is reduced, which directly results in lower latency for the application's clients. A high hit ratio means that a high number of requests can be successfully fulfilled by the cache, minimizing the amount of times the application needs to query the slow persistent storage system, which in turn improves said application's performance.

In-memory key-value caches are typically scaled through a scale-out strategy, which means that a caching layer deployment is scaled by increasing the number of servers of the cache service. This results in the distribution of requests, reducing the amount

of work each server performs while increasing the total computational power that that deployment possesses. This strategy is optimal under uniform workloads as, on average, each server receives the same amount of requests the other servers do, which limits the probability of some servers becoming saturated and bottlenecking the overall system's performance. The problem is that, in reality, most workloads are skewed [5], i.e., a few number of items receive many requests while the remaining items are rarely or never accessed. The inherent skewness of most workloads results in an uneven distribution of client requests, which in turn means that some caching servers will experience a much higher load than others, and are more likely to become saturated. This is true only if the data each server stores is not replicated, as replication would mean that any given request could be served by more than one server. Since replication is accompanied by significant overhead as servers have to spend processing and networking resources to communicate and synchronize, most caching solutions do not replicate their data. So, because most workloads are highly skewed and because any given client request can only be served by a single server, it is important to improve the performance of each individual caching server, so that an application's caching layer can provide better overall performance and be more resilient to the saturation of a single server's resources.

Our objective is to improve the performance of a caching system, namely Memcached [42], by enhancing individual caching servers. For this reason, we ignore the fact that caching systems are usually deployed as multiple caching servers and only consider scenarios where multiple clients send requests to a single server.

4.2 Profiling

The optimization of any system should start with a detailed analysis of how it functions. In the realm of performance optimization, the first step should be to *profile* the system so that improvements can be targeted at the system's most costly portions. In this Section, we present the profiling results of Memcached as well as a discussion of said results.

Each entry in the profiling results presented contains a percentage, a time measure, a function name and an observation. The percentage field represents the portion of total (accumulated) time each function took to execute during the entire program execution, excluding sub-functions (functions called from within). In other words, the percentage field denotes the total percentage of accumulated *exclusive CPU time*. The time measure represents the total absolute time that was spent on each function (also excluding sub-functions). The function name simply represents the Memcached function sampled by the profiler tool. The observation field aims to clarify the function's purpose. For example, the function *lock_wait_private* is an internal locking function involved with Memcached's locking mechanism, and for this reason, is accompanied by the observation *Internal lock function*.

Workloads Profiled. There are two main workload variables that alter Memcached's performance: the workload's access distribution; and the average item size. We utilized both synthetic and real workloads to profile Memcached. All profiled workloads follow a zipfian distribution, as it is the most common access distribution observed in in-memory caching workloads [31].

The access distribution will influence the amount of concurrent operations that conflict with each other. The aspect of an access distribution that most interests us is its skewness. Skewness is represented by a zipfian α value; the higher the α value, the higher the skewness. The more skewed a distribution is, the more it is concentrated on a fewer number of items, while a distribution with very low skewness is similar to a uniform distribution. Therefore, the higher the skewness the higher the probability that two client requests conflict with each other. Conversely, the opposite holds true as well, as a minimally skewed access distribution will have a lower number of conflicts, on average.

Item size influences the amount of requests that can be performed per time unit and therefore limits the system's maximum throughput. A higher item size decreases the amount of requests processed per time unit, which in turn also minimizes the probability of conflicts occurring.

There were three distinct workloads used to profile Memcached:

1. A synthetic workload with a minimally skewed access distribution;
2. A synthetic workload with a highly skewed access distribution; and
3. A real workload, taken from twitter's traces (cluster 1) [31].

The characteristics of each workload can be seen in Table 4.1. Workloads 1 and 2 differ only in skewness. Their purpose is to help us evaluate how different degrees of contention impact Memcached. Variations of workload 1 and 2 were used to identify Memcached's performance differences under workloads with varying network-related demands. Workload 3 was used to compare how synthetic and real workloads differ.

Table 4.1: Properties of Workloads used for Profiling.

	Key Space Size	**Key Size**	**Value Size**	**Zipfian α**	**#requests per client**	**R/W ratio (%)**
Low Skewness (1)	20 000	4	1	0.40	100000	99.00
High Skewness (2)	20 000	4	1	2.40	100000	99.00
Real Trace (3)	48 341	58	362	1.53	501657	99.50

Testing Scenario. The tests were carried out in the cluster [49] of the Department of Computer Science of the NOVA School of Science and Technology (DI-FCT-NOVA). The specification of the machines used are depicted in Table 4.2.

Table 4.2: Specifications of the test machines.

CPU	2 x Intel Xeon Gold 6346
#cores / #threads	32/64
L1/L2/L3 cache	1 MiB/40 MiB/72 MiB
DRAM	128 GiB DDR4 3200 MHz
NIC	2 x 10 Gbps Ethernet

Our tests were evaluated with five machines: one server machine — running an instance Memcached[1] — and four client machines communicating with the server over the network. Each client machine simulated 150 individual clients, for a total of 600 clients. We chose to run 600 clients since that was enough to saturate the server's resources[2].

Client machines use `libmemcached` [33] to communicate with the server over TCP. To reduce network overhead clients batch GET requests in groups of 100 [17, 36]. We do not batch SET requests, as neither Memcached nor `libmemcached` support it. GET request batching improves performance by minimizing network overhead, but it is not always realistic to use in practice. For this reason, we also evaluate how client-side GET request batching affects Memcached so that the benefits of batching could be examined.

Memcached has a feature to save item data to external storage so that server instances can have more storage capacity. We disabled this feature in all the tests since accesses to disk negatively impact performance.

Profiler Tool. Our study of Memcached was done through the use of the *gprofng* tool [20], due to its ability to profile multithreaded applications written in C. It is important to note that *gprofng*'s profiling results include the time spent executing an internal sampling function. The sampling function is used to gather profiling results, it is not part of Memcached. Because of this, we removed this sampling function from the profiling results presented, in order to facilitate the results' comprehension.

4.2.1 Contention Impact

To measure the impact that high contention had on Memcached, we profiled Memcached under two workloads: one with a low skewness access distribution; and another with high skewness access distribution. Under the low skewness workload, two requests are much less likely to conflict with each other, when compared with the high skewness workload. This stems from the fact that a high skewness distribution is more heavily concentrated on a fewer number of items, which means that a higher number of requests will try to perform operations on the same hash table bucket, hence more collisions will occur. Put simply, the more skewed a distribution is, the higher the contention levels Memcached

[1] The profiling evaluation was performed on Memcached version 1.16.19.

[2] Because the server is running a "profiling version" of Memcached, the server utilizes more resources than usual.

experiences throughout its execution. Note that every workload shown in this Section included client-side batching to lower network costs.

The results of profiling Memcached under the low skew workload can be seen in Table 4.3. It is evident that Memcached spends most of its time either performing network operations (the *sendmsg* and *read* functions) or handling blocking concurrency mechanisms (the *pthread_mutex_unlock*, *pthread_mutex_lock*, *lock_wait_private* and *lock_wake_private* functions). Network and locking-related operations take ≈ 20.67% and ≈ 68.64% of the time, respectively. Combined, network and locking operations take ≈ 89.31% of the total execution time.

Table 4.3: Profiling of Memcached under Low Skewness workload (1).

Time Spent (%)	**Absolute Time (s)**	**Function Name**	**Observation**
24.620	11.718	pthread_mutex_unlock	Unlock function
19.390	9.236	sendmsg	Send data to client
18.070	8.606	pthread_mutex_lock	Lock function
14.280	6.795	lock_wait_private	Internal lock function
11.650	5.554	lock_wake_private	Internal lock function
2.234	1.071	mcount	Profiler function
2.075	0.981	resp_start	Process requests
1.276	0.600	read	Receive message from client
0.518	0.250	MurmurHash3_x86_32	Hash Function
0.518	0.240	_transmit_pre	Process requests
0.478	0.230	tokenize_command	Process requests
0.478	0.220	item_get	Find item function locking
0.438	0.200	resp_finish	Process requests
0.399	0.190	assoc_find	Find item in hash table
0.399	0.190	process_get_command	Process requests
0.279	0.140	do_item_get	Find item function logic
0.279	0.130	??	Unknown library function
0.239	0.120	epoll_wait	Poll network event
0.199	0.100	item_remove	Remove item from data structures
0.199	0.090	drive_machine	Main loop
1.955	0.923	Other	N/A

The results of profiling Memcached under the highly skewed workload can be seen in Table 4.4. It is clear that, under a workload with high skewness, Memcached spends a considerable amount of time performing locking operations. Network operations take just ≈ 1.51% of the total execution time while locking-related operations occupy a striking ≈ 96.11% of the execution time.

By observing the low and high skewness workload profiling results in Tables 4.3 and 4.4, respectively, we can understand how higher contention affects Memcached. Firstly, note that Memcached under the low skewness workload provided 3.3× more throughput and less latency when compared to Memcached under the high skewness workload. Secondly, under the low skewness workload, Memcached already spent a considerable amount of time performing concurrency control operations (≈ 68.64%). The fact that Memcached spends a considerable amount of time executing lock operations indicates

Table 4.4: Profiling of Memcached under a High Skew workload (2).

Time Spent (%)	Absolute Time (s)	Function Name	Observation
46.060	771.550	lock_wait_private	Internal lock function
45.610	764.104	lock_wake_private	Internal lock function
2.655	44.491	pthread_mutex_lock	Lock function
1.774	29.741	pthread_mutex_unlock	Unlock function
1.141	19.103	sendmsg	Send data to client
0.372	6.334	mcount	Profiler function
0.372	6.224	read	Receive message from client
0.316	5.254	limited_get	Wrapper for item_get
0.180	2.992	process_get_command	Process requests
0.169	2.812	resp_start	Process requests
0.146	2.452	assoc_find	Find item in hash table
0.113	1.931	item_remove	Remove item from data structures
0.101	1.621	resp_finish	Process requests
0.079	1.341	_transmit_pre	Process requests
0.067	1.221	item_get	Find item function locking
0.067	1.221	item_is_flushed	Check if item's TTL is over
0.067	1.071	MurmurHash3_x86_32	Hash Function
0.056	0.991	do_item_get	Find item function logic
0.056	0.941	epoll_wait	Poll network event
0.056	0.901	__strncmp_sse42	Compare item keys
0.520	8.602	Other	N/A

that a non-blocking approach could have a significant positive impact on performance. Under the minimally skewed workload, the performance increase upper bound for a Memcached with optimized concurrency control is $\approx 3.19\times$[3]. Thirdly, under the highly skewed workload, Memcached suffered tremendous performance degradation. Under the highly skewed workload, the performance increase upper bound for a Memcached with optimized concurrency control is $\approx 25.7\times$. The increase in time spent performing lock operations is explained by the higher contention. Not only does Memcached spend a significant amount of time performing blocking concurrency control operations under minimally skewed workloads, the amount of time spent increases dramatically under highly skewed workloads.

The most noteworthy change when comparing the profiling results under the low and high skew workloads is the increased time Memcached spent in the internal lock functions (the *lock_wait_private* and *lock_wake_private* functions), when under the high skew workload. In the *lock_wait_private* function, processes are continuously alternating between trying to acquire access to the lock and waiting. In the *lock_wake_private* function, processes perform a *syscall* to awaken other processes that may be in a waiting state (for them to subsequently try to acquire the lock). The total amount of time spent in these two functions is ≈ 12 seconds in the low skew workload (Table 4.3), and ≈ 1535 seconds in the high skew workload (Table 4.4). The significant increase in time spent performing these

[3]We get this number by unrealistically assuming that it is possible to completely eliminate the cost of concurrency control.

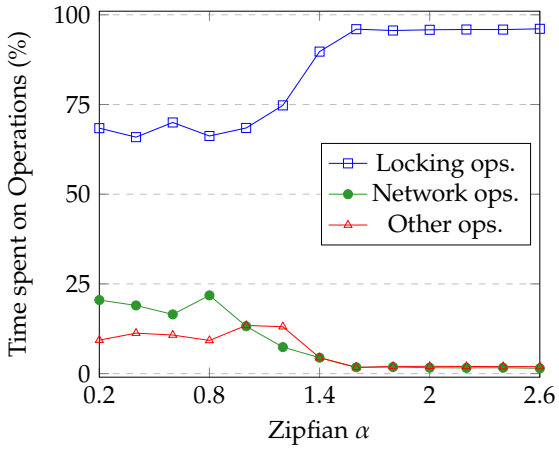


Figure 4.1: Time spent on Locking, Network and other operations, varying Zipfian α value.

internal lock operations indicates that there is a high degree of contention happening on accesses to locks when the workload has high skewness.

Moreover, through Figure 4.1 we can detect that the Memcached starts to severely suffer from contention at a zipfian α value of ≈ 1.2. When zipfian α values are lower than 1.2, while execution time is still dominated by locking-related operations, other operations still significantly impact execution time. At zipfian α values greater than 1.6, operations not related to locking have practically no impact whatsoever.

These observations lead to the conclusion that optimizations to Memcached's components not related to its concurrency control will have a limited impact, even more so under high contention scenarios.

Due to the analysis presented in this Section, it is plausible that a non-blocking in-memory application cache could help alleviate some of the performance problems present in Memcached.

Real Workload. The results presented up to this point were collected through the synthetic workloads. Synthetic workloads are useful because they are much easier to manipulate in order to gather helpful information. That said, because assumptions need to be made to produce synthetic workloads, it is desirable to acquire some degree of confirmation that real workloads do not significantly deviate from synthetic workloads.

Table 4.5 presents the profiling results gathered from profiling Memcached under a real workload (twitter's cluster 1 trace [31]). This workload has a zipfian α value of 0.22 (seen in Table 4.1), which is similar to the low skew synthetic workload zipfian α value of 0.4. The major difference between the real and the low skew synthetic workload is their item sizes: the real workload has items 84× larger than the synthetic workload's.

The real workload spends 73.32% of the time performing lock operations, while the low skew synthetic workload spent 68.64%. On the other hand, the real workload caused a 13.64% of the execution time being spent on network operations, compared to the low skew synthetic workload's 20.67%. These results are surprising because we would

expect the real workload to have spent less time on locking operations (due to the real workload's slightly lower zipfian α value that would create less contention) and more time on network operations (because of the real workload's larger items). Nevertheless, the real and synthetic workload profiling results are approximately similar.

These profiling results show how synthetic workloads can be used to estimate how well a cache server performs under real workloads, while at the same time, fail to capture the entire complexity of the data at hand.

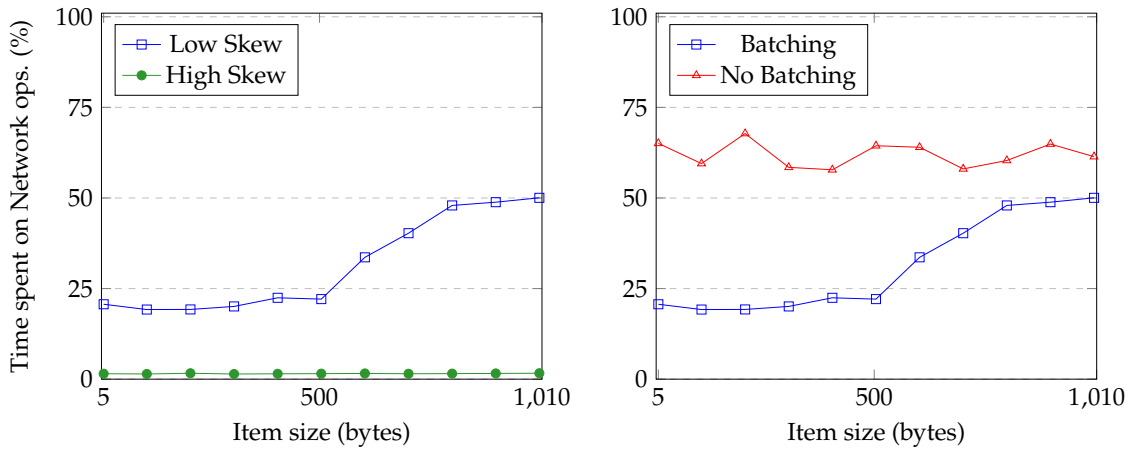
Table 4.5: Profiling of Memcached under real workload (3).

Time Spent (%)	Absolute Time (s)	Function Name	Observation
24.110	25.098	lock_wait_private	Internal lock function
21.810	22.696	lock_wake_private	Internal lock function
15.220	15.851	pthread_mutex_lock	Lock function
12.180	12.669	pthread_mutex_unlock	Unlock function
9.346	9.717	sendmsg	Send data to client
4.289	4.463	read	Receive message from client
3.096	3.212	tokenize_command	Process requests
1.875	1.961	MurmurHash3_x86_32	Hash Function
1.363	1.421	mcount	Profiler function
1.079	1.121	resp_start	Process requests
1.022	1.051	process_get_command	Process requests
0.454	0.480	_transmit_pre	Process requests
0.369	0.380	resp_finish	Process requests
0.340	0.350	epoll_wait	Poll network event
0.255	0.260	assoc_find	Find item in hash table
0.227	0.240	do_item_get	Find item function logic
0.227	0.240	item_get	Find item function locking
0.198	0.200	item_remove	Remove item from data structures
0.170	0.180	??	Unknown library function
0.142	0.140	__strncmp_sse42	Compare item keys
2.230	2.340	Other	N/A

4.2.2 Network Cost

The other potential restraint that Memcached may face is the network. Simply put, network-related operations consume a significant amount of computing resources. Additionally, a Memcached server instance is limited by its network bandwidth. These issues are difficult to solve by modifying Memcached directly. A solution to a network-related bottleneck would involve a hardware upgrade or the optimization of network operations (e.g. utilization of user space network operations, as opposed to kernel space network operations).

Figure 4.2a shows the relative amount of time the execution of network operations takes both under low and high skewness workloads while varying the size of cached items. It is evident that, in the case of the low skew workload (1), the relative amount of time network operations take starts increasing at around the 500 byte mark. This means that starting from an average item size of 500 bytes, increases in item size mean that



(a) Comparison of time spent performing network operations on Low and High Skew workloads (with batching).

(b) Comparison of time spent performing network operations on Batched and Non-Batched (Low Skew) workloads.

Figure 4.2: Comparison of time spent performing network operations.

optimizations to Memcached will have diminishing returns. Note that, under different scenarios (e.g. an execution of Memcached that does not include profiling), the point at which the system starts to spend more time on network operations (500 bytes, in the scenario presented) is likely to suffer a significant offset. On the other hand, the high skew workload (workload 2, page 33) spends the same amount of time performing network operations no matter the item size. This is due to the fact that Memcached under the high skew workload is bottlenecked by contention, not by the network, which means that it takes a higher network demand for the network to start to bottleneck Memcached's performance. Because under the high skew workload, Memcached is bottlenecked by contention, in high contention and high network demand scenarios, concurrency control optimizations can still have a significant positive impact.

Effects of Batching. Client-side batching means that clients join multiple GET requests into a single request. This way, less packets are sent and network overhead is minimized.

The use of batching is not always practical in a real scenario, since batching increases latency and application complexity. For this reason, we evaluate the performance differences between batched and non-batched workloads.

Figure 4.2b presents the relative amount of time network operations take with and without batching. To limit the negative impact of contention, the low skewness workload (workload 1, page 33) was used to gather these results. Non-batched requests mean that network operations take ≈ 60% of Memcached's total execution time, while batched requests can require anywhere from 20% to 50% of the total execution time. Batching requests can significantly diminish the amount of time spent performing network operations. However, the benefits of batching diminish as the item size increases.

Thread 1		Thread 2
Acquires hash table lock	1	
	2	Tries to acquire hash table lock
Acquires LRU lock	3	
Inserts x into hash table	4	
Inserts x into LRU	5	
Releases LRU lock	6	
Releases hash table lock	7	
	8	Acquires hash table lock
	9	Acquires LRU lock
	10	Removes x from hash table
	11	Removes x from LRU
	12	Releases LRU lock
	13	Releases hash table lock

Figure 4.3: Correct interleaving on separate blocking hash table and LRU.

4.3 FLEEC

FLEEC proposes a re-design of Memcached [42] with new non-blocking aware data structures and new non-blocking management operations.

4.3.1 Non-Blocking Aware Data Structures

When considering non-blocking concurrency, synchronization between processes takes place using atomic primitives, such as CAS, that serializes writes to a small portion of shared memory (usually these primitives atomically modify 4 or 8 bytes, depending on the architecture). Because non-blocking algorithms serialize writes through atomic operations and not through mutual exclusion, it is very difficult to make non-blocking operations on separate data structures seem atomic to an outside observer, thus avoiding undesirable interleavings that leave the data structures in inconsistent states.

Intuitively, making two blocking operations to two separate data structures seem atomic is trivial because, by acquiring the corresponding lock, it is easy to enforce that no more than one thread can access both data structures at the same time. For example, consider the interleaving in Figure 4.3, inspired by Memcached's behaviour, that represents concurrent operations on a blocking hash table and a LRU: *thread 1* performs an insert and *thread 2* a remove. We can see that no incorrect high-level interleaving occurs because *thread 2* can not interfere with *thread 1's* insertion, as it has to wait for *thread 1* to release both locks. This way, the insertion and deletion operation on the hash table and LRU execute atomically w.r.t. outside observers, meaning that all threads will observe that any given item can only be in one of two states: either the item is in both the hash table and the LRU, or it is in neither.

Inversely, the same can not be said for non-blocking operations, as there is no synchronization between two separate non-blocking data structure operations. This means that high-level interleavings can leave inconsistent state. For instance, consider the interleaving

Thread 1		Thread 2
Inserts x into hash table	1	
	2	Removes x from hash table
	3	Removes x from LRU (not found)
Inserts x into LRU	4	

Figure 4.4: Incorrect interleaving on separate non-blocking hash table and LRU.

presented in Figure 4.4, which represents the same concurrent operations applied to a non-blocking hash table and a non-blocking LRU. This interleaving shows how concurrent operations on separate data structures can leave the data structures in inconsistent states: in this case, the LRU would contain item "x" whereas the hash table would not. This difficulty dramatically increases the complexity of designing and validating a correct and performant algorithm that works in these circumstances.

The goal of Memcached's doubly linked list LRU was to implement a *least recently used eviction policy*. To avoid inconsistencies between the LRU and the hash table, we opted to completely eliminate the LRU and use the hash table to implement an *approximated least recently used eviction policy*. Keeping only one core data structure means that serialization through the CAS atomic primitive on that data structure is enough to guarantee that concurrent operations do not leave an inconsistent state.

4.3.2 Hash Table with CLOCK Eviction Policy

To implement an eviction policy without making use of an LRU linked list, we adapted CLOCK [9], a well-known Operating System page replacement algorithm that approximates an LRU eviction policy with an affordable cost. The idea to embed CLOCK into Memcached's hash table was inspired by MemC3's [17] similar approximation, although our adaptations of the algorithm made our approach diverge from MemC3.

Both the original CLOCK algorithm and MemC3 use a single bit per item in the cache. Items with their bit set (value 1) were recently used. Items with their bit cleared (value 0) were not recently used. When we need to find a victim to be evicted from the cache, we loop through every item in the cache. If we observe an item with its bit cleared, it is selected as the eviction victim; if we observe an item with its bit set, we clear its bit and proceed to the next item.

Starting from Memcached [42] as a baseline, one approach could be to apply CLOCK directly, i.e., have one bit associated with every item in every collision list[4]. However, this approach would require the eviction procedure to traverse each hash table bucket sequentially until a victim was found. An issue with this approach is that hash buckets are implemented as linked lists, which are expensive to traverse as they exhibit low cache locality (as discussed in Chapter 2). This means that traversals will likely have to fetch data from main memory, negatively impacting the performance of the eviction procedure. Another downside of this approach is that it would increase the amount of item meta-data

[4]This is the approach taken by MemC3 [17].

(that has to be stored in its slab), which would leave less space for actual useful data (this drawback would have more impact on smaller items).

The approach used by FL^{EE}C is to associate the CLOCK's bit to each hash bucket, instead of each item. Since the hash table expansion mechanism ensures that, on average, there will be 1.5 items in each bucket, we know that each bucket's CLOCK bit will represent, on average, the recency of 1.5 items. The CLOCK bits associated with each bucket are contiguously located in memory, allowing for a cache-aware search and a more efficient victim selection. The CLOCK bit now represents the recency of the most recently used items in the corresponding bucket. When this bit is found cleared, it means no item in the bucket was accessed recently and we can evict the entire bucket instead of just one item at a time. This approach not only reduces the amount of meta-data required but also means CLOCK meta-data can be stored separately from items, leaving more space in each item slab. Furthermore, we do not limit the CLOCK to just one bit, but rather a counter variable (of n bytes). In this way, we are able to more accurately differentiate very popular from mildly popular buckets.

As CLOCK only approximates a LRU eviction policy, we can consider it a more "relaxed" algorithm in the sense that it does not need to select the *true least recently* bucket as an eviction victim. So, given that we are only approximating a LRU behaviour, we can choose not to synchronize modifications to CLOCK variables. Opting for no synchronization allows read-write and write-write concurrency hazards, which may result in information loss. This loss of information is tolerable, as long as its negative impact in the hit ratio is minimal. As we are trying to maximize performance, we opted to omit synchronization when updating the CLOCK variables (and avoid the corresponding synchronization overhead). If for some reason the absence of synchronization yields a lower hit ratio than is desired, it is trivial to synchronize writes to these variables, although that would increase the overhead of maintaining the eviction policy.

4.3.3 Making FL^{EE}C Non-Blocking

Getting rid of the LRU linked list and expanding Memcached's hash table with a CLOCK-based eviction policy is the first step in transforming Memcached's concurrency control non-blocking. It is also a self-contained step, which we can evaluate on its own. We call *MemCLOCK* to the version of Memcached that uses lock-based concurrency control and has a CLOCK-based eviction policy (instead of the LRU).

4.3.4 Non-Blocking Linked List

To transform Memcached's hash table into a non-blocking hash table we substituted each hash table bucket, which was originally implemented as a simple singly linked list accessed under lock-based mutual exclusion, by Harris' pragmatic non-blocking singly linked list [22]. Harris' singly linked list algorithm requires no synchronization at all for list traversals (they exhibit *wait-free* progress). Most cache workloads are extremely

read intensive [5] and we can leverage the synchronization-free traversals with *wait-free* progress to implement fast never-stalling cache-read operations. With these lists, mutual exclusion to control accesses to the hash table is no longer required, and any number of concurrent writes and reads can take place at the same hash table bucket[5].

In the new non-blocking hash table, searching for an eviction victim follows the CLOCK algorithm presented previously: keep decrementing CLOCK counters in a round-robin fashion until we find a counter with its value set to zero. When we find a victim, we can evict (delete) from the cache all the items in that bucket. In Harris' algorithm, deletions maintain correctness under concurrent operations by first logically deleting the items through pointer tagging. Items can only be physically deleted, i.e., made inaccessible through traversing the linked list, after they are logically deleted. Thus, when evicting a bucket, we first mark all its items as logically deleted and then physically remove them all at the same time.

Logically removing items by pointer tagging is done through a CAS operation. If the CAS operation succeeds, then the item was tagged as removed. If the CAS operation fails due to a conflicting concurrent write, then we know that some other process was either inserting or deleting an item in that same bucket. In this case, the eviction of this bucket is aborted so that recently inserted items are not removed, and the algorithm is resumed to select a new victim.

FLEEC's hash buckets, implemented as non-blocking linked lists, respect set semantics (i.e. do not contain duplicate items). If FLEEC's hash table buckets were not sets, then they might have duplicate items, i.e., items with the same key. If duplicate items exist, only the most recent one is considered. This means that outdated duplicate items are not contributing to the cache's functioning, are wasting memory slots and slow down read operations because they have to be traversed through (in the linked-list search). Furthermore, if we allow duplicate items to be in the same list, there is additional overhead in guaranteeing that no "backwards reads" are possible. A backwards read happens when some item A_{old} was replaced by A_{new}, but the same thread first reads A_{new} and then reads A_{old}, which through a sequential lens of events appears as if the system is regressing and thus ruins the semantics of a cache[6]. For example, consider an example where a backwards read occurs in a non-blocking linked list where duplicate items exist, shown in Figure 4.5: *Thread 1* first reads A_{new} and, because of a concurrent delete operation performed by *Thread 2*, subsequently reads an outdated version of item A (A_{old}). In this context, one way to guarantee that a backwards read is not possible is by deleting all occurrences of an item every time we wish to delete that item, always starting from the oldest occurrence of that item. This delete operation has to perform much more work than a delete operation that does not have to consider backwards reads, which makes it complex and non-performant. Since not respecting set semantics results in slower read

[5] Although conflicting writes may have to be retried, as in Harris' linked list algorithm.

[6] More accurately, by backwards reads we are referring to a sequence of read operations that invalidate the causal consistency properties of the system, i.e., a sequence that fails to ensure the monotonic reads property.

Thread 1		Thread 2
//A_new and A_old are	1	
// in the list	2	
// (in that order).	3	
Reads A_new	4	
	5	Removes A_new
	6	//A_old remains in the list
Reads A_old	7	
	8	Removes A_old

Figure 4.5: Interleaving where a backwards read happens on a non-blocking linked list.

and delete operations, FLEEC's hash buckets respect set semantics. Additionally, FLEEC's hash buckets are sorted lists so that search operations do not have to traverse the entire list and write operations do not have to check every single item in the list before inserting (to guarantee set semantics).

Progress Conditions. FLEEC's progress conditions are tightly coupled with its hash bucket non-blocking linked list's progress conditions, as most operations can be fulfilled through the hash table alone.

Firstly, every linked list operation first calls an internal search operation to traverse the list. This means that every linked list operation must have weaker or equivalent progress conditions when compared to the search operation. Although the search operation's main purpose is to traverse each list item in a wait-free manner, the search operation is also tasked with removing any items that have been logically but not physically deleted. Due to this secondary task, the search operation might need to perform a CAS operation to remove leftover items, which means it only has lock-free progress guarantees. That being said, even under moderate to high contention, the search operation is more likely to exhibit wait-free rather than lock-free progress.

Secondly, all other hash table operations guarantee lock-free progress. The *get* operation is essentially a search operation, i.e., it has lock-free progress (but is wait-free in practice). Both the *set* and *delete* operations have to perform a search and at least one CAS operation. Since the CAS operation might fail (and the operations retried), it is guaranteed that at least one thread makes progress, both the set and delete have lock-free progress.

It is important to note that FLEEC's slab allocator is still protected by a global lock. This means that write procedures that **might** have to access the slab allocator (e.g., the procedure that handles SET requests) do not present non-blocking progress guarantees in their entirety[7]. The slab allocator has to be accessed every time FLEEC needs to allocate or deallocate memory, i.e., upon the creation of new items and the eviction of old ones. In the case that the procedure does not have to access the slab allocator, it has non-blocking progress guarantees. In the case that the procedure has to access the slab allocator, it

[7]The SET and DELETE operations always have to access the slab allocator (for allocation and deallocation, respectively). The ADD operation does not create a new item if an item with the same key is already present, meaning that it might not have to access the slab allocator.

presents the same progress guarantees as Memcached, i.e., it has deadlock and starvation-free progress guarantees.

4.3.5 Memory Reclamation

Since FLEEC's hash table buckets are non-blocking linked lists, re-utilizing an item's memory, either by eviction or by explicit deletion from a client, requires a memory reclamation scheme. FLEEC uses a memory reclamation scheme based on DEBRA [8] (described in Section 3.2.2.1), due to its performance and flexibility. DEBRA is essentially a combination of EBR and QSBR, with some performance optimizations. FLEEC's memory reclamation scheme (we will refer to it as FleeC Reclamation Scheme (FRS) for simplicity) closely resembles DEBRA, as it also allows grace periods to be detected through epoch announcement or explicit quiescent state declaration. The main difference when comparing FRS to DEBRA is that it separates these grace period detection mechanisms into two different procedures. This separation aims to make FLEEC more flexible, by allowing it to choose which of these mechanisms is more appropriate in each scenario.

DEBRA always has a process' quiescent flag unset while it performs a data structure operation, signifying that that process is not in a quiescent state (as it may hold references to shared data structure objects). This is so that if other processes observe set quiescent flags (processes in quiescent states), the memory reclamation scheme can progress. Additionally, processes also check what the current epoch is and announce it if they have not done so yet. Modifying a process' quiescent flag or announcement variable has little overhead as it only requires a total of two non-synchronized writes per operation[8]. There are two main problems with this approach. Firstly, in Non-Uniform Memory Access (NUMA) systems, writes to announcement/quiescent variables may invalidate caches in other sockets, leading to last-level cache misses and non-optimized reads [19]. DEBRA circumvents this issue by reading the announcement array in increments, i.e., each time a process performs a data structure operation it reads only a portion of the announcement/quiescent variables to distribute the weight of cache misses. Secondly, always announcing the current epoch not only increases the probability of the described cache misses but also means that epochs will advance at a high rate (and the corresponding memory reclaimed) even when no additional memory is required. Furthermore, each epoch advancement requires at least one (possibly more) expensive CAS operation. DEBRA minimizes the rate at which epochs advance by having processes announce epochs every x data structure operations, instead of every operation. The disadvantage of minimizing the rate at which epochs advance is the delay in the reclamation of some items.

DEBRA is meant to be a general memory reclamation scheme, easily applicable to a variety of non-blocking algorithms. For this reason DEBRA does not assume that the overlying non-blocking algorithm knows when the system requires memory to be

[8] As the announcement and quiescent variables can share the same address, exchanging space efficiency for a small increase in the amount of instructions performed.

reclaimed. This is not a reasonable assumption for most systems, while it is essential in a caching system, as any caching system must know when it is full so that it can evict some of its items. For this reason, FRS adapts DEBRA to its context: since we know exactly when we need to reclaim memory, processes only perform the expensive procedure of trying to advance the epoch (and reclaim retired items) when FLEEC is actively trying to evict items. When processes are not trying to evict items, they simply update their quiescent and announcement variables so that other processes are able to reclaim memory but restrain from trying to advance epochs themselves. This way we only advance epochs when we actually need to do so, minimizing the amount of cache misses and the number of CAS operations performed. This procedure segregation minimizes the total amount of work the system needs to perform for its memory reclamation as the memory reclamation scheme will only progress when it needs to. Additionally, this also means that, while processing a client's GET operation, the overhead of performing memory reclamation will be minimal as a GET operation will never provoke an eviction. Since most workloads are read heavy [5], the common path of most workloads does not have to perform memory reclamation work. This approach is correct as it is similar to DEBRA's approximation of only advancing epochs every x data structure operations, only changing the policy to "we only advance epochs when we need to".

Similarly to DEBRA, FRS also has private epoch bags instead of shared epoch bags, to limit the amount of synchronization required.

As FRS is based on DEBRA, it does not provide fault tolerance, meaning that if a process crashes while in a non-quiescent state, no memory can be reclaimed from that point on. We claim that the absence of fault tolerance was worse in the original lock-based approach: if a process dies when holding a lock, most probably the system will eventually block. Nevertheless, fault tolerance could be supported by following the philosophy of DEBRA+ [8]. The addition of fault tolerance to FRS is left as future work.

Preemptive Eviction. Write operations that cause evictions need to go through several phases: search for eviction victims; remove the chosen victim items from their hash bucket list; advance 2 memory reclamation scheme epochs (which might take a long time, depending on the state of other threads); and finally release the items' memory to the slab allocator. The problem is that the memory reclamation phase may cause threads to continually fail in their attempts to advance the memory reclamation scheme, which might reduce performance. A practical solution would be to preemptively evict items so that FLEEC always has some amount of available memory at any given time. This way, when threads need to acquire a slab for a new item, threads are much less likely to have to go through the full eviction process as it has been performed preemptively. A possible way to implement preemptive eviction in FLEEC would be to have a background thread that looks for "old" items (i.e., items that have been inserted in the cache more than x seconds ago) in buckets with CLOCK values set to 0. The background thread would then remove those items and pass them through the eviction process. Having a background

thread would not cause significant overhead as it would only be woken when memory utilization surpasses a certain threshold. Additionally, the background thread would also look for items with expired TTL, so that expired items do not remain in the cache for long periods of time.

4.3.6 Non-Blocking Hash Table expansion

As described in Section 2.1.1, hash bucket traversal is expensive as it is likely to have to fetch data from main memory. To get around this issue, Memcached and FLEEC implement a hash table expansion mechanism so that the amount of items in each hash bucket is reduced, effectively minimizing the negative impact of the cache unfriendly traversal.

Although FLEEC's hash table expansion algorithm is similar to Memcached's, there are some key differences. Firstly, Memcached's expansion algorithm starts with a stop-the-world-like phase. This is acceptable for Memcached as it allows for a less complex algorithm that does not have a significant negative impact on performance. A stop-the-world phase is not acceptable for FLEEC, as it would mean that all hash table operations could not be considered non-blocking (and would lose their progress guarantees). Secondly, FLEEC's expansion has to deal with the nature of non-blocking concurrency control. Not having access to mutual exclusivity increases the amount of undesired interleavings that must be prevented.

The hash table expansion works by having two simultaneous hash tables, where the new hash table has twice the number of buckets of the old one. The first half of the new hash table points to the buckets of the old hash table. When the number of hash table buckets doubles, approximately one half of the items are no longer in the bucket their hash value corresponds to, meaning that they must be re-inserted into the correct bucket. To this end, there is a thread (the *hash table expansion thread*) that is in charge of moving items to their correct bucket during expansion. The *hash table expansion thread* moves items to their correct bucket by re-calculating each item's hash value and subsequently removing and re-inserting them into the correct bucket if required. Since the expansion thread will be accessing and modifying the hash table, it must take part in the memory reclamation scheme previously described. When the hash table is not expanding, the expansion thread is idle and *must* be in a quiescent state, so that it does not prevent the memory reclamation scheme from progressing and memory from being reclaimed.

At the start of an expansion, it is possible that threads are not aware that an expansion is happening. For example, consider the interleaving in Figure 4.6, where an expansion starts right after a thread starts an operation, leading it to perform the operation as if an expansion is not occurring. The fact that a thread might erroneously detect that an expansion is not happening can be detrimental if the thread is performing an insertion operation. In this case, the thread might insert an item in the incorrect bucket. If, by chance, the expansion thread has already corrected the bucket in which the item is inserted, the item will stay in the incorrect bucket indefinitely. This would mean that threads that

Thread 1		Expansion Thread
Starts hash table operation	1	
Checks if expansion is happening	2	
//Stalls	3	
	4	Starts hash hable expansion
Continues hash table operation	5	

Figure 4.6: Interleaving where a thread does not detect that an expansion is happening.

search for that item in the correct bucket will not find it, i.e. the (incorrect) insertion was effectively useless[9]. We prevent such incorrect insertions from happening by delaying the expansion thread. After commencing an expansion, the expansion thread will wait exactly two epochs, as that represents a grace period and means that, from that point on, all other threads will know that a hash table expansion is happening. This way, threads can only insert items into the wrong bucket before the expansion thread starts its item re-insertion, meaning that all incorrect insertions will eventually be corrected. As it is possible for items to be in an incorrect bucket during expansion, threads that are aware of an expansion have to search both possible hash table buckets, since the item they are looking for might not have yet been moved to the correct bucket. This is only necessary in the case of search or delete operations.

The presented hash table expansion procedure has some downfalls. Since moving items requires removing them from a hash bucket, it is possible that a false miss might occur for a concurrent read on an item that has been removed but not yet re-inserted[10]. Furthermore, when deleting an item to re-insert it elsewhere, since non-blocking linked list deletion requires first logically removing items, it is possible that another process observes the logically deleted item and physically removes it, reclaiming its memory, before it can be re-inserted. This can lead to items being wrongly deleted, which although allowed by cache semantics, is undesired behaviour. We allow these types of behaviour to occur since, although harmful, it is unlikely and does not significantly detriment FLEEC's behaviour.

Finally, when expansion ends, the old hash table can be reclaimed through the same memory reclamation mechanisms, the only difference being that we are releasing the memory directly to the OS and not the slab structure.

Each time a hash expansion happens, it is exponentially more difficult to trigger the next expansion (as the number of items required to trigger the next expansion doubles). This means that, in a long enough execution (where the number of items in the cache will tend to stabilize due to the memory restrictions imposed), there will be no hash table expansions happening for long periods of time. Since cache servers are systems that tend

[9] Although harmful, as the item will use up memory and have the be traversed through, this situation does not compromise correctness.

[10] The obvious approach of inserting the item in the new location before removing from the old one is non-trivial, as the item would be temporarily double-linked (from both hash buckets) and this might cause memory reclamation problems.

to run for long periods of time without restarting, the cost of hash table expansions is negligible. To add to this, since there is no strict upper bound on the time it takes to perform an expansion, the thread responsible for the expansion can choose to back off if it detects that it is conflicting with other (normal) threads. For these reasons, the hash table expansion mechanism is satisfactory in terms of performance if it does not cause the system to become unstable during its execution.

An additional detail regarding hash table expansion is how to track the number of items that are currently in the hash table. To this end, Memcached made use of a simple counter variable protected by a lock. FL^{EE}C cannot use this approach as the use of a lock would limit its strong progress guarantees. Another possible solution would involve modifications to the counter to be made atomically (e.g., with the fetch-and-add atomic primitive), which requires the execution of expensive atomic instructions. Both these solutions require unnecessary synchronization. FL^{EE}C's solution is to have one counter variable per thread instead of one global counter, thus eliminating the need for synchronization as each thread only writes to its counter variable. When checking if an expansion should be triggered (this check happens once every second), the sum of every counter variable gives us an approximation of the total number of items in the hash table. This method is only an approximation as it is possible that some interleavings give incorrect results. For example, consider the case where thread T_1 inserted an item but stalled before incrementing its counter variable; then, thread T_2 removed that item and decremented its counter variable. At this moment, the sum of all counter variables would yield a result of -1, which is obviously incorrect as there can't be a negative amount of items in the hash table. Still, we argue that because we are only checking if the number of items is *approximately* 1.5 times the number of buckets, incorrect results do not lead to incorrect behaviour. Through this solution FL^{EE}C eliminates the use of locks (and synchronization in general) and maintains its strong progress guarantees.

4.3.7 Non-Blocking Replacement algorithm

In the context of key-value caches, an update operation is the modification of an item's value. In other words, an update operation changes a key-value pair to a pair with the same key but a different value.

In FL^{EE}C, updates stem from SET operations. A SET operation creates a new item if no item with the specified key exists; if an item with the same key is already cached, the SET operation updates that item's value instead[11].

FL^{EE}C's updates are processed by removing the item with the "old" value (the *replacee*) and substituting it with an item with the "new" value (the *replacer*). The replacee and replacer are independent from each other, their only shared characteristic being their matching keys.

[11] In contrast, ADD operations only create a new item if no item with the same key exists. If the key exists, an ADD operation *touches* an item instead, i.e., it updates their recency in respect to the eviction policy, without changing its value.

Thread 1		Thread 2
//Replacing x_old with x_new	1	//Looking for item x
Removes x_old from hash table	2	
	3	Lookup for item x
	4	//Lookup unsuccessful
Inserts x_new into hash table	5	

Figure 4.7: Interleaving where a thread does find an item being concurrently replaced.

We call a *false miss* the observation of the intermediate state between removal and insertion (where no item with the given key is present in the data structure). An example of a false miss occurring in FLEEC's hash table can be seen in Figure 4.7, where a thread looking for an item (line 3) does not find it due to a concurrent replace operation. False misses are undesirable as they result in a miss that could have otherwise succeeded under a slightly different interleaving, as the resources to satisfy the GET request were present in the cache. Our experiments (discussed in Section 6.2.7) reveal that, under highly skewed workloads, false misses alone can lower FLEEC's hit ratio from $\approx 100\%$ to $\approx 60\%$. Since false misses can significantly diminish a cache's hit ratio, limiting its effectiveness, we designed a replacement algorithm to decrease the probability of false misses occurring in FLEEC. It is important to note that, although harmful, false misses do not violate cache semantics as caches give no guarantees in relation to the data they store.

Under mutual exclusive accesses, it is trivial to guarantee that no false misses occur: simply acquire mutual exclusive access for each of the data structures where the item is present, remove the replacee and insert the replacer. Since no outside observer can access the data structures while we are replacing the item, we can ensure that no intermediate state is exposed. In contexts without mutual exclusivity, a specialized solution is required in order to prevent false misses.

Avoiding false misses. FLEEC's non-blocking hash table is the only data structure involved with item replacement. Furthermore, since both the replacee and replacer have the same key, we know that the replacement operation takes place in the same hash bucket. For this reason, the replacement operation can be seen as an expansion of Harris' non-blocking singly linked lists [22].

A seemingly trivial solution to this problem is to invert the order of the replacement's sub-operations: instead of a removal followed by an insertion, we insert before we remove. This order inversion seems to fix the problem because, at any given time, there is at least one item in the data structure, thus no false miss can occur. With this solution in mind, there are two options: either the replacer is inserted *before* or *after* the replacee. We refer to these options as *anterior* and *posterior* insertion replacement, respectively. An issue raised by this solution is that, without further care, set semantics can be violated, which may present significant negative side-effects, as discussed in Section 4.3.4.

Another possible solution is to expand the the logical removal of items into two categories: a normal logical deletion, in which the item is simply considered to be absent from the list; and an *in replacement* logical deletion, in which the item is considered to be under a replacement procedure, signaling other threads to "be patient" in an effort to avoid false misses. This solution does not infringe on the non-blocking linked list's set semantics.

Anterior Insertion Replacement. Lookups stop when an item with the searched key is found. This means that an insertion of the replacer before the replacee makes the latter obsolete, as the replacee will never be reached if the replacer is present. Thus, an *anterior insertion replacement* is serialized as soon as the replacer is inserted into the list.

The problem with this solution is that it allows backwards reads to occur, as can be seen in the interleaving shown in Figure 4.8. In this example, Thread 2 interferes with Thread 1's replacement and concurrently removes the replacee, while leaving the replacer in the list. This interference allows Thread 3 to observe x_{new} (the replacer) before it observed x_{old} (the replacee), i.e., it observed the system's state regress.

A possible solution that prevents backwards reads is for delete operations to delete all the items with the specified key (from oldest to newest), instead of the first. This way, concurrent delete operations would delete items in their chronological order, while read operations read the most recent item, thus backwards reads would not be possible. For this solution to work, all removals would have to be done in reverse order.

A considerable disadvantage of this solution is that singly linked list traversal is unidirectional, which means that, to remove multiple items from a list in reverse order requires the re-traversal of the list at least once for each item (as opposed to a normal traversal where the lower bound of required traversals is 1, as we can "remove as we go"). There is a low expected cost in a common deletion scenario, since if we only want to delete one item, both a normal and backwards deletion would have to traverse the list once[12]. The problem is that, if doing backwards removals, when FL^{EE}C evicts entire buckets it would have to perform extra costly traversals when compared to normal deletions. For this reason, backwards removals may significantly impact eviction performance due to the additional required work deletions have to perform when removing multiple items. This means that anterior insertion replacement can decrease eviction performance, limiting its use in practice.

Posterior Insertion Replacement. The other possibility would be to insert the replacer after the replacee so that the order in which the items appear in the list respects their chronological order. Immediately after the insertion of the replacer, subsequent searches will still reach the replacee first, as if no replacement occurred. It is only after the replacee is removed that the *posterior insertion replacement* is serialized, as future searches will find the replacee from that point onward.

[12]Although the normal deletion would have to perform less traversal steps, on average.

Thread 1		Thread 2		Thread 3
//Replacing x_old	1	//Deleting x	1	//Searching for x twice
//with x_new	2		2	
Inserts x_new	3		3	
	4		4	Search for x
	5		5	//Returns x_new
	6	Deletes x (x_new)	6	
	7		7	Search for x
	8		8	//Returns x_old
Removes x_old	9		9	

Figure 4.8: Interleaving where insertion before removal (inserting the new item before the old one) leads to backwards read.

The posterior insertion replacement algorithm is very simple: traverse the list until we find the place in which items with the given key should be in; the replacee might not be in the list due to a concurrent removal, in which case we simply try inserting the replacer; if the replacee is found, we keep traversing the list until we find a different key, and then insert the replacer; finally, we invoke a delete operation that will delete the replacee (or an item with the same key added by other threads concurrently performing replacements). An advantage of this solution is that it is self-contained as it does not require further modifications to the linked list algorithm.

The posterior insertion replacement has lock-free progress condition guarantees. The lock-free progress stems from the fact that if the CAS operation trying to perform the insertion succeeds, then the thread performing replacement progresses; if it fails, some other thread will make progress. Furthermore, the last step of this replacement solution is to invoke a delete operation, which in turn has lock-free guarantees, which means that the replacement method could not have stronger progress guarantees than the delete operation.

In-Replacement Logical Deletion Replacement. In order to respect set semantics, it is possible to mark item references that signal to other processes that an item is in the process of being replaced. This way, threads can retry their search operation until the replacement operation is complete, in an effort to avoid false misses.

The algorithm in itself is rather simple. We start by searching for a replacee by key (i.e., the first non-marked item with the specified key), and marking it as *in-replacement*; then, we attempt to insert the replacer in the spot immediately before the replacee; if the insertion CAS operation fails, we re-traverse the list in search of the replacee, aborting the replacement operation if some other thread concurrently inserted an item with the same key (to avoid violating set semantics); if the CAS operation succeeds, the next step is to simply logically and physically delete the replacee.

A replacement operation is considered as failed if the replacer was not inserted into the list. If the operation fails after insertion, the list is left in a transitional state, which will eventually be resolved by threads performing concurrent replacement or future search

operations. It is up to the caller of the operation to decide if a failed replacement operation should be retried. In the case of FL^{EE}C, failed replacement operations are retried, in order to guarantee that clients' write requests induce some change in FL^{EE}C's state.

There is one other required modification to the original algorithm. During search operations, if the item we are looking for is detected to be in-replacement, the list is re-traversed until that item is no longer present. A problem with this approach is that if the thread performing the replacement stalls or crashes after marking an item, threads concurrently searching for that item will be indefinitely stuck in a livelock-like loop. Obviously, livelocked threads are a huge detriment to any system. To avoid this scenario, there is a maximum amount of retries that a search operation can perform. If a thread reaches the maximum number of retries, it is tasked with removing the in-replacement item, to avoid further time-consuming re-traversals and search retrials. This way, livelock scenarios are prevented and progress is guaranteed. A side-effect of the prevention of livelocks is that it is possible that false misses still occur, although extremely unlikely. For this reason, we say that this replacement strategy is probabilistic as does not completely avoid false misses, but suppresses them with high probability.

The in-replacement logical deletion replacement works because, at any given time, at least one item with the specified key is present in the list. The combination of items that may be present in the list are as follows:

1. the unmarked replacee;
2. the replacee, marked as in-replacement;
3. both the replacer and the marked replacee;
4. only the replacer.

State 1 and 4 correspond to the phase immediately before and after the replacement operation, respectively. The observation of state 1, 3 or 4 does not require threads performing search operations to do any additional work. In the event that a thread observes state 2, it retries its search operation until it either observes state 3 or 4. The described procedure ensures that, after the replacee is marked as in-replacement, all subsequent search operations will be aware that a replacement is taking place and will try to avoid undesirable false misses.

Similarly to the other replacement algorithms, in-replacement logical deletion replacement never produces state where neither the replacee nor the replacer is in the list. The main difference is that the other replacement strategies require a temporary violation of set semantics, while in-replacement logical deletion replacement utilizes the already existing logical deletion mechanism to implement a false miss-avoidant algorithm.

By retrying search operations threads are performing redundant work in an effort to maintain a high hit ratio. It is not obvious that the exchange of computational cycles for a

higher hit ratio is a worthwhile trade. To convince the reader of the benefits of this trade, we evaluate the in-replacement logical deletion algorithm in Sections 5.1 and 6.2.7.

The in-replacement logical deletion replacement algorithm has lock-free progress conditions because it guarantees system-wide progress. The modified search operation **without** a maximum number of retries could not be classified as non-blocking as a *in-replacement mark* would act as lock to the marked item, granting the replacement thread exclusive access to that item. The modified search operation **with** a maximum number of retries is still lock-free (and wait-free in practice, as a search operation is guaranteed to finish in a bounded number of steps if no logically deleted items are found).

There is an additional small detail regarding the modified search operation. Each search retry is counted on a per-operation level, i.e., it is possible that during a search operation, the retries are caused by different in-replacement items. This means that it is possible for a in-replacement item to be deleted right after it was marked as in-replacement. Avoiding these types of interleavings would not only require extra bookkeeping work but also mean that the search operation would only ensure obstruction-free progress. For this reason, and because these interleavings should be extremely improbable, we have chosen to allow the described behaviour.

Key Takeaways. We have devised three possible replacement algorithms. The anterior replacement algorithm is the inferior choice, as it does not respect set semantics and requires possibly expensive delete operations that the other algorithms avoid. The posterior replacement algorithm is self-contained but temporarily violates set semantics. The in-replacement logical deletion replacement algorithm respects set semantics but may require concurrent searches to retry and still allows false misses to occur (although unlikely).

While it is difficult to devise an unequivocally correct algorithm, there is an increased degree of certainty that comes from empirical evidence, regarding the correctness of an algorithm. Harris' non-blocking linked list, although not formally proven, has been studied and used extensively. For this reason, a replacement algorithm that respects Harris' non-blocking linked list set semantics is more likely to avoid unknown harmful interleavings than an algorithm that does not respect set semantics. On this ground, the in-replacement logical deletion replacement might be the preferable choice. That being said, in Section 6.2.7, we conduct empirical performance evaluation to aid in the choice of replacement algorithm.

4.3.8 Contention Management

FL[EE]C was designed to maintain high performance under high and medium contention scenarios. FL[EE]C's main strategy to deal with contention is its non-blocking concurrency mechanisms. Although non-blocking concurrency control strategies handle contention

more effectively than blocking strategies [18], non-blocking algorithms' performance is still hindered by the presence of contention.

Backoff strategies can help minimize non-blocking algorithms' degradation caused by contention [14]. A backoff strategy applied to non-blocking algorithms operates by having threads sleep after failed CAS operations so that threads that conflict are "desynchronized" [13] and, thus, have a lower chance of conflicting again in the near future. The most promising backoff strategy is the exponential backoff [14] as it does not require much additional work to be performed.

The exponential backoff contention management algorithm requires per-hardware architecture parameterization, limiting its practical application. Moreover, the exponential backoff might hinder performance, as the performance improvements are dependent on many factors, namely the exponential backoff parameters, the non-blocking algorithm itself and the hardware architecture. For example, a case where the non-blocking algorithm might limit the contention management's effectiveness is FLEEC's hash table. FLEEC's hash table non-blocking linked list operations have to retry their operation after a failed CAS operation. When operations are retried, the re-traversal of the list mimics a backoff mechanism as it adds a degree of randomness to the operation's execution. This randomness might be sufficient to minimize the probability of further conflicts, meaning that the backoff mechanism would just increase the amount of work performed and hinder performance.

Fundamentally, the utilization of a contention management mechanism is dependent on many variables. For this reason, we evaluate the possible performance benefits that arise from employing an exponential backoff contention management strategy in Section 5.2. With the help of our empirical evaluation, we can observe if FLEEC does or does not benefit from contention management techniques.

4.3.9 Summary

FLEEC is designed to be a caching system with a high degree of performance, especially under high contention. FLEEC has an approximated LRU eviction policy in the form of a variation of the CLOCK algorithm; a non-blocking concurrency control strategy in the form of non-blocking linked lists; an epoch based, flexible memory reclamation scheme; a non-blocking hash table expansion mechanism; and a replacement algorithm that minimizes false misses.

[13] In this context, desynchronized means that some noise is added to threads' execution of their instructions so that the timing at which they execute CAS operations is less likely to cause further conflicts.

5

Advanced Details

In this Chapter, we present our dedicated non-blocking replacement algorithms, evaluate them and assess if an exponential backoff contention management could be a helpful addition to FIˢᵉᵉC.

5.1 The Replacement Algorithms

In this Section, we conduct an in-depth analysis of the non-blocking replacement algorithms (presented in Section 4.3.7) designed to avoid false misses that can occur in the absence of mutual exclusion. We start by examining the *in-replacement logical deletion replacement* algorithm, followed by the *posterior insertion replacement* algorithm and then present a self-contained experimental comparison between the two algorithms.

5.1.1 In-Replacement Logical Deletion Replacement.

The main idea of the in-replacement logical deletion algorithm is to utilize the pointer tagging mechanism present in Harris' non-blocking linked-list [22] to mark items as "in-replacement". Threads that search for an item and find that it is "in-replacement", re-traverse the list until they observe the new item (or reach a maximum number of retrials) in an effort to avoid a false miss. This means that the in-replacement algorithm is divided into two parts: the replace procedure, shown in Algorithm 3; and the modified search procedure, shown in Algorithm 4.

Note that the search procedure is an adaptation of the original non-blocking linked-list algorithm [22]. Lines 5, 7, 10-17, 36 and 50 (colored in blue) in Algorithm 4 diverge from the original algorithm as part of the in-replacement logical deletion algorithm. Additionally, line 54 in Algorithm 3 and lines 19, 22, 24, 43-48 in Algorithm 4 (colored in orange) are where items are added to the memory reclamation scheme.

Replace Procedure. The first step is to call the search procedure to look for the replacee (line 14). The search procedure returns two items: the item we searched for (the

$right\_item$) and the item immediately before it (the $left\_item$). It is also possible that the item is not found[1] (line 15), in which case we can simply try to perform a normal insertion.

The second step is to mark the replacee as "in-replacement" (line 20) by performing a CAS operation[2]. Note that the only way to progress is if the replacee item gets marked, either by the calling thread or by some other thread concurrently performing a replacement or deletion (as both replacements and deletions mark items as logically deleted). After the replacee is marked, concurrent search operations trying to find the item we are replacing will retry their search operation until the replacer item is inserted.

The third step is to attempt to insert the replacer (new_item) immediately before the replacee (line 37). We can attempt to skip re-traversing the list (line 26) as we have all the information necessary to complete the insertion. The CAS operation might fail due to concurrent operations (e.g. an item inserted between $left\_item$ and $right\_item$), in which case we need to re-traverse the list. The re-traversal can not search for the replacee item by key as other items might have concurrently inserted an item with the same key (since the replacee is logically deleted, it is no longer considered to be in the list). For this reason, we re-traverse the list by searching for the replacee item by its reference, ensuring that we do not mistakenly find some other item. After re-traversing the list, it is possible that we do not find the replacee item (as it was concurrently removed) or that some other thread concurrently inserted an item with the same key, in which case the replace operation fails and it is up to the caller to decide whether to retry the operation or not.

The fourth and fifth steps are to logically and physically remove the replacee item (lines 44 and 49, respectively). Finally, we add the removed replacee item to the memory reclamation scheme (line 54 or 50), so that its memory can be de-allocated.

Given that the replace procedure first logically deletes the replacee item[3] and checks that no item with the same key was concurrently inserted, it prevents duplicate items from being in the list simultaneously, hence preventing harmful interleavings that stem from not respecting set semantics (e.g. backwards reads).

Search Procedure. The modifications made to the search procedure are simple: if we observe an item marked as "in-replacement" simply retry the search operation; if too many retries have been made, delete the marked item.

When we find an item with the key we are looking for (its reference gets stored in the $right\_item$ variable) we must check if it is marked as "in-replacement" (lines 36 and 50)[4]. If the maximum number of retrials caused by items marked as "in-replacement" has been

[1] A search operation that does not find the item might return the list's tail or an item with an incorrect key. In Algorithm 3 we ignore the second case for simplicity.

[2] Items are marked as logically deleted or in-replacement logically deleted by changing their *next* field

[3] Note that marking an item as "in-replacement" is also a form of logical deletion since an item marked as "in-replacement" will inevitably be removed by some thread.

[4] Note that $is\_marked\_replacement\_ref$ in lines 36 and 50 is only shown for clarity, as it is not necessary since is_marked_ref already checks if the item is marked as being logically deleted or "in-replacement".

Algorithm 3 The In-Replacement Logical Deletion **Replacement** operation

```
 1: function GET_MARKED_REF(item: item reference)
 2:     ▷ returns the item reference marked as logically deleted.
 3: end function
 4:
 5: function GET_MARKED_REPLACEMENT_REF(item: item reference)
 6:     ▷ returns the item reference marked as in-replacement.
 7: end function
 8:
 9: function GET_UNMARKED_REF(item: item reference)
10:     ▷ returns the item reference cleared of any existing mark.
11: end function
12: function REPLACE(list: list object, new_item: the replacer item)
13:     ▷ left_item contains the previous item, right_item contains the searched item.
14:     right_item, left_item ← search(list, new_item.key)
15:     if right_item = list.tail then
16:         return insert(list, new_item)                    ▷ item not found, do normal insert.
17:     end if
18:     do
19:         right_item_next ← right_item.next
20:         if is_marked_ref(right_item_next) or
                CAS(right_item.next, right_item_next,
                        get_marked_replacement_ref(right_item_next)) then
21:             ▷ old item marked as "in-replacement".
22:             break
23:         end if
24:     while true
25:     old_item ← right_item
26:     go to 36                                             ▷ try to skip re-searching.
27:     do
28:         ▷ left_item might have changed, re-search.
29:         right_item, left_item ← search_by_ref(list, old_item)
30:         if right_item = list.tail then
31:             return false                                 ▷ item concurrently removed, fail replacement.
32:         end if
33:         if left_item.key = new_item.key then
34:             return false                 ▷ item with the same key inserted concurrently, fail replacement.
35:         end if
36:         new_item.next ← right_item
37:         if CAS(left_item.next, old_item, new_item) then                    ▷ insert new item.
38:             break
39:         end if
40:     while true
41:     do
42:         right_item_next ← right_item.next
43:         ▷ logically delete old item.
44:         if is_marked_ref(right_item_next) or
                CAS(right_item.next, right_item_next, get_marked_ref(right_item_next)) then
45:             break
46:         end if
47:     while true
48:     ▷ physically remove old item.
49:     if not CAS(new_item.next, right_item, get_unmarked_ref(right_item_next)) then
50:         cleanup(list)                                    ▷ remove logically deleted items from list.
51:         return true
52:     end if
53:
54:     add_retired_item(right_item)             ▷ add the old item to the memory reclamation scheme.
55:     return true
56: end function
```

exceeded, we delete the marked item (line 14) so that it is not possible for threads to be indefinitely stuck in a search operation. Furthermore, we delete the marked item **by reference** (i.e., by searching for it by reference and not by key) as a means of avoiding removing a recently inserted item.

5.1.2 Posterior Insertion Replacement.

The posterior insertion replacement algorithm (shown in Algorithm 5) is conceptually simpler when compared to the in-replacement logical deletion algorithm. If there are no concurrent operations being performed, the posterior insertion algorithm can be performed by simply inserting the replacer after the replacee and subsequently removing the replacee. As it is possible that other threads perform concurrent writes, these three steps form a more accurate description:

1. Search for the last item occurrence of items with the same key as the one we want to replace (line 4);
2. Insert the new item **after** the item found (line 6); and
3. Delete the first item occurrence that has the same key as the one we have just inserted (line 11).

The posterior insertion replacement is self-contained, as it does not require changes to portions of the non-blocking linked-list algorithm not related to the replacement procedure.

The $search\_last$ procedure (present in Algorithm 5) is almost identical to the original algorithm's [22] $search$ operation, the only difference being that we search for the last occurrence of an item, instead of the first. The $search\_last$ procedure is equivalent to the search operation shown in Algorithm 4, without the replacement marking-related operations (lines 5, 7, 10-17, 36 and 50, colored in blue) and with a slight modification to line 33: from **while** $is\_marked\_ref(t\_next)$ **or** $search\_key < t.key$; to **while** $is\_marked\_ref(t\_next)$ **or** $search\_key \leq t.key$.

5.1.3 Replacement Algorithm Comparison

To assess which replacement algorithm performs best, we present a self-contained experimental evaluation of each of the algorithms. We evaluate the in-replacement logical deletion replacement (referred to as *Pointer Tagging*), the posterior insertion replacement and a replacement algorithm that simply removes the old item and inserts the new one (referred to as *None*).

In our experiments, threads continuously perform replacement operations. Every thread targets the last item in the list so that all operations potentially conflict with each other. In the case of in-replacement logical deletion replacement, if a replace operation fails it is retried until it succeeds. Table 5.1 presents the default parameters (the parameters

Algorithm 4 The In-Replacement Logical Deletion **Search** operation

```
1: function IS_MARKED_REF(item: item reference)
2:     ▷ returns true if an item reference is marked as logically deleted or as in-replacement.
3: end function
4:
5: MAX_REPLACE_RETRIES ← 5000    ▷ maximum number of retires caused by replacement constant.
6: function SEARCH(list: list object, search_key)
7:     replace_retries ← 0
8:     go to 17                                   ▷ do not check for replacement maximum the first time.
9:
10:    if is_marked_replacement_ref(right_item.next) then
11:        replace_retries ← replace_retries + 1
12:        if replace_retries ≥ MAX_REPLACE_RETRIES then
13:            ▷ too many retries due to in-replacement item, delete by reference.
14:            delete_by_ref(list, right_item)
15:            return ⊥    ▷ fail search operation; operations that call search do not check for ⊥ for simplicity.
16:        end if
17:    end if
18:    do                                                          ▷ find left_item and right_item
19:        t ← list.head; t_next ← list.head.next; marked_counter ← 0;
20:        do
21:            if not is_marked_ref(t_next) then                   ▷ reset "the last marked item found"
22:                left_item ← t; left_item_next ← t_next; marked_counter ← 0
23:            else
24:                marked_counter ← marked_counter + 1
25:            end if
26:            t ← get_unmarked_ref(t_next)
27:            if t = list.tail then
28:                ▷ reached the end of the list.
29:                break
30:            end if
31:            t_next ← t.next
32:            ▷ continue while we haven't found key or are traversing through marked items.
33:        while is_marked_ref(t_next) or search_key < t.key
34:        right_item ← t
35:        if left_item_next = right_item then                     ▷ check if items are adjacent.
36:            if right_item ≠ list.tail and (is_marked_ref(right_item.next) or
               is_marked_replacement_ref(right_item.next)) then
37:                go to 9
38:            else
39:                return right_item, left_item
40:            end if
41:        end if
42:        if CAS(left_item.next, left_item_next, right_item) then ▷ remove one or more marked items.
43:            marked_item ← t
44:            while marked_counter > 0 do      ▷ add removed items to the memory reclamation scheme.
45:                add_retired_item(marked_item)
46:                marked_item ← get_unmarked_ref(marked_item).next
47:                marked_counter ← marked_counter − 1
48:            end while
49:            if
50:                right_item ≠ list.tail and ▷ if right_item is still marked (logical or in-replacement) retry.
                   (is_marked_ref(right_item.next) or
                   is_marked_replacement_ref(right_item.next)) then
51:                go to 9
52:            else
53:                return right_item, left_item
54:            end if
55:        end if
56:    while true
57: end function
```

Algorithm 5 The Posterior Insertion Replacement **Replace** operation

```
1: function REPLACE(list: list object, new_item: the replacer item)
2:     do
3:         ▷ search for last occurrence of item with the same key.
4:         right_item, left_item ← search_last(list, new_item.key)
5:         new_item.next ← right_item
6:         if CAS(left_item.next, right_item, new_item) then
7:             break
8:         end if
9:     while true
10:
11:    return delete(list, new_item.key)          ▷ delete an item with the same key.
12: end function
```

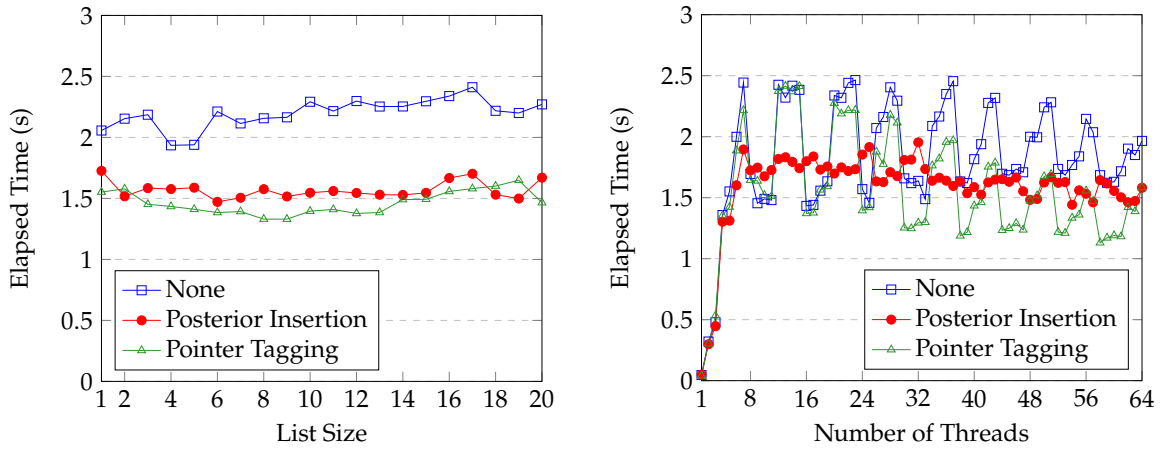
used unless otherwise specified) of our experimental tests. Every test result represents an average of 10 individual executions. The machine used for the evaluation was the same as in our profiling tests (shown in Table 4.2). Additionally, every thread reclaims items by performing FL[EE]C's memory reclamation scheme.

Table 5.1: Default experimental evaluation parameters.

Default parameters	
total item size	27 bytes
#replacements	1000000
list size	2
#writer threads	64

Figure 5.1a shows how the total execution time it takes to perform 1000000 replacement operations changes relative to the list size in an environment with high amounts of contention (due to the large number of threads). It is evident that both replacement algorithms have less overhead when compared to the naive approach. Moreover, both the pointer tagging and posterior insertion replacement algorithms seem equivalent in terms of performance, in this scenario. The fact that the pointer tagging replacement algorithm performed better likely, stems from its single-pass list traversal (when no conflicts occur) paired with its implicit backoff-like contention management: it fails when it detects that a conflicting concurrent write occurred. The posterior insertion algorithm is expected to be more performant as it is able to insert many items concurrently (as it allows duplicates to co-exist in the list), which gives it an advantage in terms of contention handling as not all concurrent replace operations will conflict with each other.

To get a better sense of how each algorithm handles varying degrees of contention, we performed experiments where we varied the number of threads performing replacement operations. Figure 5.1b shows the result of this evaluation. One can observe that when there are less than 4 threads performing replacements (i.e., a low contention scenario), all solutions perform equally as well. When there are 4 or more threads concurrently performing replace operations (i.e., medium-high contention), the pointer tagging and posterior insertion can perform better than having no dedicated replacement algorithm.



(a) Elapsed Time varying list size (64 threads).

(b) Elapsed Time varying number of threads.

Figure 5.1: Replacement algorithm comparison by varying list size and number of threads.

Both the pointer tagging and the posterior insertion solutions yielded approximately the same results, on average. The posterior insertion algorithm presented more stable results, while the pointer tagging solution was capable of performing better some of the time. Additionally, it is evident that the presence of contention resulted in a significant loss in performance, even when utilizing non-blocking concurrency control.

In terms of avoiding false misses, both the posterior insertion and in-replacement logical deletion (or pointer tagging) algorithms were extremely successful. In the tests where some threads performed the posterior insertion algorithm and other threads performed read operations, no false misses occurred. In the same tests, performed with the in-replacement logical deletion algorithm, $\approx 100\%$ of false misses were avoided, while some still ensued (as threads performing search operations are capable of deleting the "in-replacement" item and causing false misses).

From the experimental evaluation results presented, we conclude that not only are both replacement algorithms effective in avoiding false misses, but they are also more performant than a naive approach. In our evaluation, both algorithms performed equally as well.

As it is difficult to assess how well each algorithm would perform in the context of a real system (not a self-contained evaluation), in Section 6.2.7, we evaluate FL^{EE}C with each of the presented replacement algorithms.

5.2 Exponential Backoff Contention Management

Contention management mechanisms, such as exponential backoff, can be used to diminish the probability that conflicts occur and therefore improve performance (as discussed in Sections 3.2.3 and 4.3.8). In this Section, we present experimental evaluations to assess how an exponential backoff conflict management approach could help FL^{EE}C in handling

Algorithm 6 Exponential Backoff CAS

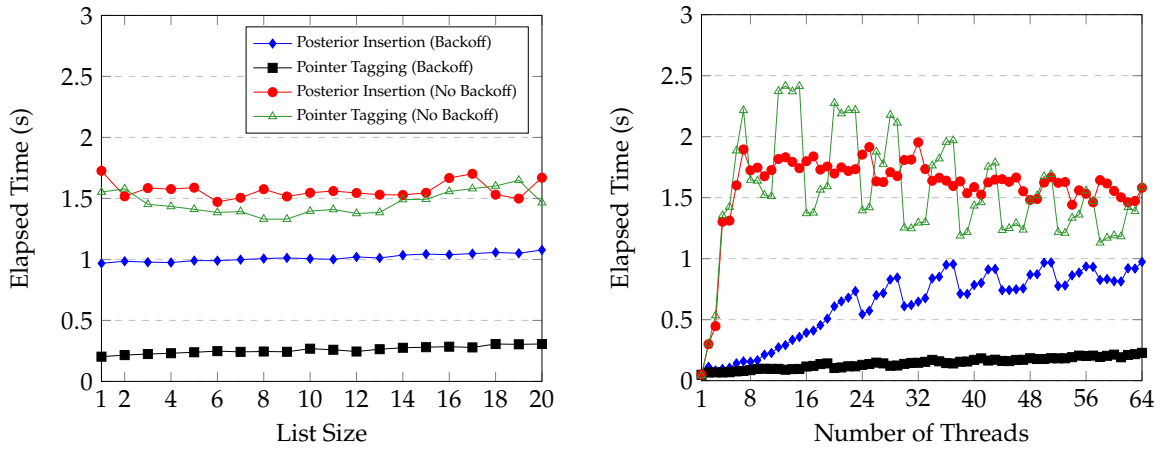
```
1: failures ← 0
2:
3: Constants: EXP_THRESHOLD; C; M;
4:
5: function ExpBackoffCAS(a: address, e: expected value, d: desired value)
6:     if CAS(a, e, ) then
7:         if failures > 0 then
8:             failures ← failures − 1
9:         end if
10:        return true
11:    else
12:        failures ← failures + 1
13:        if failures > EXP_THRESHOLD then
14:            sleep_time ← min(C × failures, M)
15:            sleep(sleep_time)
16:        end if
17:        return false
18:    end if
19: end function
```

high degrees of contention. The test environment of the results presented in this Section is the same as the one detailed in Section 5.1.3.

The exponential backoff algorithm [14], shown in Algorithm 6, is a simple wrapper for normal CAS operations: if too many CAS operations fail then sleep for some exponentially increasing time, up to some maximum; if a CAS operation succeeds, decrement the number of failures as we assume there is a lower probability of conflicts occurring and thus there is less need to backoff.

We chose to evaluate the potential benefits of the exponential backoff mechanism by applying it to the replacement algorithms presented in the previous Section (in-replacement logical deletion and posterior insertion). To know what values to assign to the $EXP\_THRESHOLD$, C and M constants, we performed a linear search by executing a benchmark with each parameter combination 10 times, taking the average and choosing the combination that yielded lower execution time. We performed the described linear search to both replacement algorithms with both 32 and 64 threads. Every linear search resulted in the same parameters.

The results in Figure 5.2a reveal how the time it takes for each algorithm (with and without exponential backoff) to perform 1000000 replacement operations changes under lists of different sizes. It is evident that the backoff mechanism is highly effective at mitigating the negative impact that contention has on execution time, as both replacement algorithms incorporating the exponential backoff mechanism outperformed their counterparts that lacked contention management Interestingly, while both replacement algorithms were approximately equivalent (in terms of performance) when no contention management was used, the pointer tagging algorithm with contention management was 3.5 to 4.7× faster than the posterior insertion algorithm with contention management. None of the algorithms (with or without contention management) was significantly impacted by an



(a) Elapsed Time varying list size (64 threads).

(b) Elapsed Time varying number of threads.

Figure 5.2: Comparison of replacement algorithms with and without **exponential backoff**, with varying list size and number of threads.

increase in the list size. Moreover, the contention management mechanism produced significantly more stable results, likely due to the fact that it lowers the probability of conflicts occurring which in turn reduces the variability between distinct executions.

Figure 5.2b shows how an increasing number of threads performing replacement operations impacts performance. When contention was non-existent (tests performed with 1 thread), the algorithms with no contention management performed $\approx 16\%$ better than the ones with the exponential backoff mechanism. In all other evaluations (i.e. when there were more than 1 thread performing replacements), the solutions with contention management performed significantly better (from 2.7 to 27× better).

In essence, the exponential backoff contention management mechanism proved effective at handling varying degrees of contention, while presenting only a slight overhead in the absence of contention. In Section 6.2.8, we evaluate FL^{EE}C with the exponential backoff mechanism, in order to judge if contention management can improve the performance in the context of a real system.

5.3 Summary

In this Chapter, we presented our non-blocking replacement algorithms in detail, evaluated them against a naive replacement approach and assessed how contention management could be helpful to mitigate the negative impact of contention. We found that not only were both of our replacement algorithms effective at reducing false misses, but they could also perform better than the naive replacement algorithm. For this reason, FL^{EE}C can potentially use any of the two replacement algorithms to avoid false misses. Moreover, we showed that contention management could be valuable to FL^{EE}C's functioning, potentially reinforcing its contention handling capabilities.

6

Evaluation

In this Chapter, we present our experimental evaluation. Our main focus when devising experiments was the performance assessment of FLEEC. As a point of reference, we also evaluate Memcached, the system FLEEC is based on, and MemCLOCK, the system that served as an intermediate step between Memcached and FLEEC.

Our experimental evaluation aims to answer the following questions:

System Saturation: when does each system exhaust its resources and reach its maximum performance?

- How many clients are needed to saturate each system?
- How does performance evolve when the server machine has more resources available (i.e., how does performance change with an increasing amount of worker threads)?

Network Overhead: how does the network impact performance?

- What performance can each system provide under workloads with different network requirements?
- What is the overhead of network communication?

Workload Changes: how do different workloads impact performance?

- Workloads with different access distributions (namely, with different amounts of skewness) impact the degree of contention systems experience. The degree of contention impacts performance. Therefore, how do workloads with different access distributions impact performance?
- How does each system perform under workloads with different read/write ratios?

Eviction Policy and Memory Reclamation: what is the impact of FLEEC's Eviction Policy and Memory Reclamation?

- How does FLEEC's approximated eviction policy impact hit ratio?

- How does FLEEC's eviction policy and memory reclamation scheme impact performance?

Replacement Algorithm: how does FLEEC perform with each replacement algorithm?

How does FLEEC with the naive replacement algorithm compare to FLEEC with posterior insertion replacement or in-replacement logical deletion replacement?

Contention Management: can FLEEC with contention management perform better?

How does FLEEC with the exponential backoff contention management mechanism compare to FLEEC with no contention management?

FLEEC Profiling: a detailed look at FLEEC's inner functioning.

What tasks does FLEEC spend more time performing and how does this change under high contention?

Real Workloads: FLEEC's performance under real workloads.

Can FLEEC perform better than Memcached (and MemCLOCK) in realistic scenarios?

We begin by describing our tests and test environment (Section 6.1) followed by a presentation of our experimental evaluations, targeted at answering each of the questions above (Section 6.2).

6.1 Tests and Test Environment

This Section aims to comprehensively describe the test environment utilized to assess how FLEEC, Memcached and MemCLOCK perform under varied scenarios. The topics discussed in this Section include:

- How test clients communicate with the cache application;
- Server related details;
- The test environment in general;
- The characteristics of synthetic workloads and how they impact the degree of contention the server experiences; and
- How real workloads are processed.

Note that the test environment was partially covered in Section 4.2, so the necessary context regarding the profiling of Memcached was given. In this Section, the test environment will be described in its entirety.

All tests were carried out in the cluster [49] of the Department of Computer Science of the NOVA School of Science and Technology (DI-FCT-NOVA). The specification of the machines used are depicted in Table 6.1.

Table 6.1: Specification of the test machines.

CPU	2 x Intel Xeon Gold 6346
#cores / #threads	32/64
L1/L2/L3 cache	1 MiB/40 MiB/72 MiB
DRAM	128 GiB DDR4 3200 MHz
NIC	2 x 10 Gbps Ethernet

Test Clients. Our test scheme involves a variable amount of clients simultaneously performing requests to the caching system. Clients use the `libmemcached` [33] library to communicate with the server over TCP. When not specified otherwise, GET requests are batched in groups of 100, to reduce network overhead [17, 36]. SET requests are never batched, as neither Memcached nor `libmemcached` support it.

Each client starts by generating/loading (for synthetic and real workloads, respectively) the requests that it will subject the server to. Clients keep their requests in memory so that they can send requests to the server instance in a timely manner. We call the phase in which clients are making requests to the server the *workload phase*. All clients start the workload phase simultaneously.

Before starting the workload phase, clients load the entirety of the item space onto a Redis [52] instance. The Redis instance acts as an external database, as it stores items persistently. During the workload phase, clients that observe a cache miss will perform a read request to the Redis instance and subsequently send a write (SET) request to the cache server. The purpose of the Redis instance is to simulate the extra work that an application making use of a caching layer must perform when a cache miss occurs. Clients handle Redis-related read requests asynchronously, i.e., clients communicate with Redis, requesting the missing items, and continue performing other work until a response from Redis is received. Clients make use of the `hiredis` [27] library to communicate with the Redis instance over TCP.

Clients have the possibility of batching GET requests so that communication overhead is reduced. Since most real workloads are read-heavy [5], batching GET requests means that most client-server communication is batched, which allows for significantly higher performance. That being said, since batching is not always possible in real applications, clients are also able to make non-batched GET requests so that it is possible to evaluate the performance of non-batched workloads as well.

Synthetic and Real workloads are organized as individual requests to a single item at a time. When clients are not batching GET requests, they simply consume workload requests linearly; making sure they communicate with Redis when a GET request is unsuccessful. When GET requests are being batched, workload GET requests are not immediately executed, but are collected instead. Once a certain number of GET requests are accumulated (the batch size), a *batched* GET request is performed. Once the response to the batched GET request is received, the client processes it and identifies the missing

items. The missing items are then requested by sending a batched GET request to the Redis instance.

During the workload phase, the main responsibility clients have is to perform caching requests as rapidly as possible. Their only other responsibility is to collect performance metrics: Throughput, Latency and Hit Ratio. It is worth noting that the only operations that count towards total throughput are successful operations. In other words, GET requests that result in a cache miss do not count towards throughput (independent of batching). This way of measuring throughput without considering failed operations is known as *goodput*.

Lastly, before the workload phase, each client pre-loads the cache with the first 5000 write requests of their individual workload. The purpose of the pre-load is so that tests simulate an environment of continuous use, i.e., tests are performed as if the cache server had been up for a longer period of time and had already stored some items. The items that are pre-loaded into the cache are highly workload-dependent (e.g. a highly skewed workload will likely only pre-load the cache server with the workload's most popular items) as the items stored depend on the workload's write request distribution. As an added benefit, the pre-load phase makes sure all clients are connected to the cache server when the workload phase begins (and the weight of handling client connections does not vary between tests).

Test Server. Memcached has a feature to save item data to an external storage medium, instead of main memory. This feature exists so that server instances have fewer memory-related requirements, and deployments can possess a significantly higher storage capacity without exorbitant costs. The external storage feature is disabled in all the tests since accesses to disk negatively impact performance.

An important factor that influenced the choice of server machine hardware is the amount of Rx/Tx (Receive/Transmit) hardware queues the NIC possesses. The NIC should contain as many Rx/Tx hardware queues as the number of hardware threads present in the CPU. A NIC that contains less Rx/Tx hardware queues than the number of CPU hardware threads will likely result in the network-related work being unevenly distributed amongst the hardware threads (as a subset of hardware threads would be tasked with processing all network-related software interrupts), which would result in a significant performance detriment.

Test Overview. Our tests were evaluated with five machines: one server machine — running an instance of either Memcached, MemCLOCK or FLEEC — and four client machines communicating with the server over the network. Each of the client machines simulates N individual clients. In general, each client machine simulated 250 individual clients (i.e., $N = 250$), for a total of 1000 clients. We chose to run 1000 clients since that was enough to saturate the server's resources (discussed in Section 6.2.1).

Each test presented constitutes an average of a minimum of 3 samples. If the samples result in a relative standard deviation greater than 10%, more samples are gathered until the relative standard deviation is lower or equal to 10% or until 10 samples are collected. If the relative standard deviation does not converge to a value inferior to 10%, the test results are posteriorly processed so that only the statistically significant samples are considered (i.e., the outlier samples are removed). We utilized throughput as the metric to determine which of the samples are statistically significant[1].

In most tests performed with synthetic workloads, the cache server has enough memory to contain the entirety of the item space (when not specified otherwise), which in most cases means the cache server has 5120 Megabytes for item storage. A caching server with enough memory to contain the entire workload's item space will never need to evict an item, although it still performs work to maintain an eviction policy and allocate/deallocate slabs for new and deleted items. This way, most tests do not incur the cost of eviction and it is possible to analyse the weight of item eviction in targetted evaluations (Section 6.2.5). Additionally, the cache server always has 64 hardware threads available, unless otherwise stated.

Finally, the machine hosting the Redis instance is one of the client machines, i.e., any machine except the one hosting an instance of FLEEC, Memcached or MemCLOCK.

Synthetic Workloads. Most of the tests presented in this Chapter stem from subjecting the server to synthetic workloads. Synthetic workloads are easily manipulated and therefore allow us to explore a vast amount of possible scenarios, to a degree that is not practical with real workloads. Our evaluation strategy was to utilize synthetic workloads to observe how the variability of one parameter affects performance.

Our synthetic workloads vary in six different ways:

1. Key Space size — how many unique keys the workload contains;
2. The size of each item's key (in bytes);
3. The size of each item's value (in bytes);
4. The Read/Write ratio — a value between 0 and 1, which represents the ratio of **read** requests in the workload;
5. The zipfian α value, i.e., how skewed is the workload (the higher the zipfian α value, the more skewed the workload is); and
6. The number of requests each client will make throughout the workload.

[1] Latency was also considered as the metric to determine statistic significance since it is typically inversely proportional to throughput. Latency was not chosen as it fluctuates less than throughput, which might be less helpful in the process of finding outliers.

The Key Space size parameter determines the maximum number of items present in the cache server. A small Key Space size means that it is unlikely for item eviction and hash table expansion to occur since the maximum number of items present in the cache is lower. Conversely, the larger the Key Space size, the higher the probability of eviction and hash table expansion to occur.

The item's key and value size determine the total item size, which determines how network-intensive the workload is. The higher the item's key and value size, the fewer network requests the cache server will be able to process per time unit. In other words, when item size increases, throughput decreases and latency increases. A consequence of the throughput decrease is the subsequent decrease in contention: if fewer requests are processed per time unit, then there is a lower chance of concurrent requests conflicting in some way, leading to a decrease in contention.

The Read/Write ratio determines the type of workload — although most workloads are read-heavy, a cache ought to provide good performance under write-intensive workloads as well. Additionally, the Read/Write ratio also influences the number of requests that can be batched (since write requests are not batched).

The zipfian α value determines the item access distribution the workload follows. An increase in zipfian α value means that the workload's access distribution is more focused on a fewer number of items. Therefore, zipfian α value increase leads to an increasing amount of requests that refer to the same items, which leads to a higher probability of conflicts and thus higher contention.

The number of client requests simply determines how long the test will run for.

A limitation of our synthetic workloads is that there is no temporal change in the access distribution. In a real workload, item popularity (the items that are more frequently accessed) changes periodically. Another limitation of our synthetic workloads is that every item has the same exact size, which could interfere with the Slab Allocator's functioning (although this is not expected to happen, as Memcached's slab allocator has a rebalancing feature to adapt to the items it is receiving).

The default values for the synthetic workload's parameters can be seen in Table 6.2. When a test does not specify a given workload parameter, the value chosen is the one present in Table 6.2. We chose to have a default Key Space size of 20000 as it is large enough for access distribution differences to have an impact on performance, while it is small enough to guarantee that item evictions and hash table expansions do not happen[2]. Item evictions are studied on dedicated tests. The Key (4 bytes) and Value size (1 byte) default values were chosen so that network-related requirements did not bottleneck the system. The impact of increasing item size is studied on targetted tests (see Section 6.2.2). The Read/Write ratio parameter's default value of 99% stems from the fact that most workloads are read-heavy [5, 31]. Similarly to the other parameters, the impact of changes

[2]Hash table expansions only occur when the number of items is greater than 1.5 times the number number of hash table buckets. Since the default number of hash table buckets is 65536, by default, a key space size of at least 98304 is required to trigger a hash table expansion.

Table 6.2: Default Parameters for Synthetic Workloads.

Default parameters	
Key Space size	20,000
Key Size	4 bytes
Value Size	1 byte
Zipfian α value (low skew)	0.4
Zipfian α value (high skew)	2.4
Read/Write ratio	99%
Number of client requests	500,000

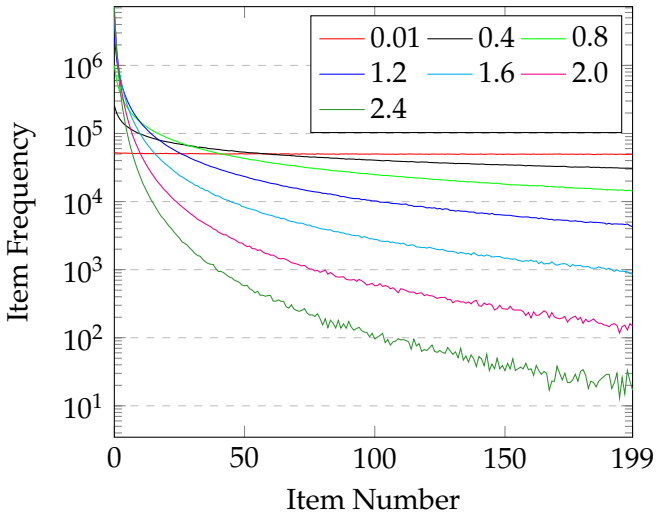


Figure 6.1: Sampling of Zipfian distributions with different Zipfian α values.

to the Read/Write ratio is studied on targetted tests (see Section 6.2.4). The zipfian α parameter has two default values: a zipfian α of 0.4 when the workload is said to be of "low skew", i.e., a workload that, although some items are more requested than others, is still relatively similar to a uniform distribution; and a zipfian α of 2.4 when the workload is said to be of "high skew", i.e., the workload's access distribution is extremely unbalanced since a few items are the target of almost all workload requests. Figure 6.1 shows how the change in zipfian α affects a workload's access distribution. The results shown in Figure 6.1 were gathered through our synthetic workload generator. Finally, the number of requests per client was chosen so that the size of each workload allowed the performance results to converge.

Furthermore, when not specified, the reader should assume that the tests are batched and that the total number of (simulated) clients is 1000.

Real Workloads. The real-world workloads used originate from Twitter's traces [31]. Due to their huge size, each trace was truncated. Furthermore, the traces are pre-processed so that a coherent work division is possible: each simulated client process is mapped to N

Twitter trace clients, i.e., each process simulates the behaviour of N (usually $N \approx 5$) Twitter clients so that each simulated client process has a similar amount of work to perform.

Each Twitter trace is composed of a collection of client requests, ordered by the time the cache server received them. Every request has a timestamp, a key, a key size, a value size, a client id, an operation descriptor (if the operation is a read or a write) and a TTL. The absolute times in the Twitter traces (the timestamps) are ignored, as each simulated client executes requests as fast as the server can handle, while the relative order of the requests made by the N clients is respected. The request's client id only serves a purpose during the pre-processing of the traces. The rest of the request parameters (the key, key size, value size, operation descriptor and the TTL) are untampered with, the only exception being the value size (so that the network intensity of the traces is modifiable).

During testing, the only difference between the synthetic and real workloads is that synthetic workloads are generated from a set of parameters, while the real traces are loaded from a file. Section 6.2.10 presents tests done with real workloads.

6.2 Results and Analysis

In this section, we present the performance tests to evaluate FLEEC. The performance analysis is focused on three performance metrics: throughput (measured as *goodput*), latency and hit ratio.

To facilitate the comprehension of the performance disparities between systems, some tests also present *throughput speedup*. Put simply, throughput speedup measures the relative throughput between two systems. More specifically, in our case, throughput speedup compares the throughput of a given system to Memcached's throughput. For example, if FLEEC had a throughput of 10 MOPS, MemCLOCK a throughput of 4 MOPS and Memcached a throughput of 2 MOPS, we say that FLEEC presented a throughput speedup of 5, MemCLOCK a throughput speedup of 2 and (naturally) Memcached a throughput speedup of 1.

Tests presented in Section 6.2.1 evaluate the saturation point of each system, as well as the impact of increasing the system's number of threads. Tests in Section 6.2.2 analyse performance under workloads with varying degrees of network requirements, as well as estimate the cost of networking in general. Section 6.2.3 evaluates how a workload's access distribution impacts performance, more specifically, how each system behaves under varying degrees of access distribution skewness. In Section 6.2.4, we evaluate how workloads with different Read/Write ratios impact performance. Section 6.2.5 presents scenarios where the workload's entire item space can not be stored in the cache server at once so that we can assess the Hit Ratio each system's Eviction Policy can provide. Section 6.2.6 presents evaluations where item eviction is forced, so that we can observe the overhead of memory reclamation (and eviction) procedures. In Section 6.2.7, we analyse how the performance FLEEC can provide with each Replacement algorithm (proposed in Section 4.3.7) changes with different workloads. In Section 6.2.8, we compare how

FLEEC with contention management pairs against FLEEC with no contention management. We profile FLEEC in Section 6.2.9, so that we can observe how much time FLEEC spends performing each task. Finally, in Section 6.2.10 we perform tests with real workloads so that we can estimate FLEEC's impact in real applications.

6.2.1 System Saturation

A way to determine if a system's resources are saturated is through its performance metrics. When a system's observed throughput reaches its limit, it is because the system is utilizing its resources the best way it can. Usually, when a throughput ceiling is reached, an increase in the number of requests per time unit will lead to a subsequent increase in average latency and a possible decrease in observed throughput.

A system's performance upper bound is tightly coupled with its available resources (the server's hardware). That being said, the system's performance upper bound is also correlated with the manner in which it uses its resources. For example, if a system does not use its CPU processing time or its memory bandwidth efficiently, a CPU or memory bus upgrade will have diminishing returns.

In this Section we focus on evaluating FLEEC, MemCLOCK and Memcached's performance limitations, so that we can evaluate how each system's performance progresses under increasing amounts of computational stress and how effectively each system makes use of its available resources.

Client Load Increase. Figures 6.2a and 6.2b present a throughput–latency graph that demonstrates how each system's throughput and latency evolves when the number of clients increases. Each measurement represents a different number of clients making requests; the first measurement represents 100 clients making requests, while the last measurement represents 2000 clients. Each sample represents an increase in 100 clients from the previous sample, meaning that each Figure (6.2a and 6.2b) presents 20 distinct measurements (each with a different number of clients) for each tested system. The only difference between Figures 6.2a and 6.2b is that in Figure 6.2a the workload presented low skew (zipfian $\alpha = 0.4$), while the results shown in Figure 6.2b were gathered with a high skew (zipfian $\alpha = 2.4$) workload.

Regarding the results presented in Figure 6.2a (low skew workload), it is evident that each system offers similar performance when the number of clients is less than or equal to 500. When the number of clients is greater than 500, FLEEC simultaneously provides higher throughput and lower latency when compared to Memcached and MemCLOCK. Additionally, FLEEC only reaches its throughput limit at $\approx$ 800 to 1000 clients, while Memcached and MemCLOCK's throughput stops increasing when the number of clients reaches 500.

With respect to the results presented in Figure 6.2b (high skew workload), it is evident that FLEEC can provide significantly higher performance than Memcached or MemCLOCK.

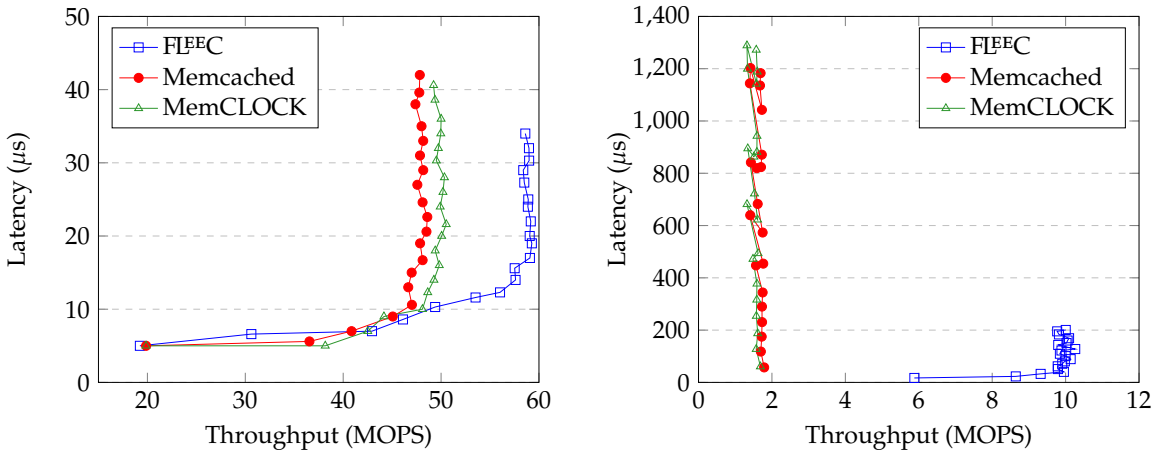
Not only does FLEEC provide higher throughput in every scenario, FLEEC also offers much lower average latency than the other systems. Moreover, Memcached and MemCLOCK's throughput limit is reached when there are as little as 100 clients making requests, whereas FLEEC's throughput cap is only reached at $\approx$ 400 clients. When the number of clients increases, Memcached and MemCLOCK's throughput remains unchanged and their latency increases substantially. FLEEC's highest measured average latency was $\approx 200\mu$s, whereas Memcached and MemCLOCK's highest average latency was $\approx 1200\mu$s. Furthermore, FLEEC was capable of a throughput of $\approx$ 10 MOPS, while Memcached and MemCLOCK's maximum throughput was $\approx$ 2 MOPS. The tests presented in Figure 6.2b not only show that FLEEC was able to provide 6× less latency and 5× more throughput, but also that FLEEC could effectively handle a significantly higher number of clients.

The disparity between the throughput–latency evaluations in low and high skew workloads are explained by *contention*. A higher workload skewness means that more requests target the same items, and subsequently the number of conflicts caused by the handling of said requests increases. The higher number of conflicts causes a higher degree of system-wide contention. The reason why FLEEC is able to perform better than Memcached and MemCLOCK is due to its non-blocking concurrency control strategy inherent robustness in the presence of contention [18]. The high skew workload test (Figure 6.2b) shows how FLEEC is better suited to deal with high contention scenarios. Nevertheless, FLEEC also performs better under the low skew workload (although to a lesser degree) because the systems are still experiencing a significant amount of contention.

Interestingly, Memcached and MemCLOCK seem to be equivalent in terms of performance. The throughput and latency differences between Memcached and MemCLOCK are small and mostly insignificant. This performance equivalence is interesting since MemCLOCK has to acquire fewer locks than Memcached, which would mean that MemCLOCK has less concurrency control overhead and could be capable of higher performance when compared to Memcached.

From the latency-throughput evaluations, we can conclude that FLEEC is able to perform better and more efficiently handle a larger number of clients than Memcached and MemCLOCK. Additionally, the throughput–latency tests also show that 1000 clients are enough to saturate every system's resources under virtually all scenarios. For this reason, we chose to run the remainder of our evaluations with 1000 clients.

Server Thread Increase. Generally, a common strategy to increase performance is to increase the amount of parallelism that the system is capable of, i.e., to increase the amount of threads that are working in parallel. Increasing the degree of parallelism has distinct effects on different systems, which leads them to have a different amount of performance improvement. Increasing the degree of parallelism also increases the degree of contention the system experiences. We aim to study how each system behaves under different degrees of parallelism and contention so that we can predict how well performance scales with an increasing amount of available resources (namely, cores/hardware threads).



(a) Low skew (zipfian $\alpha = 0.4$) workload.

(b) High skew (zipfian $\alpha = 2.4$) workload.

Figure 6.2: Throughput–Latency study under low and high skew workloads from 100 to 2000 clients, with an increase of 100 clients at each step (note the different x and y axis scales between both graphs).

Figures 6.3a, 6.3b and 6.3c show throughput, throughput speedup and latency results, respectively, with an increasing number of server threads (the number of worker threads the cache server uses to process client requests). These results were obtained with **low skewness** workloads.

Firstly, Figure 6.3b shows us that both FLEEC and MemCLOCK have $\approx$ 15% higher throughput than Memcached when each system is running with a single worker thread. In this scenario, if MemCLOCK did not present higher throughput when compared to Memcached, one might be inclined to justify FLEEC's increase in performance through the fact that non-blocking operations have less access overhead since non-blocking reads and writes have to perform 0 or 1 atomic operations (respectively) whereas all blocking operations have to perform at least two atomic operations (lock and unlock). Since MemCLOCK also presents better performance than Memcached when running with 1 worker thread, a more compelling explanation would stem from the commonalities between FLEEC and MemCLOCK: their eviction policy. It is possible that FLEEC's and MemCLOCK's approximated eviction policy significantly minimizes the amount of work performed to maintain an eviction policy and therefore improves performance. It is also possible that these two systems present better performance for two distinct purposes. A method of verifying which aspects of single-threaded FLEEC and MemCLOCK give them an advantage over single-threaded Memcached would be to perform a detailed profiling of the three systems.

Figure 6.3b also demonstrates how, in terms of performance, Memcached and MemCLOCK become more alike the higher the server thread count, which likely stems from their shared concurrency control strategy: higher thread counts lead to higher contention, which lead to the amplification of Memcached and MemCLOCK's disadvantages that stem from their blocking concurrency strategy (which lead to their similar performance).

On the other hand, FLEEC distinguishes itself through its ≈ 20% higher throughput (when compared to Memcached) independently of the number of server worker threads.

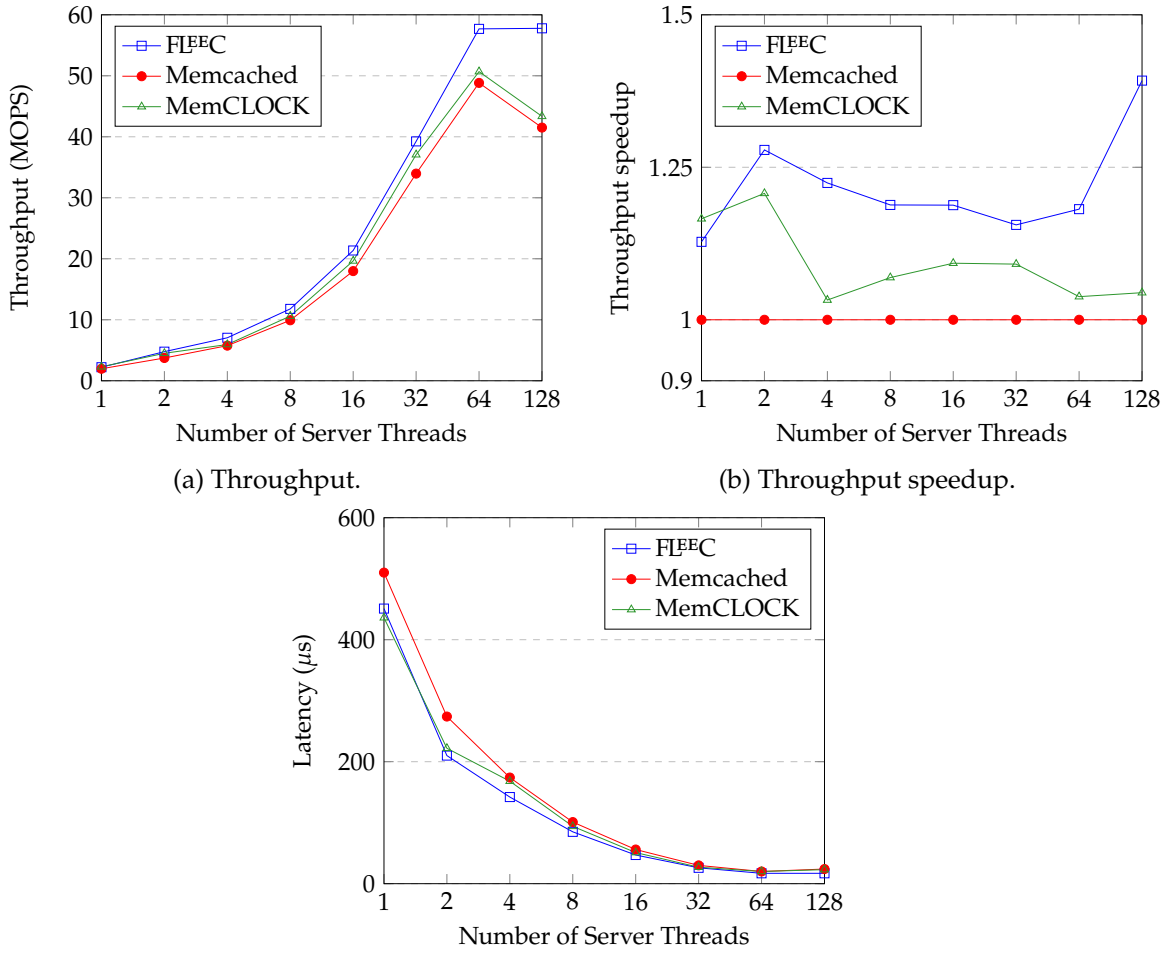
Through Figure 6.3c we can observe the average latencies each system presented. The latency results are inversely proportional to the throughput results, as FLEEC's latency is ≈ 20% lower than Memcached's.

Finally, Figure 6.3a shows us that each system's throughput increases when we increase the number of worker threads as long as we do not exceed the number of the server machine's hardware threads (64). When we attempt to utilize more worker threads than the number of available hardware threads (128 worker threads), Memcached and MemCLOCK's performance decreases ≈ 15% (relative to the throughput when using 64 worker threads), whereas FLEEC's throughput stays the same when using 64 and 128 worker threads. Memcached and MemCLOCK's performance decrease when exceeding the number of available hardware threads likely stems from the fact that the number of context switches also increased: when a thread holding a lock is suspended by the OS, other threads waiting for the same resource will remain idle for longer since the thread holding the lock will be incapable of releasing the lock until it is allowed to utilize computing resources again. FLEEC is partially immune to this problem since threads do not idle when accessing the hash table (only when accessing the slab allocator), which is the only data structure that is accessed in most operations, and subsequently leads to a significant reduction of the negative performance impact of having more worker threads than hardware threads. This behaviour means that FLEEC has the *small* advantage of being more resilient to improper configuration, in terms of performance.

Figures 6.4a, 6.4b and 6.4c show throughput, throughput speedup and latency results, respectively, with increasing numbers of server threads. These results were obtained with **high skewness** workloads.

Figures 6.4a and 6.4b reinforce the notion that FLEEC is better fitted to handle high contention scenarios than both Memcached and MemCLOCK, as FLEEC presents higher throughput in every scenario except when servers only have 1 worker thread.

Surprisingly, FLEEC's achieves slightly more throughput with 2 worker threads, than with 4, 8 or 16 threads. This slight performance decrease is likely to be linked to the increase in contention that is accompanied by the increase in thread count: since there is a very high likelihood that threads are processing a request related to the same item (due to the high access distribution skewness), there is also high probability that CPU caches have duplicates of the same data block. If CPU caches have duplicates of the same data, then a modification to said data by one thread will lead to a cache data block invalidation, which hinders performance as the hardware is ensuring cache coherence (and cache blocks of cores that had a copy of the modified data block have to be updated). A methodology to test this hypothesis could be to measure the amount of cache invalidations that occur during tests with low and high skewness access distributions: if there is a significant difference in cache invalidations in low and high skewness evaluations, it is likely that it is the hardware cache coherence mechanisms that are lowering performance. With



(a) Throughput.

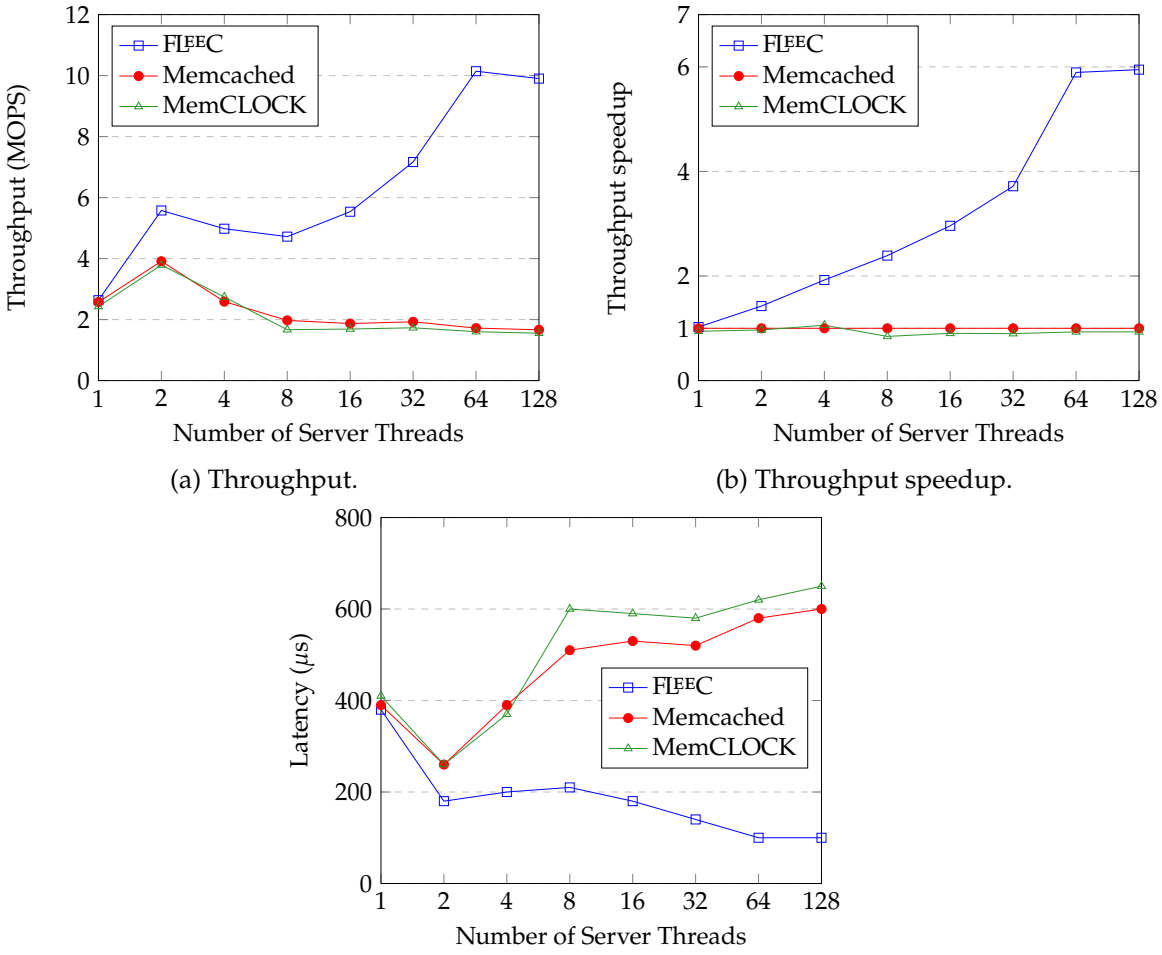
(b) Throughput speedup.

(c) Latency.

Figure 6.3: Throughput, Throughput speedup and Latency with varying amount of server threads, for a low skew (zipfian $\alpha = 0.4$) workload.

configurations of more than 16 worker threads, FLEEC's performance recovers (in the sense that it is greater than a configuration with 2 threads), which is likely explained by the fact that the positive impact of the higher degree of parallelism supersedes the negative impact of contention.

Equally surprising is the fact that both Memcached and MemCLOCK achieve their maximum throughput when utilizing only 2 worker threads. The performance loss stems from Memcached and MemCLOCK's inability to adapt to high contention levels due to their blocking concurrency control strategy: in high contention scenarios, a considerable number of threads will be competing for mutually exclusive access to a resource; the threads that lose the race for mutual exclusion access will idle, which leads to a significant amount of time spent not performing useful computation and subsequent performance loss. With configurations with more than 2 worker threads, Memcached and MemCLOCK's performance decreases because the increase in contention outweighs the benefits of having parallel executions.



(a) Throughput.

(b) Throughput speedup.

(c) Latency.

Figure 6.4: Throughput, Throughput speedup and Latency with varying amount of server threads, for a high skew (zipfian $\alpha = 2.4$) workload.

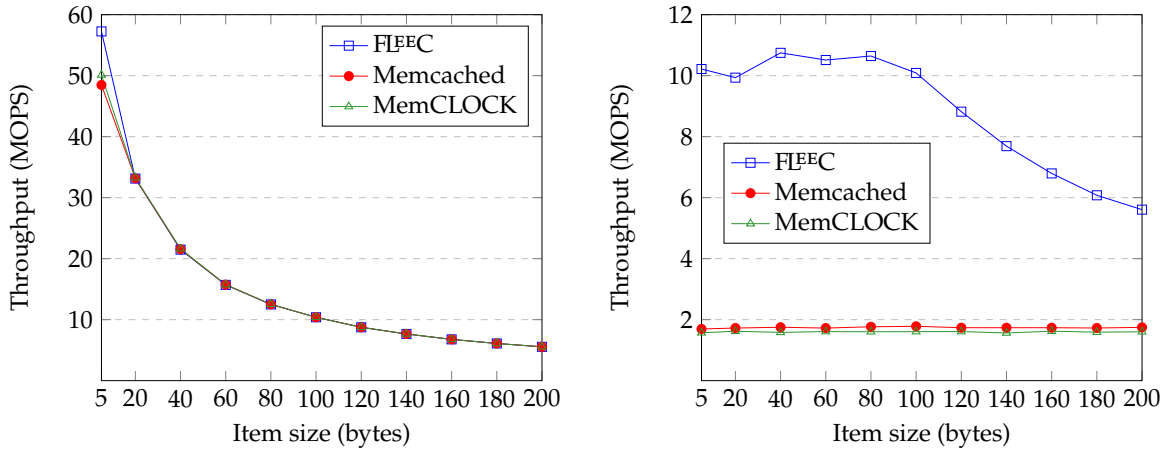
Additionally, MemCLOCK presents 5-10% slower throughput than Memcached, which further highlights FLEEC's ability to sustain good performance under high contention scenarios, since FLEEC employs the same eviction policy as MemCLOCK.

6.2.2 Network Overhead

A key-value cache server (typically) communicates with its clients through the network[3]. Given the need to communicate through the network, a caching server's performance can easily be bottlenecked by the network's inability to deliver enough requests per time unit. Additionally, the processing of network operations may also require significant work that reduces performance.

In this Section, our main concern is to understand how the network impacts each system's performance. There are two main possible strategies to evaluate the network

[3]There are some deployments that do not use the network for communication as the cache server and its clients are hosted in the same machine.



(a) Low skew (zipfian $\alpha = 0.4$) workload.

(b) High skew (zipfian $\alpha = 2.4$) workload.

Figure 6.5: Throughput with varying item sizes for low and high skew workloads (note the different y axis scales between both graphs).

impact: either limit the network's bandwidth to simulate a deployment with a less capable network or increase the workload's network requirements (increase item value size)[4]. We opted for the second approach as it was more manageable since it was easy to manipulate item size in our existing tests.

Figures 6.5a and 6.5b show each system's throughput related to an increase in item size when applying a low and high skew workload, respectively. Figure 6.5a shows a scenario where, from item sizes of 20 bytes or more, the network is bottlenecking each system's performance. The network bottleneck means that each cache server is receiving fewer requests per time unit than it is able to handle. Due to the fact that each cache server receives fewer requests per time unit, there is a subsequent decrease in contention experienced by the system: if each thread is processing fewer requests per time unit, two or more threads are less likely to conflict by concurrently accessing a data structure. The significant decrease in contention means that FLEEC's advantages are subdued, which leads to FLEEC's identical performance when compared to Memcached and MemCLOCK. Conversely, Figure 6.5b presents results where each system was subjected to a high degree of contention stemming from the workload's high access distribution skewness. In this second scenario, FLEEC's benefits manifest themselves. Since contention is what is bottlenecking each system (not the network), FLEEC is able to provide significantly higher throughput than Memcached and MemCLOCK. These results show how FLEEC can be able to provide a higher degree of performance even under workloads with high network requirements or suboptimal network conditions and emphasize FLEEC's ability to deal with high contention.

[4]These approaches are not equivalent but produce comparable behaviour.

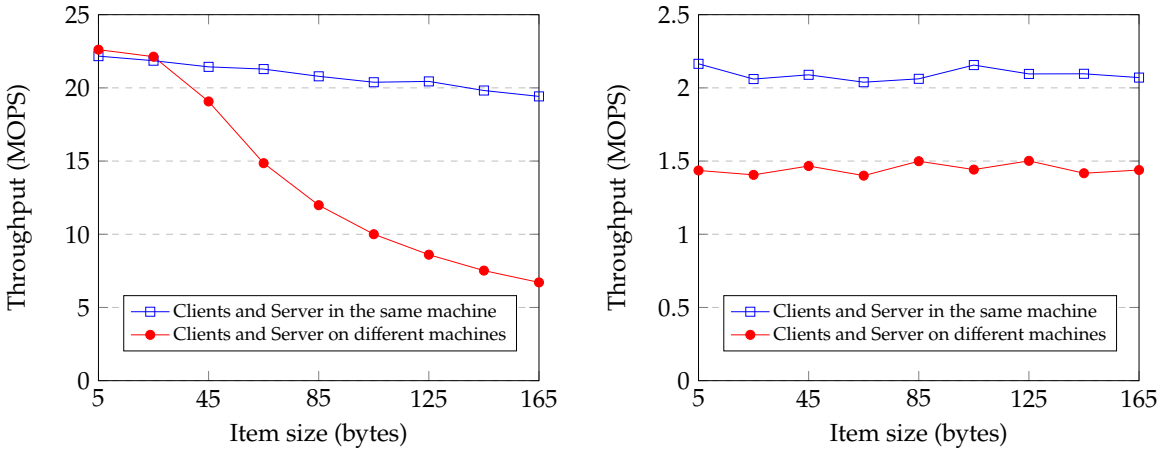
Local *vs* Network Communication To better understand the impact of network communication on performance, we compare the throughput of clients communicating with the server through the network with clients communicating locally with the server.

To accurately evaluate the differences between local and non-local communication, we ought to compare two identical scenarios. The issue is that, when performing local evaluations (clients and server executing on the same machine), the client and server processes will compete for the machine's resources, adding undesired noise to measurements. Our evaluation aims to avoid client/server competition, to more precisely simulate a realistic environment. To this end, we pin client and server processes to pre-determined hardware threads, such that the set of hardware threads occupied by client processes is disjoint from the set of hardware threads occupied by server processes. By pinning client and server processes to a disjoint set of hardware threads we ensure that the server does not compete with clients for CPU processing time, and vice-versa. This evaluation strategy does not ensure that processes do not compete for other resources not related to the machine's CPU component, but we claim that it constitutes a satisfactory environment for our evaluation purposes as it significantly diminishes resource competition.

The test machines we utilized (Table 6.1) have 32 cores. The local evaluation scenario executes clients and the server in the same machine; whereas the non-local scenario uses 2 machines, one for the server and the other for the clients. In *both* scenarios, client processes were pinned to 16 cores and the server's processes were pinned to the remaining 16 cores. By comparing the throughput achieved in both scenarios we can approximately determine the impact network communication has on performance by observing the difference in throughput from both test scenarios.

Figure 6.6a shows how throughput changes when increasing item size for tests performed with batched workloads. Surprisingly, for items of 25 bytes or less, there is no performance difference between local and non-local communication. When comparing tests performed with items of 5 and 165 bytes, performance diminishes $\approx$ 12.5% in the local communication scenario and $\approx$ 70% in the non-local communication scenario. The disparity in performance decrease between both evaluations approximately represents the impact of network communication, i.e., network communication represents an additional $\approx$ 57.5% performance decrease when comparing workloads with item sizes of 5 and 165 bytes.

Packet fragmentation can be a possible explanation for the extra loss in performance that occurs when clients and the server are communicating through the network: the Maximum Transmission Unit (MTU) value used for network communication was 1500, whereas the MTU value for local communication was 65536. This means that, in the case of network communication, the NIC fragments packets with items with more than $\approx$ 15 bytes (because the batch size is 100, which means that items with more than 15 bytes lead to packets with more than 1500 bytes). Because of fragmentation, batching becomes progressively less effective, as the network is nullifying the positive performance impact that batching has by splitting one batched request into multiple fragments. Conversely, the



(a) With batching.

(b) Without batching.

Figure 6.6: Throughput comparison between local (single machine) and network (2 machines) communication, with and without batching, with FLEEC as the caching server and 1000 clients making requests, for a read-intensive (99% reads) workload (note the different y axis scales between both graphs).

local communication evaluation never leads to fragmented packets, which allows batching to continue having a significant positive impact. Figure 6.6b shows the same results but with batching disabled. We can observe that when not batching read requests, the cost of communicating through the network is constant since the throughput in both test scenarios has a constant disparity. This constant disparity is likely explained by fragmentation since no evaluation scenario in Figure 6.6b would result in packet fragmentation (since the largest item size is 165 and batching is disabled, all packets are smaller than the MTU size of 1500). To verify this hypothesis we would perform evaluations with larger MTU sizes: if the performance detriment was less significant the higher the MTU size, we would have a high degree of confidence that, in our evaluation, the MTU was the main cause of the network's performance bottleneck.

6.2.3 Access Distribution Impact

Contention is highly influenced by the workload's access distribution, as threads concurrently processing requests referring to the same item will be more likely to conflict in accesses to the cache server's data.

Key-value cache workloads typically follow a zipfian access distribution, which are distributions that are characterized by their skewness [5]. A skewed workload is an unbalanced workload, i.e., a small portion of items are very popular and are a target of a high number of requests, while the remaining items are rarely accessed. Workload skewness is denoted by the access distribution's zipfian α parameter, the higher the zipfian α the higher the skewness (as can be seen in Figure 6.1). Thus, the higher zipfian α parameter the higher the amount of contention level the cache server experiences. In this

Section, we analyse how the access distribution's skewness affects the caching server's performance.

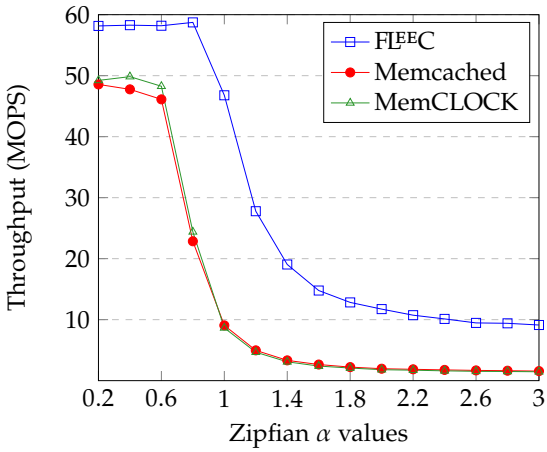
Figures 6.7a, 6.7b and 6.7c show how throughput, throughput speedup and latency changes when the access distribution's zipfian α increases. In Figure 6.7a we notice that FLEEC can achieve significantly higher throughput than Memcached or MemCLOCK. By observing Figure 6.7b we immediately perceive FLEEC's performance benefits. FLEEC achieves between $\approx 5\times$ and $\approx 6\times$ more throughput than Memcached when the zipfian α is greater than 1. When the zipfian α is lower than 1, FLEEC achieves between $\approx 1.2\times$ and $\approx 2.6\times$ higher performance. Similarly, Figure 6.7c shows how FLEEC can provide up to $\approx 6\times$ less latency than Memcached or MemCLOCK. Furthermore, when compared to Memcached, MemCLOCK presents $\approx 1\%$ to $\approx 6\%$ higher performance when the workload's zipfian α is between and 0.2 and 0.8 and $\approx 5\%$ to $\approx 10\%$ lower performance when the zipfian α is between 1 and 3.

FLEEC only suffers from a performance decrease for zipfian α values greater than 0.8, whereas Memcached and MemCLOCK suffer a performance deficit from zipfian α values greater than 0.6. Memcached and MemCLOCK's throughput stabilize at $\approx$ 1.5 MOPS, while FLEEC's throughput stabilizes at $\approx$ 9 MOPS. Additionally, when under high contention, FLEEC presents $\approx 6.4\times$ less performance than FLEEC under low contention. In the case of Memcached and MemCLOCK, the performance decrease from low to high contention scenarios is of $\approx 31\times$ and $\approx 33.8\times$, respectively.

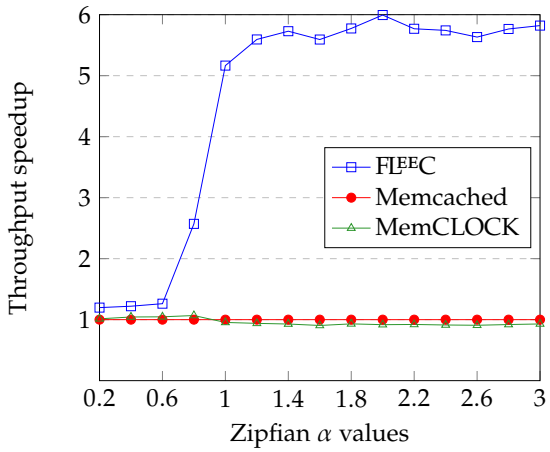
The results presented in Figure 6.7 were obtained with batched read operations. Although batching allows for the best possible performance, it is not always feasible for an application to batch requests. For this reason, we also evaluate the performance that non-batched workloads can achieve, to further understand what performance benefits to expect from FLEEC.

Figure 6.8 presents the same evaluation as Figure 6.7, where the only difference is that read requests were not batched. It is evident that, when not batching read requests, none of the systems can achieve the same degree of performance that is possible when batching is enabled. Nevertheless, starting from a zipfian α value of 1, FLEEC's benefits start manifesting. In the best case, FLEEC achieves $\approx 3\times$ more throughput than Memcached and MemCLOCK when the workload is not batched. These results indicate that even when batching is not practicable, FLEEC still manages to provide significantly higher throughput and lower latency mainly because it can maintain its degree of performance under high contention scenarios (whereas Memcached and MemCLOCK suffer a significant performance deficit).

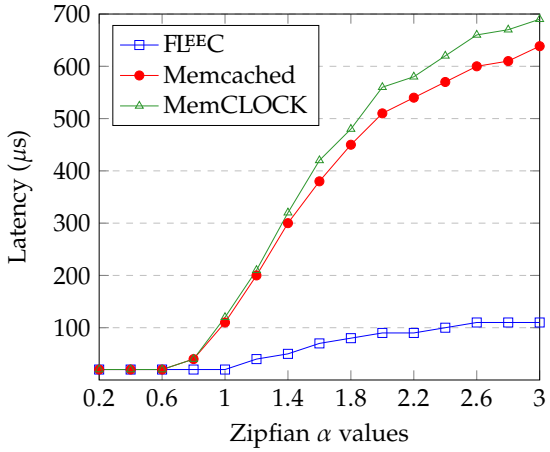
To summarize, the results presented in this section strongly reinforce FLEEC's ability to handle scenarios with a low, medium or high degree of contention.



(a) Throughput.



(b) Throughput speedup.



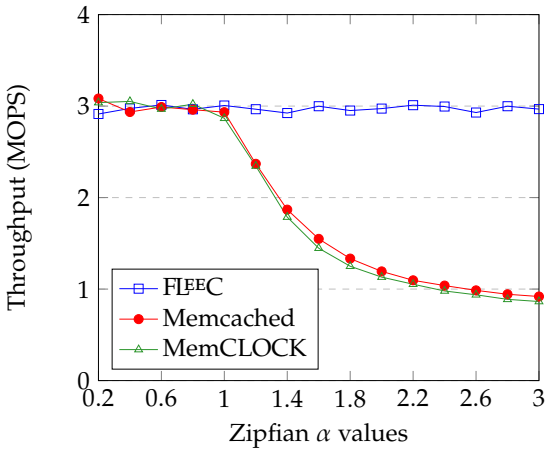
(c) Latency.

Figure 6.7: Throughput, Throughput speedup and Latency with varying zipfian α values, for a read-intensive (99% reads), batched workload.

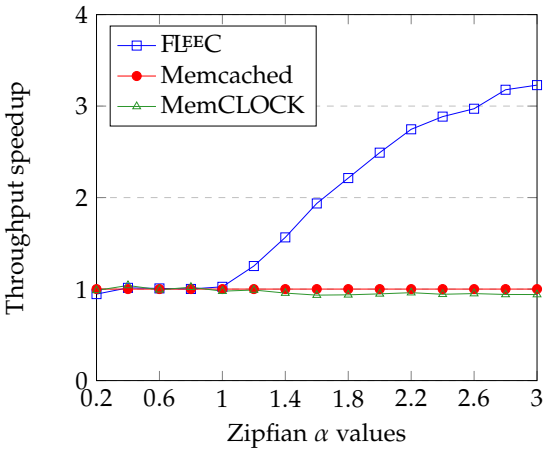
6.2.4 Read/Write Ratio Impact

Every workload evaluated thus far was extremely read-intensive, as 99% of all operations were read operations. The reason to focus our evaluation on read-intensive workloads stems from the fact that the majority of real workloads are also read-intensive [5, 31]. Nevertheless, it is still desirable for a caching server to be able to provide good performance for other types of workloads, as a cache that can only perform well under certain workloads is limited in practice (mainly because it is difficult to predict workload statistics beforehand). In this Section, we aim to evaluate how generic each caching system is, i.e., how well each system can perform under workloads with different read/write ratios. To this end, we evaluate how FLEEC, Memcached and MemCLOCK perform under workloads with 1%, 5%, 25%, 50%, 75%, 95% and 99% of reads (and 99%, 95%, 75%, 50%, 25%, 5% and 1% write requests, respectively) under low and high contention scenarios.

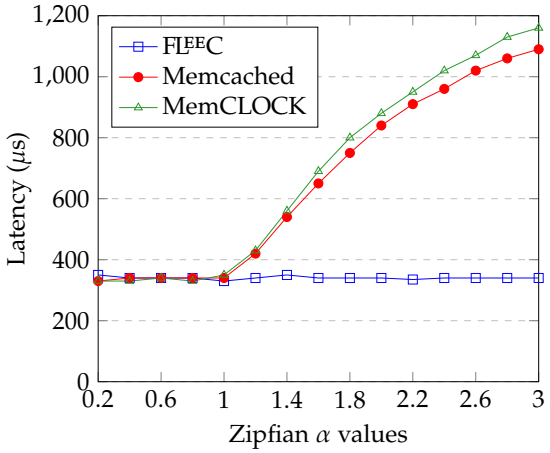
Figures 6.9a, 6.9b and 6.9c show how throughput, throughput speedup and latency change when the workload has a varying ratio of read/write operations and when the



(a) Throughput (non-batched).



(b) Throughput speedup (non-batched).



(c) Latency (non-batched).

Figure 6.8: Throughput, Throughput speedup and Latency with varying zipfian α values, for a read-intensive (99% reads), non-batched workload.

access distribution has low skewness. By observing the results shown in Figure 6.9a, it is evident that all systems perform better when the workload consists of a high portion of read requests. This happens for two reasons: firstly, read requests are batched and write requests are not, which means that write-intensive workloads benefit very little from batching and therefore present lower performance; secondly, write requests are more costly than read requests (independently of the system in question) as they have to perform the same work read requests do (search the hash table and update the eviction policy) and also have to make modifications to the caching system's data structures. It is difficult to understand how well each system performs under write-intensive workloads by simply observing Figure 6.9a. Figure 6.9b shows throughput comparisons; and demonstrates how FLEEC provides $\approx$ 20% higher performance (when compared to Memcached) at every read/write ratio configuration. From Figure 6.9b it is evident that FLEEC's performance advantages under low to medium contention are not exclusive to read-heavy workloads, but also carry over to write-intensive and balanced workloads (workloads with 50%

reads/50% writes). Figures 6.11a and 6.11b show throughput and throughput speedup obtained from a low skew non-batched workload. From these results, we gather that FLEEC's benefits when contention is low also apply to workloads where read requests are not batched. The main difference is that FLEEC does not present an ≈ 20% increased throughput all around, but a ≈ 60% increased throughput when the workload is write-intensive. FLEEC's throughput advantage under low contention decreases when the workload's percentage of writes decreases.

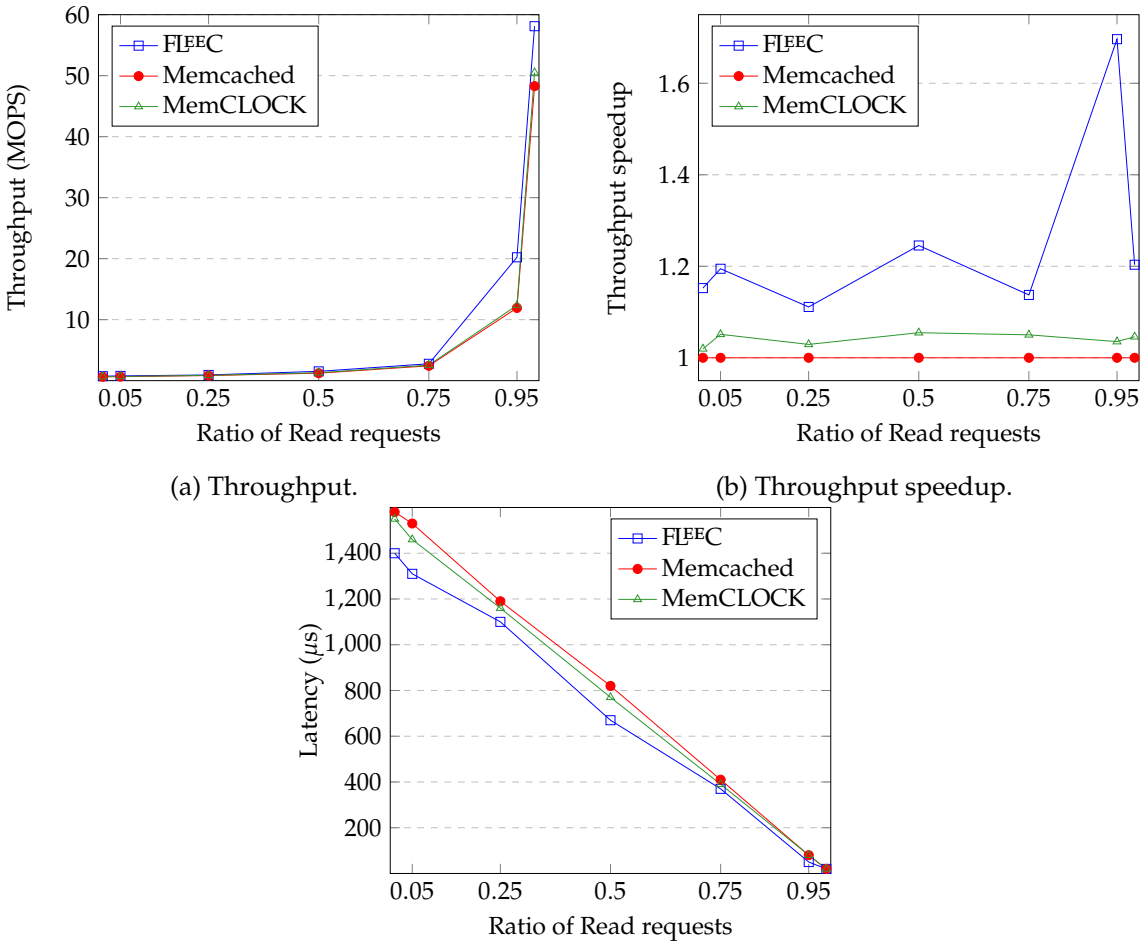
Figures 6.10a, 6.10b and 6.10c show how throughput, throughput speedup and latency change when the workload has a varying ratio of read/write operations and when the access distribution has high skewness. From Figure 6.10a we observe how FLEEC's throughput increases dramatically (increases ≈ 3.6×) when the workload's read/write ratio increases from 75% to 95%. On the other hand, Memcached and MemCLOCK's throughput does not increase as significantly (only ≈ 1.5× and ≈ 1.4×, respectively) from a workload with 75% read/write ratio to one with 95%. FLEEC's high throughput when the workload is read-intensive is due to its synchronization-free read operations. Conversely, Memcached and MemCLOCK's read operations have to synchronize between themselves to guarantee mutual exclusive accesses. To aggravate matters, Memcached and MemCLOCK's read operations may synchronize even when accessing different buckets, as each lock protects several buckets, which may lead to an unnecessary performance loss. Figure 6.10b shows that, under high contention, FLEEC is able to provide between 2× and 6× higher performance when compared to Memcached, depending on the workload's read/write ratio. Regarding FLEEC, the reason for the decreased throughput speedup[5] when workloads have read/write ratios between 25% and 75%, is likely due to the impact that contention has on the number of times the CPU's cache coherence protocol is triggered (already discussed in Section 6.2.1).

Figures 6.12a and 6.12b show throughput and throughput speedup obtained from a high skew non-batched workload. When requests are not batched, FLEEC provides ≈ 2× higher throughput than Memcached when the workload is write-intensive, and ≈ 3× higher throughput when the workload is read-intensive (due to FLEEC's synchronization free reads). Once more, FLEEC can still provide a higher degree of performance when compared to Memcached and MemCLOCK, further supporting the notion that FLEEC's improved performance is not reliant on the workload having batched requests or read-intensiveness.

6.2.5 Eviction Policy Impact

FLEEC and MemCLOCK employ a CLOCK-based, approximated LRU eviction policy as a heuristic to know what items to evict from the cache. We evaluate each system under

[5]The decreased throughput speedup refers to the fact that, regarding FLEEC, a workload with 50% read requests results in a throughput significantly lower than throughput obtained from a workload with 99% read requests. One would expect that a workload with 50% read requests would result in approximately half the throughput obtained from a workload with 99% read requests.



(a) Throughput.

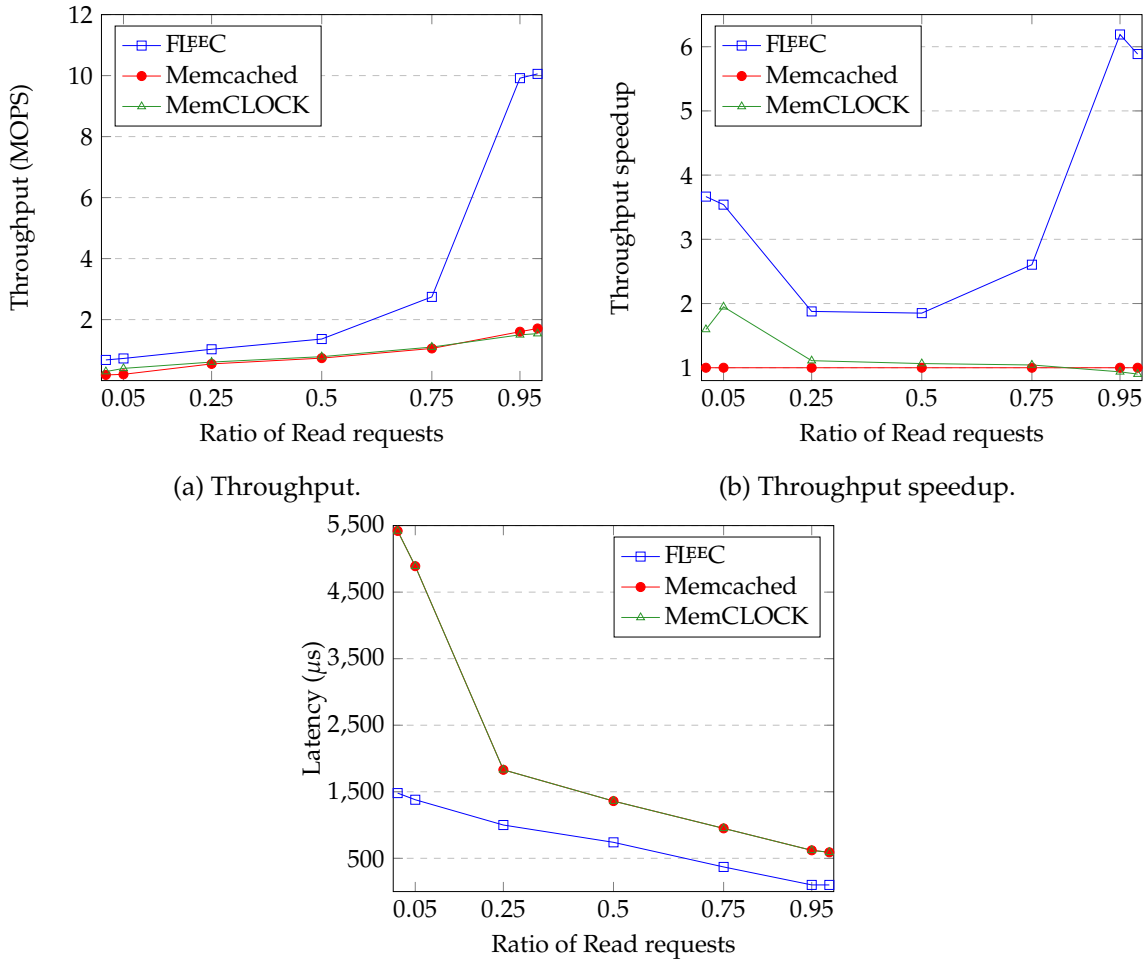
(b) Throughput speedup.

(c) Latency.

Figure 6.9: Throughput, Throughput speedup and Latency with varying read/write ratios, for a low skew (zipfian $\alpha = 0.4$) workload.

scenarios where the memory available to store data is scarce, to better understand the hit ratio that each cache system can provide in these scenarios. It is difficult to decouple a caching system's eviction policy from its memory reclamation, as a memory reclamation component's job is to reclaim the memory of items that were removed from the cache. This decoupling is even more difficult in the case of FLEEC, as the absence of mutual exclusive accesses further complicates its memory reclamation scheme. Even though in this Section we focus on the eviction policy's impact, it is important to consider that the memory reclamation scheme also has an impact on performance when an eviction occurs.

To force item eviction but not decrease performance, our evaluation of the eviction policies was done with workloads with an item space of size 100000, an item key size of 4, an item value size of 1 and a zipfian α value of 0.4. With a larger item space and an access distribution with low skewness, there is a higher likelihood that the caching system will run out of slabs in which to place new items and need to evict old ones.



(a) Throughput.

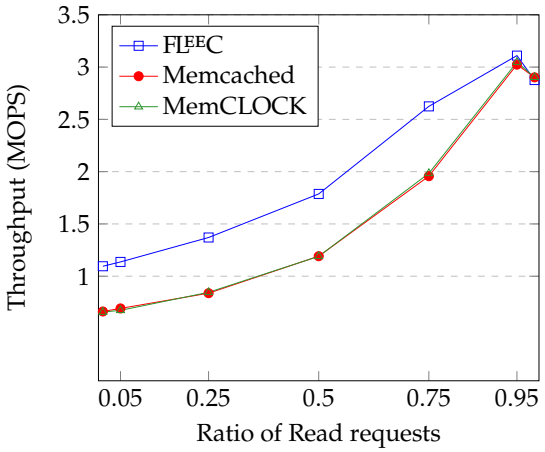
(b) Throughput speedup.

(c) Latency.

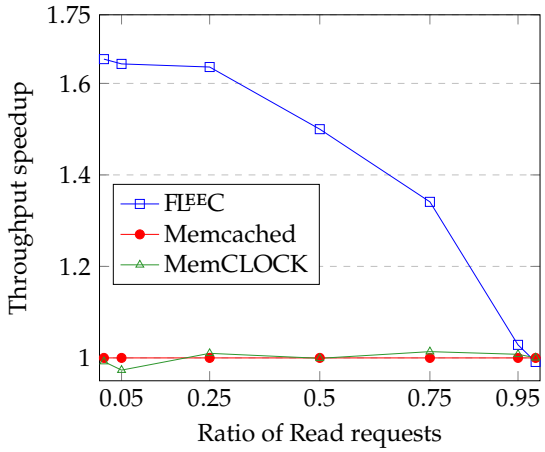
Figure 6.10: Throughput, Throughput speedup and Latency with varying read/write ratios, for a high skew (zipfian $\alpha = 2.4$) workload.

In our evaluations, we restrict the amount of main memory that the cache server can use to store items and observe the resulting throughput, latency and hit ratio. The workload's entire item space data takes up 500000 bytes (0.5 megabytes). When including meta-data, Memcached can store the entire item space in 4700000 bytes (4.7 megabytes), whereas FLEEC and MemCLOCK can store the item space in 3900000 bytes (3.9 megabytes), assuming no fragmentation.

Figures 6.13a, 6.13b and 6.13c show throughput, hit ratio and latency when each system has a variable amount of storage space available. It is by observing how hit ratios evolve in Figure 6.13b that one can understand the evaluation results in general. When a system's hit ratio is low, it will waste resources answering unsuccessful requests (which the clients will not consider towards throughput, as they measure *goodput*) and will force the application to query an external storage system (Redis [52], in our evaluations) in search of the required data, which will lower throughput and increase latency, respectively.

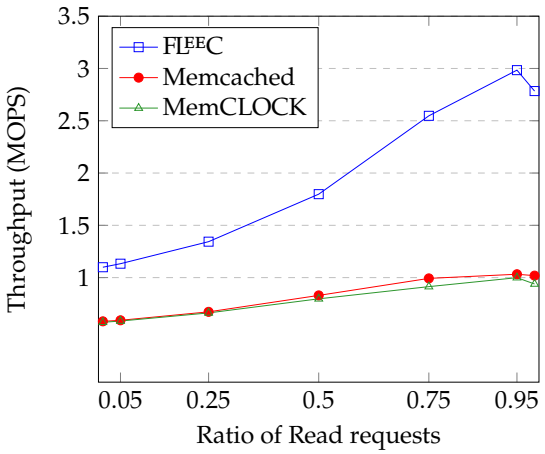


(a) Throughput (non-batched).

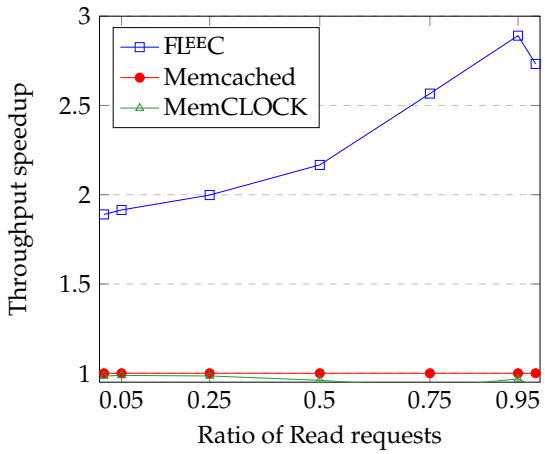


(b) Throughput speedup (non-batched).

Figure 6.11: Throughput and Throughput speedup with varying read/write ratios, for a low skew (zipfian $\alpha = 0.4$), non-batched workload.



(a) Throughput (non-batched).



(b) Throughput speedup (non-batched).

Figure 6.12: Throughput and Throughput speedup with varying read/write ratios, for a high skew (zipfian $\alpha = 2.4$), non-batched workload.

FLEEC does not provide a high hit ratio in all configurations, which causes it to have low and unstable throughput, seen in Figure 6.13a, and higher latency, seen in Figure 6.13c.

FLEEC's hit ratio only reaches $\approx$ 100% when it has 58 megabytes available for item storage, whereas Memcached and MemCLOCK can achieve $\approx$ 100% hit ratio with only 10 megabytes for item storage. Additionally, even though FLEEC can provide a hit ratio of $\approx$ 100% with 58 megabytes for available storage, its throughput only stabilizes when it has 82 megabytes of storage. The presented results show that FLEEC is not capable of providing a high hit ratio when the workload is composed of a large item space and FLEEC is not given enough storage space to hold items "comfortably". It should be noted that FLEEC's presented limitations do not stem from its memory overhead, as it has a lower memory overhead when compared to Memcached. FLEEC's lower hit ratio is likely due to

the fact that it does not synchronize modifications to its eviction policy's CLOCK values (as discussed in Section 4.3.2). By choosing not to synchronize accesses to CLOCK values, FLEEC allows data races to occur in exchange for reduced overhead when updating the eviction policy. Given that MemCLOCK provides a high hit ratio and employs the same eviction policy as FLEEC, only without the mentioned data races (since MemCLOCK's bucket locks also partially protect its CLOCK values), it is likely that the data races on FLEEC's CLOCK values are what is preventing it from providing a high hit ratio. To test this hypothesis we would have to synchronize modifications to FLEEC's eviction policy CLOCK values (e.g. with the *fetch-and-add* atomic primitive) and perform the tests presented in this section again[6].

It is also possible that it is FLEEC's memory reclamation scheme that is lowering its hit ratio and/or throughput: if it takes too long to reclaim items, FLEEC's slab allocator will not have slab slots available which will force more evictions than necessary (since there are items that have already been evicted but have not yet been reclaimed) and will lower FLEEC's hit ratio and throughput. To judge whether or not it is FLEEC's memory reclamation scheme that is lowering its performance, we would perform tests with an *incorrect* version of FLEEC that lacked its memory reclamation scheme[7] and observe its hit ratio and throughput in those circumstances. If the latency between eviction and reclamation proves to be what is hindering FLEEC in the scenarios presented in this Section, a possible solution would be to perform preemptive evictions (discussed in Section 4.3.5).

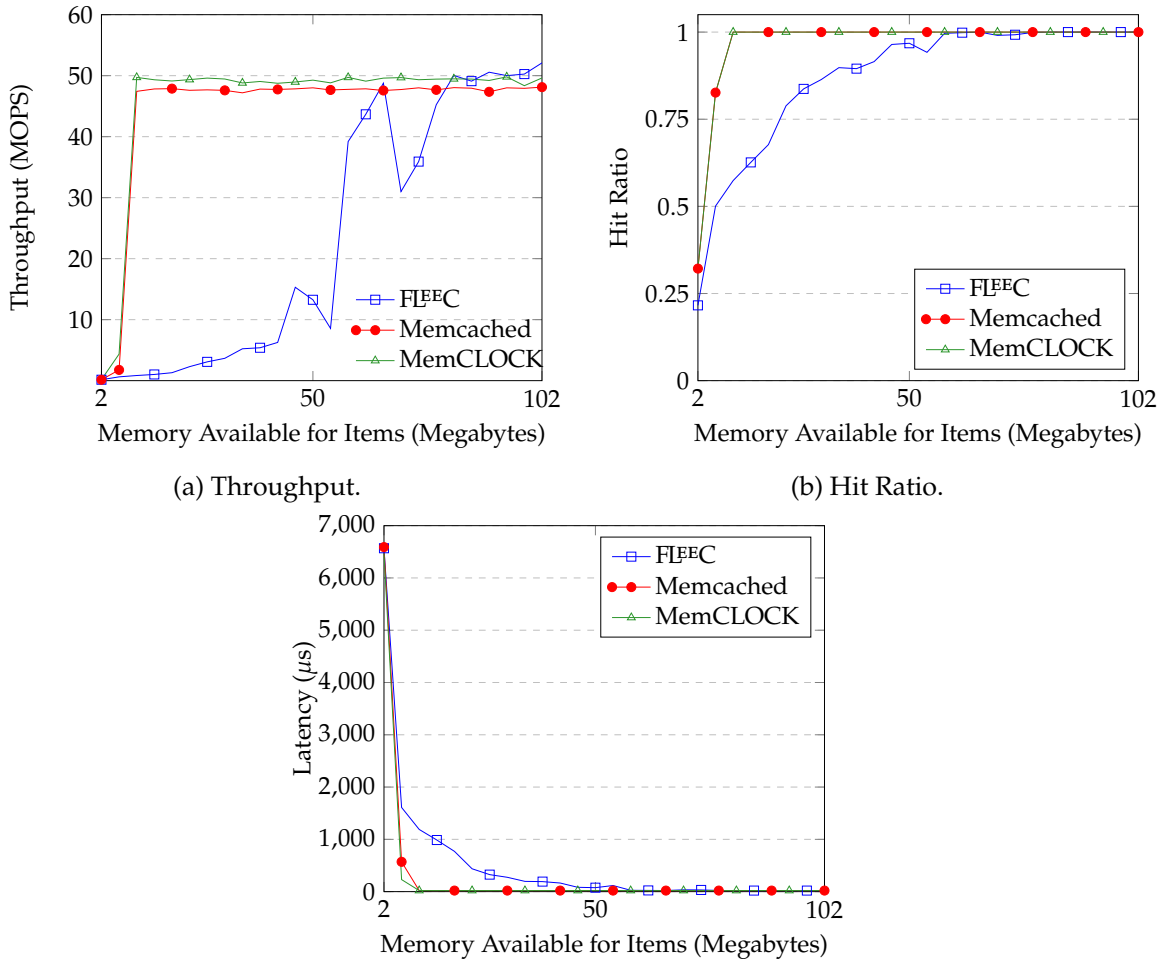
6.2.6 FLEEC's Memory Reclamation Impact

The absence of mutual exclusion in FLEEC's data structures means it requires a dedicated memory reclamation scheme to function correctly. Since the most attractive advantage of non-blocking concurrency (when compared to blocking concurrency) is its higher performance under medium and high contention scenarios, the accompanying memory reclamation scheme must have low overhead so that it does not nullify the performance benefits that come with non-blocking concurrency. In this Section, we aim to evaluate each system's memory reclamation scheme, more specifically, the impact that each memory reclamation scheme has on performance. As was also mentioned in Section 6.2.5, the memory reclamation schemes and eviction policies of a caching system are tightly coupled, which means that the results presented in this Section are not a completely isolated study of the memory reclamation scheme when applied to a caching system, but a mixture of the impact that the memory reclamation scheme and eviction policies have on performance.

To observe the cost that each system's memory reclamation has on performance, our evaluations force evictions to happen when they otherwise wouldn't. Although forceful

[6] If FLEEC with synchronized updates to its CLOCK values can provide better hit ratio, further performance evaluations should be done to verify that this modification does not hinder its performance in normal contexts.

[7] FLEEC without memory reclamation would be able to return wrong data to the client, but would not access un-allocated memory as incorrect accesses would be done to slab slots (which must have been allocated previously)



(a) Throughput.

(b) Hit Ratio.

(c) Latency.

Figure 6.13: Throughput, Latency and Hit Ratio with varying Server Memory restraints, for a low skew (zipfian $\alpha = 0.4$) workload.

evictions do not constitute a realistic scenario, they allow us to precisely control the number of evictions that happen and observe how throughput changes because of said evictions. To this end, a parameterizable percentage of write requests force evictions. Item replacements may cause evictions if the cache is out of storage space, which makes it difficult to predict the amount of evictions that occurred. For this reason, the write requests in the tests presented in this Section are purposefully ADD requests, as ADD requests do not cause item replacements when the items already exist in the cache. Additionally, because our tests will force evictions, it is still important to re-populate the cache during the test's execution time, which ADD requests are capable of doing as they create new items if they don't already exist in the cache. The workloads used for testing consist solely of ADD requests since, if the workload contained read requests, read requests that result in misses would hinder performance (and the number of misses would be unreasonably high due to the forced evictions). Put simply, our evaluations consist of workloads with

100% ADD requests, so that the only thing that changes between each evaluation is the amount of forced evictions that occur.

It is important to note that Memcached and MemCLOCK only differ in eviction policy, but share the same memory reclamation scheme. FLEEC and MemCLOCK share the same eviction policy and partially the same memory reclamation scheme. FLEEC's memory reclamation scheme has to perform extra steps when compared to MemCLOCK's (FLEEC requires items to first pass through its DEBRA based memory reclamation scheme; after this point, the steps to reclaim an item's slab are the same in all systems).

Figure 6.14 shows how throughput evolves when an increasing percentage of requests force eviction (from 0%, i.e., no requests force evictions; to 100%, i.e., all requests force an eviction). Memcached and MemCLOCK provide significantly lower throughput than FLEEC ($\approx$ 40% less throughput), when no forced evictions occur. FLEEC has higher performance than Memcached when the percentage of forced evictions is lower than 42.5%, and lower performance when the percentage of forced evictions is greater than 42.5%. These results show that, in the scenario where 42.5% or more operations induce an eviction, Memcached performs better than FLEEC, which means that FLEEC's eviction and memory reclamation procedures have more overhead than Memcached's (the same can be said for MemCLOCK), while it is difficult to know to what degree. It should be noted that the scenarios with such a high amount of evictions are not only highly unlikely (and unrealistic) but are scenarios where it is not expected for a cache system to perform well or to be beneficial to an application using it.

A possible explanation for Memcached's ability to handle a high number of evictions may be due to its HOT LRU (discussed in Section 2.1.3). Memcached's HOT LRU acts as a probationary queue as items recently added to the cache are likely to not be useful again in a short period of time. For this reason, items in the HOT LRU are not bumped and can not be evicted, as they are left to "flow" out of the LRU. Since items in the HOT LRU can not be evicted, the tests presented in Figure 6.14 may benefit Memcached as when the number of forceful evictions becomes high, no items can be evicted (as the items in the HOT LRU are protected, and the WARM and COLD LRUs are empty) while there are items in the cache, which means that Memcached's is not constantly creating new items and its eviction procedure does not continuously execute until it finds eviction victims. On the other hand, FLEEC and MemCLOCK's eviction policy may continue to run even when no items are present in the cache (as in a real scenario evictions are not triggered when the cache is empty). Furthermore, the fact that both FLEEC and MemCLOCK have similar performance losses, indicates that FLEEC's memory reclamation is not the main cause for FLEEC's performance loss.

The presented results show that Memcached's eviction policy is able to better handle workloads that cause a high number of evictions when compared to MemCLOCK and FLEEC. Additionally, due to FLEEC's higher throughput (when compared to MemCLOCK) when the number of forced evictions is high, it is possible to estimate that FLEEC's memory reclamation scheme does not significantly hinder its performance.

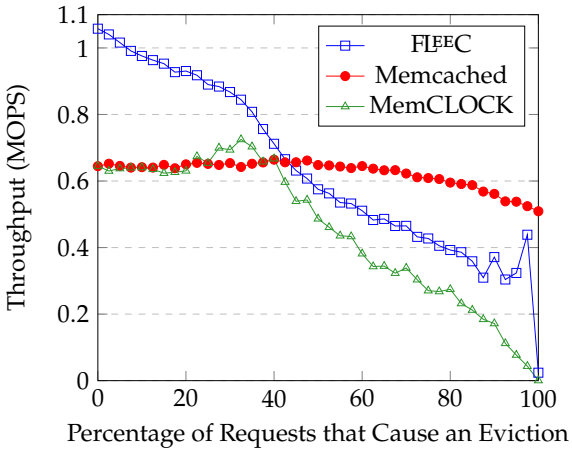


Figure 6.14: Throughput with a varying quantity of requests that force Eviction, for a low skew (zipfian $\alpha = 0.4$) workload with 25 million requests.

6.2.7 Replacement Algorithm Comparison

A simple approach to item replacement in non-blocking linked lists is to first remove the item that we want to replace (the *replacee*) and subsequently insert the new item (the *replacer*). The problem with this approach (discussed in 4.3.7) is that it allows concurrent reads to observe the intermediate linked list state where neither the old nor the new item was present, which leads to a *false miss*. In an effort to reduce the amount of false misses that occur, FLEEC implements a non-blocking replacement algorithm that prevents false misses from happening in its hash table buckets:

- A replacement algorithm that inserts the replacer after the replacee, which means that no false misses occur but temporarily violates set semantics, which we refer to as the **Posterior Insertion** replacement algorithm; and
- A replacement algorithm that utilizes Harris' non-blocking linked list [22] pointer tagging to indicate that an item is concurrently being replaced, which we refer to as the **Pointer Tagging** replacement algorithm.

In this section, we evaluate FLEEC with both replacement algorithms as well as with the naive replacement algorithm (the simple remove and re-insert approach, which we refer to as the **None** replacement algorithm).

The replacement algorithm evaluation presented looks at the throughput, latency and hit ratio that each one of FLEEC's replacement algorithms can provide when under workloads with varying degrees of skewness. The reason for testing with varying degrees of skewness is that skewness affects how many operations refer to the same values, which in turn directly affects the amount of false misses as more threads are reading items that are concurrently being replaced. The workloads used consist of a read/write ratio of 99% reads/1% writes, where every write is a SET operation that replaces the item if it is already present in the cache.

Figure 6.15c shows how the hit ratio of each replacement algorithm progresses when the zipfian α value increases. It is immediately evident that in workloads with high skewness, the hit ratio that FLEEC with the naive replacement algorithm can provide decreases significantly. In fact, the hit ratio of FLEEC with the naive replacement algorithm starts to decrease with a zipfian α value as low as 0.8. The worst case evaluated was with a zipfian α value of 3, where the hit ratio of FLEEC with the naive replacement algorithm was $\approx 60\%$.

To perceive the impact that the decreased hit ratio has on performance Figure 6.15a shows the throughput that each version of FLEEC can achieve and Figure 6.15b shows throughput speedup (throughput of *posterior insertion* and *pointer tagging* replacement relative to the naive replacement algorithm). In most scenarios, the version of FLEEC with the naive replacement algorithm performs much more poorly than the other versions. When the workload's skewness is relatively low (from zipfian α values of 0.2 to 0.6), all FLEEC versions perform equally. On the other hand, when the workload skewness is greater than 0.6, the versions of FLEEC with dedicated replacement algorithms provide higher throughput (up to 8.6×[8]) than FLEEC with the naive replacement algorithm. Finally, Figure 6.15d shows the latency each version of FLEEC can provide, which reinforces the inability of FLEEC with the naive replacement algorithm to perform well under workloads with medium to high skewness, stemming from false misses that cause a low hit ratio.

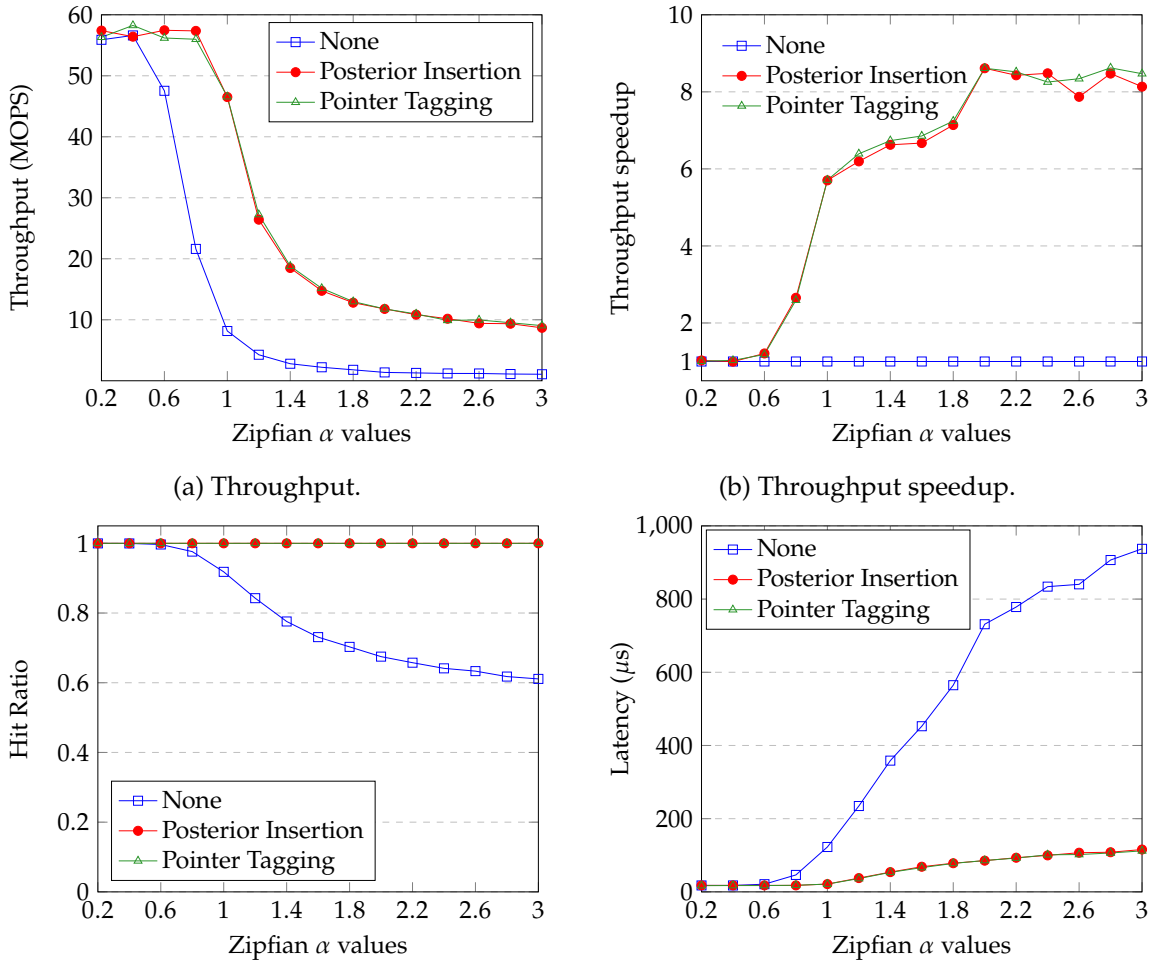
The performance tests presented also show that both replacement algorithms perform equally well under the scenarios tested. As the replacement algorithms are interchangeable in terms of performance, we argue that the *pointer tagging* algorithm is the preferable choice between the two, as it utilizes a mechanism that existed in the original linked list algorithm to maintain set semantics. By maintaining set semantics, the *pointer tagging* replacement algorithm is more likely to be correct under obscure interleavings than the *posterior insertion* replacement algorithm.

6.2.8 Contention Management

In Section 5.2 we presented a self-contained experimental evaluation of non-blocking linked-list replacement operations that utilized **exponential backoff** contention management mechanisms to mitigate the negative impact that contention has on performance. The results proved promising, as replacement operations with exponential backoff produced significantly better results than the ones without any contention management. In this Section, we assess if the exponential backoff mechanism can benefit a real system. In other words, we aim to answer the following question: is FLEEC with contention management more capable at handling contention?

We evaluate three different versions of FLEEC:

[8] It is important to note that the performance decrease that comes from (false) misses stems from the need to communicate with an external storage component (Redis [52]). To what degree performance is affected depends significantly on the details of the external storage system (meaning that in different scenarios the performance decrease caused by misses may vary).



(a) Throughput.

(b) Throughput speedup.

(c) Hit Ratio.

(d) Latency.

Figure 6.15: Throughput, Throughput speedup, Hit Ratio and Latency of FLEEC's different Replacement Algorithms under varying zipfian α values, for a read-intensive (99% reads), batched workload.

- FLEEC without any type of explicit contention management;
- FLEEC with unoptimized (i.e., unoptimized parameters) exponential backoff contention management;
- FLEEC with optimized (i.e., parameters found through a linear search[9]) exponential backoff contention management;

Figure 6.16 shows how each version of FLEEC performed with an increasing level of contention (caused by the increasing level of workload skewness). FLEEC with no contention management presented between ≈ 1.16 and $\approx 1.65\times$ higher throughput than FLEEC with optimized contention management in every scenario. FLEEC with contention management presented significantly less throughput in low contention scenarios (zipfian α values

[9]The linear search performed was similar to the one in Section 5.2, with the in-replacement logical deletion replacement algorithm and with 64 threads.

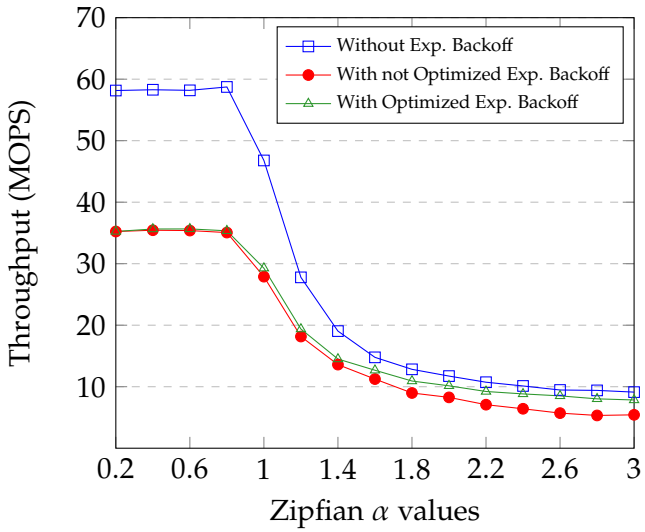


Figure 6.16: Throughput of FLEEC without Exponential Backoff, with not Optimized Exponential Backoff and with Optimized Exponential Backoff.

between 0.2 and 0.8), and similar performance in high contention scenarios. Additionally, the optimized parameters had no impact in low contention scenarios but could perform $\approx 1.45\times$ better in high contention scenarios (when compared to FLEEC with unoptimized parameters).

Applying contention management to FLEEC did not prove effective as it only added additional overhead. While it is possible that the exponential backoff parameters were suboptimal, having to perform costly computations to adjust these parameters is not a feasible requirement. For these reasons, FLEEC does utilize any explicit contention management mechanism.

6.2.9 Profiling FLEEC

To understand how the changes made to FLEEC affected its internal functioning, we profiled FLEEC under different workloads. The profiling results provide detailed information about where FLEEC spends its time, which is helpful for comparing FLEEC to Memcached as well as devising further performance optimizations. The test setup used to gather the profiling results presented in this Section is equivalent to the one described in Section 4.2[10].

It is important to note that FLEEC still uses locks to protect the *slab allocator* and *statistics* structures. The slab allocator is protected by a coarse-grained lock-based synchronization strategy, i.e., a global lock. Naturally, as the slab allocator is only accessed when memory is allocated or de-allocated, read operations are never required to access the slab allocator, and thus never acquire its lock. As most workloads are read-intensive [5], it is feasible to assume that (in most workloads) most operations rarely access the slab allocator, which means that it is unlikely that the slab allocator's lock significantly constrains performance.

[10]The only main difference between the profiling of Memcached is in the size of the workloads, which means that the absolute times are not comparable.

That being said, given the slab allocator's coarse-grained concurrency control strategy, it is possible that the slab allocator's lock becomes a contention hotspot (especially under write-intensive workloads).

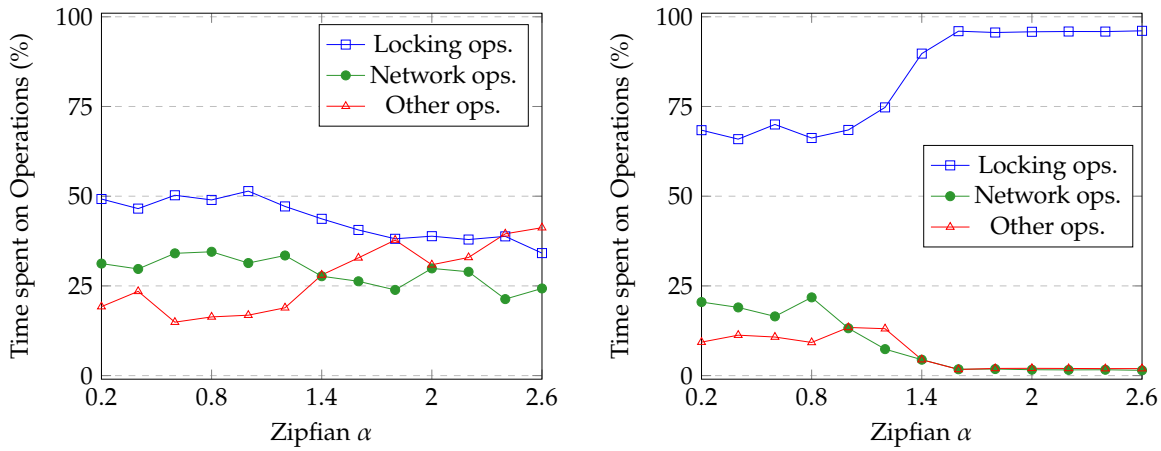
Since FLEEC protects the *stats* structure with a global lock[11] and virtually all cache operations update the cache's statistics, the statistics structure lock is a potential contention hotspot. As the statistics are constantly changing, it is not feasible to expect that the statistical results are 100% accurate, which means that an approximate approach might improve performance and produce similar results. A possible approximate solution would be to completely remove synchronization from updates to statistics (similar to the approach taken in Section 4.3.6): each thread maintains its view of the current statistics; when a client requests server statistics, we simply gather the values maintained by each thread.

Table 6.3: Profiling of FLEEC under Low Skewness (zipfian $\alpha = 0.4$) workload.

Time Spent (%)	Absolute Time (s)	Function Name	Observation
26.900	19.934	sendmsg	Send data to client
25.270	18.713	pthread_mutex_lock	Lock function
18.950	14.050	pthread_mutex_unlock	Unlock function
5.636	4.193	mcount	Profiler function
3.074	2.252	resp_allocate	Process requests
2.647	1.971	read	Receive message from client
1.537	1.121	_transmit_pre	Process requests
1.366	1.031	search	Non-blocking list search
1.281	0.951	lock_wait_private	Internal lock function
1.110	0.821	process_get_command	Process requests
1.024	0.741	lock_wake_private	Internal lock function
0.939	0.670	epoll_wait	Poll network event
0.768	0.560	tokenize_command	Process requests
0.768	0.550	assoc_find	Find item in hash table
0.683	0.500	itoa_u32	uint32_t to ASCII function
0.597	0.440	do_item_get	Find item function logic
0.512	0.380	MurmurHash3_x86_32	Hash Function
0.512	0.380	resp_add_iov	Process requests
0.512	0.360	??	Unknown library function
0.427	0.320	resp_start	Process requests
5.487	4.130	Other	N/A

Table 6.3 shows FLEEC's profiling results, obtained through the use of a low skew workload. Although FLEEC uses non-blocking concurrency for accesses to its main data structure (a hash table with embedded CLOCK-based eviction), a total of 46.54% of its time is still spent on performing operations related to locks. This indicates that there are further performance optimizations to be made to FLEEC. Additionally, as was the case with Memcached (Section 4.2), FLEEC also spends a significant amount of time (29.72%)

[11] FLEEC, Memcached and MemCLOCK utilize the same strategy for statistics, as this component was unchanged.



(a) FLEEC Profiling.

(b) Memcached Profiling.

Figure 6.17: Time spent on Locking, Network and other operations, varying Zipfian α value.

on network operations. Ultimately, ≈ 72.26% of FLEEC's execution time is spent on locking and network operations, a ≈ 20% decrease compared to Memcached's ≈ 89.31%.

Profiling FLEEC under a high skew workload yields the results seen in Table 6.4. Under a high skew workload, FLEEC spends 38.83% of its time performing operations related to locks and 23.24% of its time performing network operations. Interestingly, when compared to the low skew workload profiling results, FLEEC spends more time performing operations related to finding items in the hash table (the *assoc_find* and *search* functions), and updating its eviction policy (the *assoc_bump* function). This increase in time spent in hash table-related operations likely stems from the cache coherence protocol (discussed in Section 6.2.1) and the search operations' retrial when they find an item that is being replaced. Most notably, the time spent in the internal locking functions (the *lock_wait_private* and *lock_wake_private* functions) did not increase significantly when comparing the low skew to the high skew workload, as was the case with Memcached (Section 4.2.1). As the time spent in these functions represents an estimate of the time that processes spent waiting to acquire a lock, it is clear that the skewness of the access distribution did not cause FLEEC's processes to spend more time waiting for each other.

Figure 6.17 shows a comparison between the time that FLEEC and Memcached spent on operations related to locks, networking and other procedures. FLEEC spends less time on lock operations the higher the contention level (i.e., the higher the zipfian α), as can be seen in Figure 6.17a. On the other hand, Memcached spends a significantly higher amount of time on locking operations when the contention level increases, as seen in Figure 6.17b. Given Memcached's lower performance (when compared to FLEEC) under high contention scenarios, we reinforce the notion that FLEEC is successful at handling high contention levels because its main data structure uses non-blocking blocking concurrency control for its synchronization strategy.

Table 6.4: Profiling of FLEEC under High Skewness (zipfian $\alpha = 2.4$) workload.

Time Spent (%)	Absolute Time (s)	Function Name	Observation
19.400	16.822	sendmsg	Send data to client
18.900	16.411	pthread_mutex_lock	Lock function
15.490	13.449	pthread_mutex_unlock	Unlock function
5.899	5.104	process_get_command	Process requests
4.690	4.103	mcount	Profiler function
3.909	3.422	assoc_find	Find item in hash table
3.909	3.402	item_is_flushed	Check if item's TTL is over
3.837	3.362	search	Non-blocking list search
2.914	2.542	assoc_bump	Update CLOCK values
2.700	2.322	resp_allocate	Process requests
2.345	2.061	lock_wake_private	Internal lock function
1.990	1.711	lock_wait_private	Internal lock function
1.918	1.651	read	Receive message from client
1.847	1.631	limited_get	Wrapper for item_get
0.995	0.881	_transmit_pre	Process requests
0.852	0.771	epoll_wait	Poll network event
0.639	0.570	__strncmp_sse42	Compare item keys
0.639	0.570	do_item_get	Find item function logic
0.639	0.530	tokenize_command	Process requests
0.568	0.470	get	Wrapper for search
5.920	5.031	Other	N/A

Figure 6.5 shows FLEEC's profiling results obtained with a workload that has 95% write operations (and 5% read operations). Virtually all write operations are replacement operations (as it is very likely that they refer to already existing items), which means that almost all operations will need to allocate a new item and de-allocate an old one, thus the number of times that the slab allocator is accessed substantially increases. The impact of the increased slab allocator access frequency can be seen in the additional time spent in the *lock_wait_private* and *lock_wake_private* functions (when compared to the read-intensive profiling results shown in Tables 6.3 and 6.4). These results stem from the fact that accesses to FLEEC's slab allocator are still synchronized through a (coarse-grained) blocking concurrency control strategy and that there is still room for improvement relative to FLEEC's concurrency control.

6.2.10 Performance under Real Workloads

The workloads utilized thus far have been synthetic workloads. Synthetic workloads are extremely useful for evaluation purposes as they allow instant one-dimensional changes to the workload that are difficult to accomplish in real workloads (e.g. in a real workload, simply truncating keys to lower their key size also affects the workload's access distribution). Our evaluations take advantage of synthetic workloads to understand how each system handles different workloads that differ very slightly from one another. That being said, it is also desirable to conduct evaluations with real workloads, to try to simulate a more

Table 6.5: Profiling of FLEEC under a workload with 0.95% writes (and zipfian $\alpha = 0.4$) workload.

Time Spent (%)	Absolute Time (s)	Function Name	Observation
35.672	64.635	sendmsg	Send data to client
17.596	31.922	lock_wake_private	Internal lock function
16.912	30.641	lock_wait_private	Internal lock function
5.922	10.748	pthread_mutex_lock	Lock function
5.340	9.657	pthread_mutex_unlock	Unlock function
5.203	9.407	read	Receive message from client
1.711	3.092	mcount	Profiler function
1.506	2.742	epoll_wait	Poll network event
1.027	1.891	__strncmp_sse42	Compare item keys
0.684	1.221	drive_machine	Main loop
0.650	1.201	slabs_alloc	Allocate new slab (lock logic)
0.547	1.001	resp_allocate	Process requests
0.410	0.761	??	Unknown library function
0.342	0.650	libc_alloca_cutoff	libc allocation function
0.342	0.630	out_string	Process requests
0.342	0.610	do_item_alloc	Allocate new slab
0.308	0.560	slabs_clsid	Get slab of item
0.308	0.550	search	Non-blocking list search
0.308	0.530	tokenize_command	Process requests
0.239	0.440	conn_set_state	Process requests
4.624	8.330	Other	N/A

realistic scenario to better understand how each system would perform in practice. For this reason, in this Section, we assess the performance of each system with real workloads.

We chose to use twitter's traces [31] for our real workloads. Due to the huge size of the real workloads, we opted to truncate them into more manageable sizes of 100 to 500 million requests. Given that real workloads present temporal variation in their parameters (e.g. the access distribution might change every so often), we analysed the truncated workloads in their entirety so that their key space size, average key size, average value size, zipfian α value and read/write ratio were known[12]. The truncated twitter traces used and their characteristics can be seen in Table 6.6. Note that the presented zipfian α values of real workloads are estimates taken from measuring the access frequency of (all) keys present in the workloads.

We changed the value size of all items in each workload to *1 byte*, in an effort to reduce the limitations that the network infrastructure had on our experimental evaluation. The exception to the item value size modification is the evaluation presented in Figure 6.18b (and, in a sense, the test presented in Figure 6.19b as *cluster 21* already had items of size 1).

Figure 6.18a shows the performance results that stem from *cluster 1*. These results show that FLEEC provided between 11% and 18% higher performance than Memcached and MemCLOCK when executing with 16 threads or less. After that point (with 32 or more

[12]The parameters of the truncated workloads differed from the complete workloads (the complete workload parameters can be seen in: twitter traces – github).

threads), all systems performed equally. Furthermore, every system reached its maximum throughput, ≈ 15.7 MOPS, with just 32 threads. This limitation is consistent with the evaluation performed in Section 6.2.2 (Figure 6.5a) — when the total item size was equal to 60 bytes, every system presented a throughput of ≈ 15.7 MOPS — which indicates that the performance stagnation shown in Figure 6.18a is due to a network-related bottleneck. Considering that FLEEC demonstrated superior performance when network constraints were not a limiting factor (when the servers had between 1 and 16 threads), it is highly probable that FLEEC would be able to perform better (than Memcached and MemCLOCK) in a more capable network infrastructure.

Figure 6.18b presents results gathered by utilizing an unmodified version of the *cluster 1* real workload. It is evident that, as the network requirements were more severe, there are no performance differences between each system as all of them can process requests at a higher rate than the network is providing them (when the number of threads is greater or equal to 2).

Figures 6.19a and 6.19b show the results gathered through *cluster 8* and *cluster 21*, respectively. Relative to *cluster 8*, FLEEC provided 0% to 72% higher throughput than the other systems. Relative to *cluster 21*, FLEEC provided 17% to 82% higher throughput than Memcached and 0% to 35% higher throughput than MemCLOCK. The larger performance disparity (when comparing *cluster 8* and *cluster 21* to *cluster 1*) stems from the disparity in item key size between the two workloads.

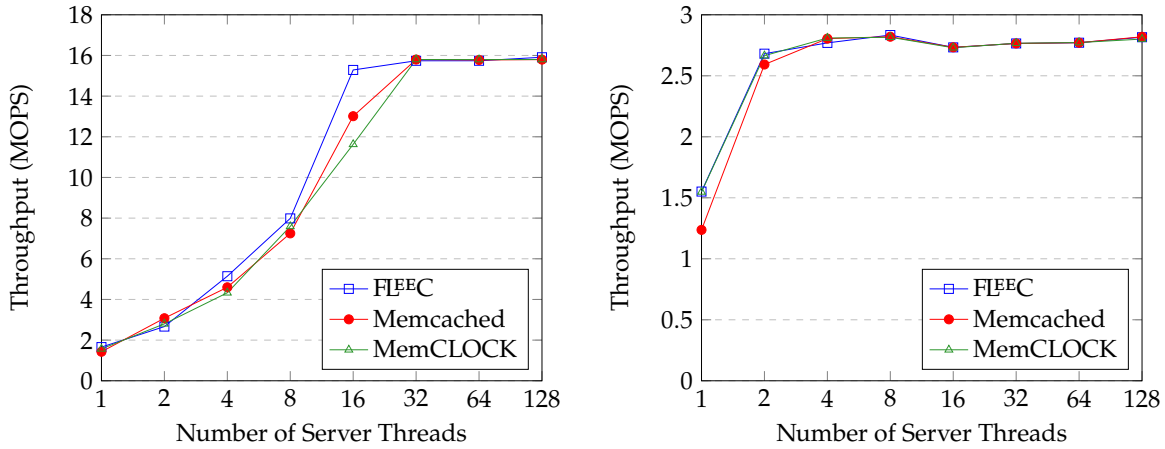
We conclude that, although highly dependent on the network infrastructure available and the network requirements of the workload, FLEEC is able to provide significantly better performance than Memcached and MemCLOCK when performing experimental evaluations with real workloads.

Table 6.6: Properties of Workloads used for Profiling.

	Key Space Size	**Key Size Average**	**Value Size Average**	**Zipf α (estimated)**	**R/W (%)**	**#requests per (simulated) client**	**#clients**
cluster 1	48 341	58.0	362.0	1.53	99.50%	434 782	6687
cluster 8	211 137	23.0	18 428.4	0.95	50.00%	139 710	2648
cluster 21	48 937 955	24.6	1.0	0.05	0.00%	136 035	631

6.3 Summary

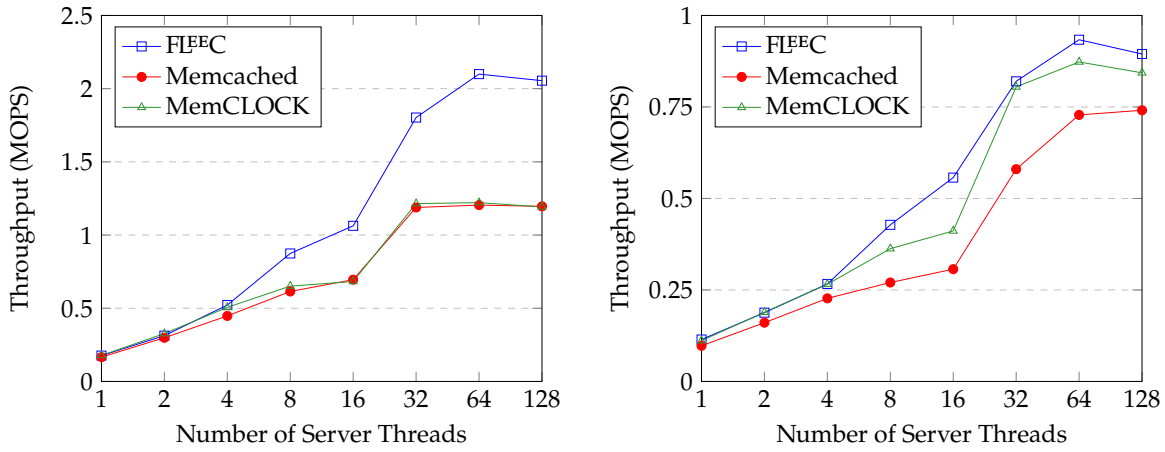
In this Chapter, we presented an experimental study of FLEEC, Memcached and MemCLOCK. We showed that FLEEC can provide higher performance than Memcached or MemCLOCK when under varying degrees of contention and equivalent performance when contention is practically non-existent. We also identified FLEEC's eviction policy limitations and presented possible causes and solutions. We evaluated FLEEC with different replacement algorithms and showed that a dedicated replacement algorithm is crucial



(a) Cluster1 with item value sizes of 1 byte.

(b) Cluster1 with unaltered item sizes.

Figure 6.18: Throughput with a modified and unmodified version of the *cluster 1* real workload, varying the number of server threads (note the different *y* axis scales between both graphs).



(a) Cluster8.

(b) Cluster21.

Figure 6.19: Throughput when using real workloads (cluster 8 and cluster 21) with item value sizes of 1 byte, varying the number of server threads.

in avoiding false misses and maintaining high performance. Furthermore, we evaluated FLEEC with an exponential backoff contention management mechanism and found that it significantly reduced performance in all scenarios. We also profiled and analysed FLEEC to understand where it spends its time. Finally, we carried out evaluations with real workloads to judge how well FLEEC performed under realistic scenarios and found that it was able to perform better than Memcached and MemCLOCK when the network was not bottlenecking performance.

7

Conclusion and Future Work

Conclusion. The main focus of this work was on improving the performance of an applicational cache (of which the baseline system was Memcached [42]), especially when under high degrees of contention, through the utilization of non-blocking concurrency control strategies.

Our cache system — FL^{EE}C, a Fast Lock-Free Cache — was devised by:

- Merging Memcached's main indexing and eviction policy data structures into a single data structure that performed both functions, so that a non-blocking strategy could be correctly applied;
- Substituting Memcached's blocking concurrency control strategy by a non-blocking one and guaranteed its correct operation through the application of a memory reclamation scheme (based on DEBRA [8]) that took advantage of relaxed cache semantics;
- Replacing Memcached's stop-the-world style hash table expansion mechanism by a non-blocking one, so that the strong progress guarantees of non-blocking read operations could be maintained; and
- Devising two distinct replacement algorithms to prevent performance-degrading false misses that were possible due to the absence of mutual exclusion: one that uses pointer tagging (a mechanism present in the original algorithm) to signal to other threads that an item is being replaced; and another that inserts the new item after the old one, temporarily allowing duplicate items to exist, while avoiding false misses since at any given time, there is always at least one item present in the data structure.

Our experimental evaluation concluded that FL^{EE}C was capable of providing significantly higher performance when the degree of contention was high, better performance when contention was low and equivalent performance when contention was practically non-existent. Thus, FL^{EE}C can be easily used to further improve the performance of any application that currently uses Memcached, with no significant drawback.

Lessons Learned. We learned that, although a non-blocking approach to concurrency can have a significant positive impact on performance, it requires considerably more conceptualization and implementation effort when compared to other known strategies. A great deal of complexity stems from the large amount of interleavings that are possible in a non-blocking algorithm. This complexity is magnified when the system contains more than one non-blocking data structure. To this end, we consider the approach of first merging data structures into a single data structure with more responsibilities, helpful in order to simplify the process of guaranteeing correctness under concurrent non-blocking modifications.

We also found that synthetic workloads were invaluable throughout the evaluation process as they allow for the evaluation of widely different scenarios due to their flexibility. Although real workloads constitute more realistic scenarios, they present some significant challenges:

- As real workloads are difficult to manipulate (due to their huge size), it is not realistic to change some of their characteristics (e.g. access distribution; number of clients). Due to these restrictions, there is a stricter limitation in what types of experimental evaluation can be done with realistic workloads, which means that, although the scenarios evaluated are more realistic, it is difficult to diverge from those scenarios and perform a broad analysis of the systems at hand.

- Some aspects of real workloads are overly complex (e.g. access distribution), which makes the results obtained with real workload A difficult to compare to results obtained with real workload B (or even some synthetic workload).

Lastly, hardware characteristics proved, once again, a determining factor in the interactions observed at a higher level. In our case, we observed that some hardware threads tended to receive more work than others. Proper hardware-related knowledge (and adequate documentation) was a key factor in the understanding, diagnosing, and fixing, of some unexpected results that were being conditioned by the hardware.

Future Work. Some improvements to FLEEC and the experimental evaluation presented have been left as future work. The following ideas present possible avenues in which to further improve FLEEC's performance:

- Change FLEEC's slab allocator structure's concurrency control into a non-blocking strategy to improve the performance of operations that have to allocate and/or de-allocate memory and enhance FLEEC's progress conditions;

- Change the slab allocator's memory management so that an item's data and (some of its) meta-data are stored separately as a means of providing faster hash table searches — if meta-data required to search items is kept contiguously in memory, linked list searches would utilize the CPU's cache more efficiently as they would

contain more relevant (meta-) data and the number of CPU cache misses would be reduced;

- Transform FLEEC's hash table buckets into non-blocking unrolled linked lists, which are linked lists where each entry contains more than one item (i.e., entries are blocks of items) and are more performant as they utilize the CPU cache more efficiently because entries are stored contiguously in memory (would only work in unison with the previous point, as entries need to be contiguous in memory for this improvement to be worthwhile);
- Add preemptive eviction to FLEEC (as discussed in Section 4.3.5);
- Add fault tolerance to FLEEC's DEBRA [8] based memory reclamation scheme (e.g. through the DEBRA+ [8] methodology); and
- Remove synchronization that protects the data structure that maintains statistics to avoid unnecessary overhead.

Additionally, a possible way to enhance our evaluation methodology would be to introduce temporal changes in our synthetic workload's access distribution.

Bibliography

[1] K. Alagarsamy. "Some myths about famous mutual exclusion algorithms". In: *ACM SIGACT News* 34.3 (2003-09), pp. 94–103. DOI: 10.1145/945526.945527. URL: https://doi.org/10.1145/945526.945527 (cit. on p. 2).

[2] D. Alistarh, K. Censor-Hillel, and N. Shavit. "Are lock-free concurrent algorithms practically wait-free?" In: *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*. ACM, 2014-05. DOI: 10.1145/2591796.2591836. URL: https://doi.org/10.1145/2591796.2591836 (cit. on p. 23).

[3] D. Alistarh et al. "ThreadScan". In: *ACM Transactions on Parallel Computing* 4.4 (2017-12), pp. 1–18. DOI: 10.1145/3201897. URL: https://doi.org/10.1145/3201897 (cit. on pp. 14, 25).

[4] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. "Operating systems". In: North Charleston, SC: Createspace Independent Publishing Platform, 2018-09, pp. 359–371 (cit. on p. 2).

[5] B. Atikoglu et al. "Workload analysis of a large-scale key-value store". In: *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*. ACM, 2012-06. DOI: 10.1145/2254756.2254766. URL: https://doi.org/10.1145/2254756.2254766 (cit. on pp. 2, 19, 32, 43, 46, 67, 70, 81, 83, 95).

[6] J. Bonwick. "The Slab Allocator: An Object-Caching Kernel Memory Allocator". In: *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*. USTC'94. Boston, Massachusetts: USENIX Association, 1994, p. 6 (cit. on pp. 5–7, 21).

[7] C. Breshears. *The art of concurrency: A thread monkey's guide to writing parallel applications*. " O'Reilly Media, Inc.", 2009 (cit. on p. 10).

[8] T. A. Brown. "Reclaiming Memory for Lock-Free Data Structures". In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. ACM, 2015-07. DOI: 10.1145/2767386.2767436. URL: https://doi.org/10.1145/2767386.2767436 (cit. on pp. 4, 15, 16, 25, 27–29, 45, 46, 102, 104).

[9] F. Corbato. *A Paging Experiment with the Multics System*. Project MAC MAC-M-384. 6.033 Notes: M.I.T., 1968-07 (cit. on pp. 4, 41).

[10] D. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998. ISBN: 9780080573076 (cit. on p. 9).

[11] *DCAS*. URL: https://en.wikipedia.org/wiki/Double_compare-and-swap (visited on 2023-01-09) (cit. on p. 24).

[12] D. Dechev, P. Pirkelbauer, and B. Stroustrup. "Understanding and Effectively Preventing the ABA Problem in Descriptor-Based Lock-Free Designs". In: *2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. IEEE, 2010. DOI: 10.1109/isorc.2010.10. URL: https://doi.org/10.1109/isorc.2010.10 (cit. on p. 24).

[13] D. L. Detlefs et al. "Lock-free reference counting". In: *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM, 2001-08. DOI: 10.1145/383962.384016. URL: https://doi.org/10.1145/383962.384016 (cit. on p. 25).

[14] D. Dice, D. Hendler, and I. Mirsky. "Lightweight Contention Management for Efficient Compare-and-Swap Operations". In: *Euro-Par 2013 Parallel Processing*. Springer Berlin Heidelberg, 2013, pp. 595–606. DOI: 10.1007/978-3-642-40047-6_60. URL: https://doi.org/10.1007/978-3-642-40047-6_60 (cit. on pp. 29, 55, 63).

[15] E. W. Dijkstra. "Solution of a problem in concurrent programming control". In: *Pioneers and Their Contributions to Software Engineering*. Springer, 2001, pp. 289–294 (cit. on p. 2).

[16] *Transmission Control Protocol (TCP)*. Tech. rep. 2022-08. DOI: 10.17487/rfc9293. URL: https://doi.org/10.17487/rfc9293 (cit. on p. 18).

[17] B. Fan, D. G. Andersen, and M. Kaminsky. "{MemC3}: Compact and Concurrent {MemCache} with Dumber Caching and Smarter Hashing". In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 2013, pp. 371–384 (cit. on pp. 17, 19, 20, 22, 34, 41, 67).

[18] K. Fraser. *Practical lock-freedom*. Tech. rep. University of Cambridge, Computer Laboratory, 2004. DOI: 10.48456/TR-579. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-579.html (cit. on pp. 2, 3, 16, 23, 25, 26, 28, 29, 55, 74).

[19] *Full Paper version of "Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way"*. URL: http://www.cs.utoronto.ca/~tabrown/debra/fullpaper.pdf (visited on 2023-09-25) (cit. on pp. 28, 45).

[20] *gprofng*. URL: https://sourceware.org/binutils/docs-2.40/gprofng.html (visited on 2023-08-07) (cit. on p. 34).

[21] M. Greenwald and D. Cheriton. "The synergy between non-blocking synchronization and operating system structure". In: *OSDI*. Vol. 96. 1996, pp. 123–136 (cit. on pp. 2, 3).

[22] T. L. Harris. "A pragmatic implementation of non-blocking linked-lists". In: *International Symposium on Distributed Computing*. Springer. 2001, pp. 300–314 (cit. on pp. 2, 4, 42, 50, 56, 59, 92).

[23] T. E. Hart et al. "Performance of memory reclamation for lockless synchronization". In: *Journal of Parallel and Distributed Computing* 67.12 (2007-12), pp. 1270–1285. DOI: 10.1016/j.jpdc.2007.04.010. URL: https://doi.org/10.1016/j.jpdc.2007.04.010 (cit. on p. 25).

[24] T. E. Hart. *Comparative performance of memory reclamation strategies for lock-free and concurrently-readable data structures*. Citeseer, 2005 (cit. on p. 28).

[25] M. Herlihy et al. *The art of multiprocessor programming*. Newnes, 2020 (cit. on pp. 2, 3, 10–13, 24).

[26] T. H. Hetherington, M. O'Connor, and T. M. Aamodt. "MemcachedGPU". In: *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015-08. DOI: 10.1145/2806777.2806836. URL: https://doi.org/10.1145/2806777.2806836 (cit. on pp. 17, 22).

[27] *Hiredis, a C Redis client*. URL: https://github.com/redis/hiredis (visited on 2022-08-25) (cit. on p. 67).

[28] S. Jiang, F. Chen, and X. Zhang. "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference*. ATEC '05. Anaheim, CA: USENIX Association, 2005, p. 35. DOI: 10.5555/1247360.1247395 (cit. on p. 8).

[29] S. Jiang and X. Zhang. "LIRS". In: *ACM SIGMETRICS Performance Evaluation Review* 30.1 (2002-06), pp. 31–42. DOI: 10.1145/511399.511340. URL: https://doi.org/10.1145/511399.511340 (cit. on p. 8).

[30] S. P. Jones. "Beautiful concurrency". In: *Beautiful Code: Leading Programmers Explain How They Think* (2007), pp. 385–406 (cit. on p. 2).

[31] R. V. Juncheng Yang Yao Yue. "A large scale analysis of hundreds of in-memory cache clusters at Twitter". In: *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 2020-11. URL: https://www.usenix.org/conference/osdi20/presentation/yang (cit. on pp. 33, 37, 70, 71, 83, 99).

[32] S. Li et al. "Architecting to achieve a billion requests per second throughput on a single key-value store server platform". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ACM, 2015-06. DOI: 10.1145/2749469.2750416. URL: https://doi.org/10.1145/2749469.2750416 (cit. on pp. 3, 17, 20).

[33] *libmemcached github*. URL: https://github.com/awesomized/libmemcached (visited on 2023-06-02) (cit. on pp. 34, 67).

[34] H. Lim et al. "{MICA}: A Holistic Approach to Fast {In-Memory}{Key-Value} Storage". In: *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. 2014, pp. 429–444 (cit. on pp. 17–22).

[35] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[36] Y. Mao, E. Kohler, and R. T. Morris. "Cache craftiness for fast multicore key-value storage". In: *Proceedings of the 7th ACM european conference on Computer Systems*. ACM, 2012-04. DOI: 10.1145/2168836.2168855. URL: https://doi.org/10.1145/2168836.2168855 (cit. on pp. 17, 20, 22, 34, 67).

[37] *Masstree Github*. URL: https://github.com/edemko/masstree (visited on 2023-01-17) (cit. on p. 22).

[38] M. McCool, J. Reinders, and A. Robison. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012 (cit. on pp. 8, 9).

[39] P. E. McKenney. "Is Parallel Programming Hard, And, If So, What Can You Do About It?(Release v2022. 09.25 a)". In: *arXiv preprint arXiv:1701.00854* (2017) (cit. on p. 12).

[40] P. E. McKenney and J. D. Slingwine. "Read-copy update: Using execution history to solve concurrency problems". In: *Parallel and Distributed Computing and Systems*. Vol. 509518. Citeseer. 1998, pp. 509–518 (cit. on p. 28).

[41] *MemC3 Github*. URL: https://github.com/efficient/memc3 (visited on 2023-01-17) (cit. on p. 22).

[42] *Memcached Github*. URL: https://github.com/memcached/memcached (visited on 2022-12-28) (cit. on pp. 1, 17, 20, 22, 32, 40, 41, 102).

[43] *MemcachedGPU Github*. URL: https://github.com/tayler-hetherington/MemcachedGPU (visited on 2023-01-17) (cit. on p. 22).

[44] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. "CPHASH: A Cache-Partitioned Hash Table". In: () (cit. on p. 18).

[45] *MICA Github*. URL: https://github.com/efficient/mica/tree/isca2015 (visited on 2023-01-17) (cit. on p. 22).

[46] M. Michael. "Hazard pointers: safe memory reclamation for lock-free objects". In: *IEEE Transactions on Parallel and Distributed Systems* 15.6 (2004-06), pp. 491–504. DOI: 10.1109/tpds.2004.8. URL: https://doi.org/10.1109/tpds.2004.8 (cit. on pp. 3, 15, 16, 25, 26).

[47] M. M. Michael. "Safe memory reclamation for dynamic lock-free objects using atomic reads and writes". In: *Proceedings of the twenty-first annual symposium on Principles of distributed computing*. 2002, pp. 21–30 (cit. on p. 26).

[48] M. M. Michael and M. L. Scott. "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms". In: *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing - PODC '96*. ACM Press, 1996. DOI: 10.1145/248052.248106. URL: https://doi.org/10.1145/248052.248106 (cit. on pp. 3, 23).

[49] Nova FCT DI. *DI-Cluster Wiki*. URL: https://cluster.di.fct.unl.pt/ (visited on 2022-11-29) (cit. on pp. 33, 66).

[50] D. Ongaro et al. "Fast crash recovery in RAMCloud". In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. 2011, pp. 29–41 (cit. on p. 17).

[51] R. J. Recio et al. *A Remote Direct Memory Access Protocol Specification*. RFC 5040. 2007-10. DOI: 10.17487/RFC5040. URL: https://www.rfc-editor.org/info/rfc5040 (cit. on p. 18).

[52] *Redis' website*. URL: https://redis.io/ (visited on 2022-08-24) (cit. on pp. 67, 87, 93).

[53] A. Tanenbaum. "Modern operating systems". In: Pearson Education, Inc., 2009, p. 25 (cit. on p. 1).

[54] J. D. Valois. "Implementing Lock-Free Queues". In: (1994) (cit. on p. 24).