FEDERAL UNIVERSITY OF RIO GRANDE DO SUL
INFORMATICS INSTITUTE
BACHELOR OF COMPUTER SCIENCE

LUCAS DOS SANTOS LERSCH

A Lock-free Buffer for WattDB

Graduation Thesis

Prof. Dr. Renata de Matos Galante
Advisor

Dipl.-Inf. Daniel Schall
Coadvisor

Porto Alegre, July 2013

CIP – CATALOGING-IN-PUBLICATION

Lersch, Lucas dos Santos

A Lock-free Buffer for WattDB / Lucas dos Santos Lersch. – Porto Alegre: Graduação em Ciência da Computação da UFRGS, 2013.

54 f.: il.

Graduation Thesis – Federal University of Rio Grande do Sul. Bachelor of Computer Science, Porto Alegre, BR–RS, 2013. Advisor: Renata de Matos Galante; Coadvisor: Daniel Schall.

1. WattDB. 2. Database. 3. Buffer. 4. Non-blocking synchronization. 5. Lock-free. 6. Replacement policy. I. Galante, Renata de Matos. II. Schall, Daniel. III. Título.

“Men love to wonder, and that is the seed of science.”
— RALPH WALDO EMERSON

ACKNOWLEDGEMENTS

First of all I would like to thank my parents Suzana Barbosa dos Santos and Nelson Strohmeier Lersch for have invested hard in my education. I would like also to thanks all my family and relatives for all the support, time and attention during these past years.

I would like to thank the Informatics Institute of UFRGS for providing an education of excellence to all its students. Thanks for all employees, professors and students that work hard to make it one of the best computer science courses of Brazil. In special I would like to acknowledge Prof. Dr. Raul Weber and Profa. Dr. Taisy Weber for all the incondicional attention and support to all the students from the Informatics Institute. I would like to thank Profa. Dr. Renata Galante for having accepted to advisor this work at UFRGS.

Thanks to the Technische Universität Kaiserslautern, for offering the opportunity of this exchange program with UFRGS and for receiving all the students with open arms. I would like to thank also the ISGS office and in special Dipl.-Math. Arthur Harutyunyan for guiding and supporting the foreign students in the TU Kaiserslautern. Special thanks also to Prof. Dr. Theo Härder for receiving students from UFRGS in the Database and Information Systems group and to Dipl.-Inf. Daniel Schall for his patience and kindness on advising my work during the year spent in Kaiserslautern.

Finally, I would like to thank all my friends and colleagues that shared with me all the difficulties and glories from the past years. I am grateful for having such good friends that I can count on hard times and laugh together on good times.

CONTENTS

LIST OF ABBREVIATIONS AND ACRONYMS

CPU	Central Processing Unit
DBMS	Database Management System
FIFO	First-in First-out
LRU	Least Recently Used
CAS	Compare And Swap
OS	Operating System

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

The current approach to improve CPU performance is to focus on a higher parallelism, with multiple core processors. In highly concurrent environments, classical mutual exclusion locks to deal with concurrent access to shared data may present serious limitations and scalability issues. As an alternative, lock-free synchronization offers robust performance and avoids problems related to blocking techniques like deadlock, priority inversion and convoying. Concurrency in a database buffer is an important aspect in terms of providing processor scalability. This work provides a lock-free buffer implementation for WattDB using non-blocking synchronization techniques. WattDB is a locally distributed database system that runs on a cluster of lightweight nodes. It aims to balance power consumption proportionally to the system's load by dynamically powering its nodes individually up and down.

Keywords: WattDB, database, buffer, non-blocking synchronization, lock-free, replacement policy.

RESUMO

A atual abordagem para melhorar o desempenho de CPU é focar em um alto paralelismo, com processadores de múltiplos núcleos. Em ambientes altamente concorrentes, mecanismos clássicos de exclusão mútua para tratar acesso concorrente a dados compartilhados podem apresentar sérias limitações e questões de escalabilidade. Como uma alternativa, sincronização *lock-free* oferece um desempenho robusto e evita problemas relacionados a técnicas bloqueantes, como *deadlocks*, inversão de prioridade e *convoying*. Concorrência em uma cache de banco de dados é um importante aspecto em termos de proporcional escalabilidade no processador. Este trabalho proporciona uma implementação de uma cache *lock-free* para WattDB, usando técnicas de sincronização não-bloqueantes. WattDB é um sistema de banco de dados localmente distribuído. Seu objetivo é balancear proporcionalmente o consumo de energia com a carga de trabalho do sistema, dinamicamente ligando e desligando seus nodos individualmente.

Palavras-chave: WattDB, database, buffer, non-blocking synchronization, lock-free, replacement policy.

1 INTRODUCTION

This chapter provides an introduction for this bachelor's thesis, which is the design and implementation of a lock-free database buffer. The motivation of this work will be discussed. The objectives will be listed and defined. Finally, the organization of the text will be detailed.

1.1 Motivation

Nowadays, even if CPU manufacturers still deliver microprocessors with an increasing number of transistors, clock rates stabilized somewhere around 3 Ghz. The current approach to improve performance is to focus on a higher parallelism, i.e, multiple core processors and multiple threading units. This recent hardware trends toward multithreading have raised critical challenges in software engineering.

In parallel programming, whenever you need to share data, the classic approach is to serialize access to it. The most common synchronization technique is to use a mutual exclusion lock. However, since there is no limitation for what can be done while a mutex is locked, lock-based shared objects tend to suffer significant performance degradation when faced with the inopportune delay of the thread holding the lock. In such cases, other active threads that need access to the locked shared object are prevented from making progress until the lock is released by the delayed thread. In the worst case scenario, the thread holding the lock might want to access some other piece of shared data and attempt to lock its mutex. If another thread has already locked that last mutex and wants access to the first mutex, both of the threads will hang in a "deadlock" situation.

As an alternative to mutual exclusion lock, lock-free objects are inherently immune to problems like priority inversion, convoying and deadlock. Lock-free synchronization also offers robust performance, even with indefinite thread delays and failures. These techniques make use of a precious small set of things that you can do atomically, limitation that makes lock-free programming way harder.

In this context of parallel programming, the concurrent access to the buffer manager of a database is major factor that prevents database scalability to processors. Concurrency in a database buffer should be considered an important aspect in the scenario where the buffer operations become CPU-bound tasks rather than I/O-bound. In this case, a naive implementation for handling concurrency may become particularly problematic in multiprocessor systems. Suppose concurrent requests from multiple users. If one request provoke a page fault and it holds an exclusive lock, the exclusive lock is going to prevent the other threads from holding either a shared or an exclusive lock. Since system-wide mutexes tend to appear for each scan of pages, it would cause "mutex ping-pong" in multiprocessor and multi threaded environments. Moreover, high traffic access to a lock may causes the convoy phenomenon. In order to deal with such problematic scenarios, lock-free synchronization emerges as an alternative to classical mutual exclusion locking techniques.

1.2 Objective

This work provides the implementation of a database buffer based on non-blocking synchronization techniques and lock-free programming for the WattDB project. Since the fix and unfix operations on a buffer frame are among the most frequent basic operations of a DBMS, the efficiency of these operations is extremely important in order to build an efficient database system. Considering that the buffer is a focus of concurrency, the use of non-blocking techniques for synchronization aims at improving performance scalability in environments with large-scale multithreaded processors and highly concurrency. To achieve this behaviour, non-blocking techniques are used to avoid critical sections contentions often caused by priority inversion and convoying, which are typical problems of high concurrent environments. The lock-free buffer is achieved by combining the use of a non-blocking replacement policy and lock-free data structures.

1.3 Organization of the Work

The remainder of this work is organized as follows: Chapter 2 discusses the basic concepts used in this work and required for the full understanding of it. Chapter 3 presents important aspects, such as algorithms and data structures, that should be considered in the design of a lock-free buffer. Chapter 4 shows how the buffer is implemented, starting from a typical locking scheme and introducing changes in order to make achieve a non-blocking synchronization. Chapter 5 presents the results achieved by the buffer with respect to tests performed in a simulation environment. Finally, conclusions and remarks are outlined in Chapter 6.

2 THE CONCEPTS OF LOCK-FREE BUFFER

This chapter aims to review and define important concepts required for the clear understanding of the work. At first we are going to give a brief overview of databases and its hierarchical architecture. We are going to define a database buffer and some of the most known replacement policies. Later the WattDB project is introduced with a simple motivational scenario. Finally we are going to present the concepts related to non-blocking algorithms, its motivation and terminologies.

2.1 Database

In the modern history of computing, data has shown to be more valuable than hardware itself. Be it for business or research purposes, data plays a major role in scientific experiments, analysis of results and decision making. The concept of databases arrived with the need to store, analyse and manage all this data in a meaningful way and to fill a long list of requirements for each different kind of application. During the years it became a research area inside computation itself, closely related to the industry, where the efforts were focused on developing new technologies to attend more and different requirements imposed by the real world problems.

In a simple way, a database can be described as an organized collection of data to model relevant aspects of reality. A database management system (DBMS) is a specially designed application that interact with the used, other application, and the database itself to define, create, query and update databases. In the context of databases there is the concept of transaction, which is a unit of work performed within a DBMS against a database. Most modern DBMS offer a set of properties defined in (Haerder e Reuter 1983) known as ACID (atomicity, consistency, isolation, durability) properties. By providing the ACID properties, a DBMS guarantees the transactions to be processed in a reliable and isolated way.

Many alternatives were proposed to represent and model a database, being one of the most popular and widely adopted the one known as relational databases. A relational DBMS represents a database based on the relational model (Codd 1983), in such way that data is described and organized as a collection of tables of data items. By using a query language, it is possible for the user to describe queries into a database and retrieve meaningful information. With the popularization of relational DBMS there was the need of a good architectural model to design such a system.

The model described in (Haerder 2005) proposes a hierarchical architecture model in an attempt to design a DBMS which offers an appropriate application programming interface (API) to the user and whose architecture is open for permanent evolution. This model is based on successive data abstractions composing a five-layer model. Each layer, from bottom to top, is responsible to introduce a level of abstraction in a way that basic objects become more complex allowing more powerful operations and being constrained by a growing number of integrity rules. The uppermost level enables the user to access data stored in a meaningful way interpreted by the DBMS. An overview of the five-layer hierarchical model is shown at Table 2.1.

	Level of abstraction	**Objects**	**Auxiliary mapping data**
L5	Nonprocedural or algebraic access	Tables, views, tuples	Logical schema description
L4	Record-oriented, navigational access	Records, sets, hierarchies, networks	Logical and physical schema description
L3	Record and access path management	Physical records, access paths	Free space tables, DB-key translation tables
L2	Propagation control	Segment, pages	DB buffer, page tables
L1	File management	Files, blocks	Directories, VTOCs, etc

Table 2.1: Description of the DBMS mapping hierarchy. Extracted from (Haerder 2005).

2.2 Buffer

As seen in the previous section, the architectural design of a database can be divided in a hierarchical layered structure. In this work we are not going to get in details of each layer, its abstractions, purposes or objects. Instead, we are going to focus in an important concept present in the described model: the buffer. However, if we consider the bottom-up hierarchy of the model, it is important to give an overview of the layers below the layer 2, in which the buffer is present.

The bottommost layer, File Management, is responsible for handling the bit pattern stored on external non-volatile storage devices. This task is frequently performed in cooperation with the operating system's file management. Through the File Management layer and its defined interfaces, it is possible to easily integrate different storage technologies into the proposed architectural model.

The Propagation Control layer offers an abstraction to the lower layer by introducing the concept of pages. While a block in the File Management layer is defined as a sequence of bits or bytes, with a certain length, stored in a storage device, a page is fixed-length continuous sequence of data that defines the smallest unit of manageable data. To make a clear distinction between a page and a block, we define that a certain page can be stored in different blocks during its lifetime in the database. At the Propagation Control layers it is also introduced the in-memory buffer that holds a fraction of the pages stored in the database. At this layer the DBMS can locate almost all logical page references in the buffer, reducing disk accesses related to I/O operations of physical storage devices.

As mentioned, the File Management layer frequently works in cooperation with the operating system's file management, the DBMS could also make use of the buffer defined by the operating system for I/O operations. However, there are certain reasons for the

DBMS bypass the operating system's buffer and make its own implementation. A buffer managed by the operating system do not offer certain features required by a DBMS, such as forcing a page to disk for transaction management, controlling the order of page writes to disk for recovery and the ability to control prefetching and the replacement policy based on predictable access patterns.

2.3 Replacement Policy

In this section we are going to explain the behavior of some replacement policies important for the understanding of this work.

As mentioned before, the DBMS buffer defines a portion of memory in which data pages are contained in order to provide faster access when compared to the delay of disk I/O operations. For the DBMS to operate on a page, this page must be present in the buffer memory. However, usually the memory is expensive and not big enough to hold all the pages of a database stored in disk. For this reason, whenever the buffer is full and there is a request for a page not present in the buffer, the DBMS must pick one of the buffer pages not in use in the moment and remove it from the buffer to create space for the requested page to be loaded from disk to memory.

A replacement policy is the algorithm by which the DBMS chooses which page to remove from the buffer when needed. There is a wide variety of well-known replacement policies. The choice of the most appropriate policy is important, since it directly affects performance by having a big impact on the number of disk I/O requests. It is worth noting that no policy is uniformly appropriate for typical DBMS access patterns, being necessary a careful analysis of tradeoffs to chose the most suitable policy for a certain situation.

The "First-in, First-out" (FIFO) (Tanenbaum 2007) is one of the most simple, cheap and intuitive algorithm for page replacement. The idea is to have pages in buffer organized in a queue in such a way that requested pages are pushed into the back of the queue in order of arrival. The page at the front of the queue will contain the earliest page requested and the back of the queue will contain the most recent page requested. When the buffer is full and a page needs to be removed, the page at the front of the queue is selected.

The "Second Chance" (Tanenbaum 2007) algorithm offers an improvement on the FIFO algorithm by introducing a reference bit which is set when the page is inserted into the buffer. When looking the front of the queue for a page to be removed, it first verifies the reference bit. The page is selected if the reference bit is clear. If the reference bit is set, it is then cleared and the page is pushed into back of the queue again.

Another version of the FIFO and "Second Chance" algorithm, called "Clock" (Tanenbaum 2007), keeps a circular list of the pages in buffer memory instead of a queue. A "hand" iterator indicates the last examined page in the circular list. When a page needs to be removed from buffer, the reference bit of the page indicated by the "hand" iterator is examined. If it is not set, the page is selected to be replaced. If the reference bit is set, it is cleared and the "hand" iterator is moved to the next position following a circular clock-like order. The process repeats until a page is selected to be removed.

The "Generalized Clock" algorithm (GCLOCK) (Smith 1978) is a variant of "Clock" that introduces a weight integer value instead of a reference bit. Whenever a page is requested or referenced in the buffer, the page weight is incremented. The page weight value is decremented when it is examined and the "hand" iterator pass by to the next page in the circular list. A page is selected when its weight value is zero. The next steps are similar to the "Clock" algorithm described above.

The algorithm known as "Least Recently Used" (LRU), defined in (1), uses a linked-list data structure to keep track of pages usage over a short period of time. At the back of the linked-list resides the least recently used page and at the front the most recently used page. Whenever a page is requested or referenced, it is moved to the front of the list, causing pages with less access frequencies to move towards the back of the list. When the buffer is full and a page needs to be removed, the page at the back of the list, at the least recently used position, is selected.

2.4 The WattDB Project

In todays world, with the growth of modern technology and its increasing presence in society the concept of energy efficiency is drawing more attention and interest than it did a few years ago. Energy efficiency can be defined as using less energy to provide the same service. Investments are made in the research and development of new technologies focusing on energy efficiency in a way to reduce energy consumption and consequently its related costs. It is worth noting that such concept is important from a sustainable economy point of view, making it also present in all the wide range of technology fields.

From database technologies perspective, large servers are composed by powerful multi-core processors, a great amount of memory and storage disks. These servers consume a considerable amount of energy and because of the need to maintain response and latency times low, they usually do not employ energy saving mechanisms such as standby or spin-down idle disks. Even if peak load times require fast and heavyweight hardware to guarantee performance, most of the energy will be waste during times of low load. With the energy costs steadily increasing, it is desirable to have low consumption and energy scalable systems to avoid waste of energy by providing what is called energy proportionality. The energy proportionality paradigm means a server should only consume energy proportionally to its workload-driven utilization.

The WattDB project (Haerder e Schall 2012) proposes a locally distributed database system that runs on a cluster of lightweight nodes. By switching the nodes on and off, it is possible to the server to adapt the cluster to the current workload and to consume almost no energy when being idle. The system is capable of managing the nodes in a way that the overall energy consumption will scale proportionally with the given load, making it energy proportional. An overview of the WattDB logical cluster architecture can be seen in Figure 2.1.

2.5 Non-blocking Algorithms

When developing a multithreaded application, the most common problem a developer may face is to manage the access to a shared resource by multiple threads. For years the traditional approach to solve this problem is to use mechanisms like mutexes, semaphores and monitors, for example. Such mechanisms define a portion of the code that is not executed concurrently. This portion of code is called a critical section and all the access to shared resources should be within it. Whenever a thread needs to execute a critical section, it should acquire the lock associated with this section, and if the lock is held by another thread, it should block and wait until the lock is free. All the managements of locks and threads is done by the operating system.

This technique was created and largely used in times where most of the applications were developed for a single CPU environment. Even though we had the concept of par-

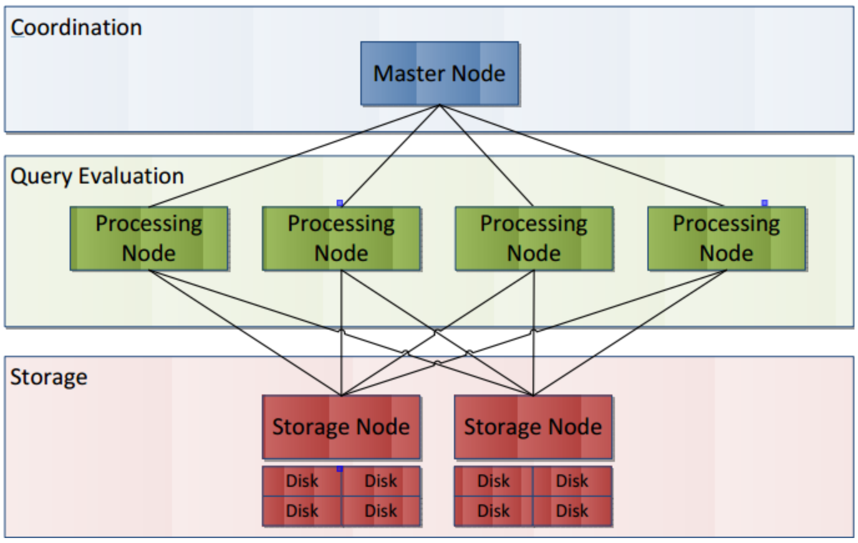

Figure 2.1: Logical architecture of WattDB cluster. From (Haerder e Schall 2012).

allelism in a single CPU by using time-sharing, nowadays we have what is called true parallelism with multicore processors environments becoming more and more common. In such environments, locking mechanisms may not be the best solution for synchronization issues. Worse than that, locks may even become a problem itself and affect the performance of a whole system in a negative way. Some alternatives have been suggested to deal with the problems introduced by locks, and one of them is simply not to use them.

In a simple way, a multithreaded algorithm is considered to be non-blocking if the execution of a thread is not indefinitely postponed by mutual exclusion mechanisms. These algorithms can be classified as:

- Wait-free: each thread completes in a finite number of steps.
- Lock-free: allow individual threads do starve, but the system as a whole make progress.
- Obstruction-free: a single thread executed in isolation for a bounded number of steps will complete its operation.

Wait-freedom is the strongest non-blocking property, guaranteeing system-wide throughput and individual thread progress. This guarantee is very hard to achieve, and in some cases the performance is even worse than most blocking approaches. It is important to note that any wait-free algorithm is also lock-free, the same way any lock-free algorithm is considered obstruction-free. In this work we are going to focus on lock-free class.

The idea of lock-free programming is not really not having any lock, but to minimize the number of locks or critical sections, by using some techniques that allow us not to use locks for most operations. The usual approach is to use hardware-intrinsic atomic operations. As a matter of fact, even locks themselves must use those atomic operations. The difference is that a lock-free program can never be stalled entirely by any single thread. Also this technique allows concurrent update of shared data structures without resorting to critical sections protected by operating system managed locks.

Atomic operations are ones which manipulate memory in a way that appears indivisible: no thread can observe the operation half-complete. On modern processors, lots of operations are already atomic. For example, aligned reads and writes of simple types are usually atomic. Other important atomic operation is the Compare-And-Swap (CAS) operation and can be seen in Code 2.1.

```
bool CompareAndSwap (address, expectedValue, newValue)
  atomically:
    load value_at(address) into oldValue
    if oldValue == expectedValue then
      store newValue at address
      return true
    else
      return false
```

Code 2.1: Semantics of Compare-And-Swap.

The semantics of this operation is defined by atomically comparing the value referenced by *address* with *expectedValue*, and if they are equal, swap the value referenced by *address* by *newValue*. The call also indicates the result of the comparison. The pattern typically involves copying a shared variable to a local variable, performing some speculative work, attempting to publish the changes to the shared variable using CAS, and retrying if the attempt failed. The following code snippet illustrates an example on how to use the defined CAS pattern.

```
do {
    // Copy a shared variable to a local.
    oldValue = shared_resource;

    // Do some speculative work, not yet visible to other threads.
    newValue = oldValue;

    //Quit if old shared_resource meets a certain condition.
    if(oldValue == CONDITION)
        break;

//Try to publish changes to the shared_resource.
} while(CAS(shared_resource, oldValue, newValue));
```

Code 2.2: Compare-And-Swap use pattern.

However, to implement the underlying mechanism to atomic operations, the CPU usually has to maintain the coherence between the individual caches of each core by using mechanisms to lock the communication bus. Figure 2.2 illustrates a environment with multiple processors, P1..Pn, in which each processor has its own cache. The shared bus works as a broadcast medium in such a way that each cache controller “snoops” the transaction on the bus. If the transaction is related to a block contained in a certain cache, its controller considers the transaction as relevant, otherwise it invalidates the transaction or supply a value in order to ensure coherence. The amount of time the data will be locked is supposed to be kept to minimum, providing a lock granularity reduced to a single hardware instruction.

The main advantages of lock-free programming is to avoid problems intrinsic to typical locks, such as contention, priority inversion and convoying. Lock-free algorithms are also kill-tolerant, meaning that the whole system will not halt if any thread is suspended indefinitely. If we consider performance issues, not using locks avoid all the overhead related to manage such structures, like process context swapping and scheduling.

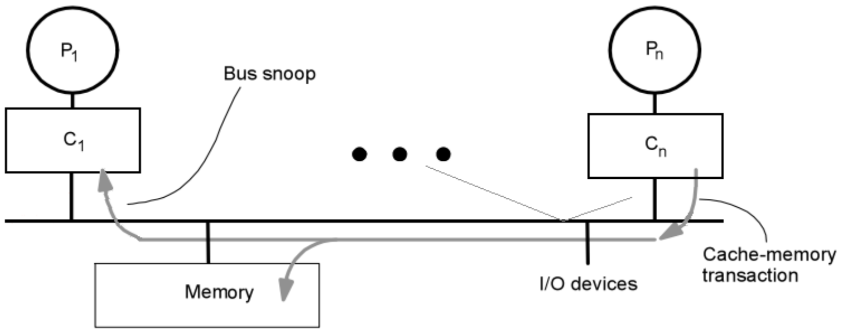

Figure 2.2: Cache coherence scheme.

It is worth noting that we will soon be facing desktop systems with 64, 128 and 256 cores. Parallelism in this domain is unlike our current experience of 2, 4, 8 cores; the algorithms which run successfully on such small systems will run slower on highly parallel systems due to contention. In this sense, lock-free is important since it is contributes strongly to solving scalability. In general lock-free programming trades throughput and mean latency throughput for predictable latency. That is, a lock-free program will usually get less done than a corresponding locking program if there is not too much contention, but it guarantees to never produce unpredictably large latencies.

However, lock-free programming is not a magic bullet, and there are some drawbacks that must be considered when choosing the best technique for a task. First of all, designing a lock-free algorithm is something non-trivial. It must handle all possible interleaving of instruction streams from contending processors. Also there are many ways to do the "atomically commit" part. In terms of performance, in a environment with high contention, it performs worse than locks because we are repeatedly doing work that gets discarded/retried. And finally, it is virtually impossible to design a lock-free algorithm that is both correct and "fair". This means that (under contention) some tasks can be lucky (and repeatedly commit their work and make progress) and some can be very unlucky (and repeatedly fail and retry).

3 DESIGNING A LOCK-FREE BUFFER FOR WATTDB

In this chapter we are going to discuss important aspects that should be considered when designing the lock-free buffer in the context of WattDB.

3.1 Design Overview

As described in (Gray e Reuter 1992), the buffer works as a mediator between the basic file system and the tuple-oriented file system. Its basic function is to move pages between a disk and the main memory. For an improved performance and faster response time, the buffer minimizes the number of disk accesses by holding and managing pages in main memory, as well as coordinating the writing of such pages to disk. It is important to point that a certain page must be present in the buffer memory so the DMBS can access it through read/write operations. WattDB operates on raw disk devices instead of using the OS file system, so there is the need of circumvent the OS buffer to minimize management overhead and also to meet the special needs of a database access patterns.

The buffer manager administers an area of shared virtual memory, which is partitioned into portions of equal size called frames. Each frame can hold exactly one page. The basic operation of the buffer manager is illustrated in the flow chart in Figure 3.1 and is described as follows:

1. A certain page is requested to the buffer manager and it checks if the page is in its memory.
2. If the page is found, the buffer manager returns the address of the frame which contains the page and operation ends here.
3. If the page is not found, the buffer manager searchs for a free frame which can hold the page to be loaded from disk to memory and returns such frame.
4. If there is no free frame, the buffer manager determines a page that can be removed from the buffer based on a replacement policy.
5. If the page to be replaced has been changed while in the buffer, the buffer manager writes it back to its block on disk, else the page can be simply overwritten by the requested page in the frame. The buffer manager returns the address of the frame.

It is important to note two basic differences between a conventional file buffer and the database buffer. First, the database buffer returns to the caller the memory address of a requested page rather than a simple copy of it. This is done because multiple transactions

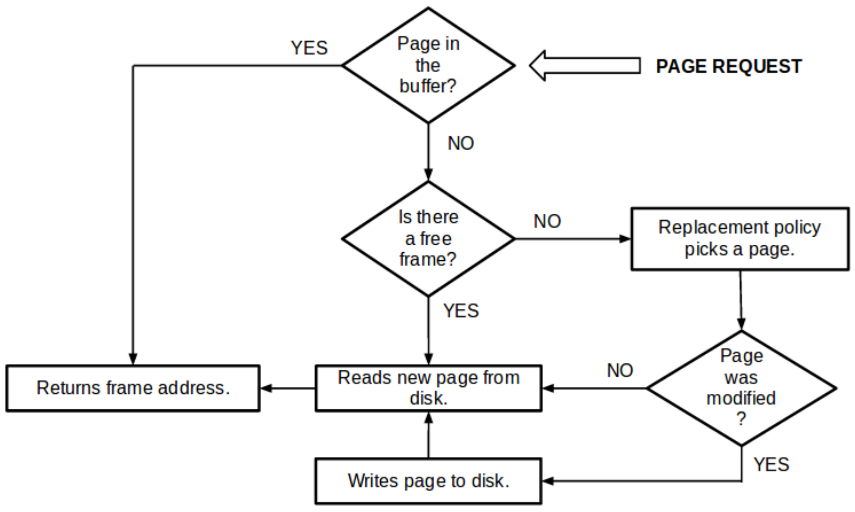

Figure 3.1: Buffer manager basic operation.

might have to access and modify the same page and if each transaction get a copy of the page, modify it and try to rewrite the updated version of the page it would generate an inconsistency because of the difference versions of the page. The second difference is that the higher modules who interacts with the buffer only have to inform if their page access resulted in an update of the page, but it is buffer manager who decides by its own criteria when the modified page is written back to disk.

3.2 Fix-Use-Unfix Protocol

The buffer manager offers its services to the higher-level layers in the hierarchical architecture by using a FIX-USE-UNFIX protocol (Gray e Reuter 1992) (Effelsberg e Haerder 1984). The interactions happen via an *fix/unfix* interface implemented by the buffer manager.

A page identifier is represented by the file to which the page belongs and the page number in the file. Whenever the caller wants to request a certain page, it must inform the page identifier to the buffer manager through a *fix* call and expect the address of the page frame as the result. However, if the caller informs the buffer manager only the page identifier, when the buffer becomes full and receives a new request for a page not present in the buffer, it will have to replace one of the pages currently stored in the buffer. This creates addressing problems by invalidating the address of the page chosen to be replaced.

The FIX-USE-UNFIX protocol defines an order of interactions that a caller should follow to access a certain page. A *fix counter* is used to represent if the status of the page is available or if it is being currently used. When a *fix* call is made, in addition to return the address of the frame which contains the page, it also increases the *fix counter* by 1. If the buffer memory is full, the request for a new page is made and the buffer manager must determine a page to be replaced, the replacement policy will not pick a page which has its *fix counter* greater than the default initial value. Thus, after the *fix* call, the caller is allowed to use the page and it has guarantees that the page frame address in the buffer memory will remain valid. When the caller is done with the page, it must explicitly make an *unfix* call on the page, which decrease the *fix counter* by 1, in order to inform the buffer manager that it no longer wants to use that page.

It is worth noting that multiple callers may access the same page at the same time, each one making its own *fix* call to a page. However, the callers must make sure that for each *fix* call there is a corresponding *unfix* call, making it eligible for replacement when no one is using it anymore. In addition, all callers using the buffer manager interface must operate strictly using the FIX-USE-UNFIX protocol. In order to avoid the situation in which the buffer is filled with irreplaceable pages, it is required that the duration between a *fix* call and its corresponding *unfix* call as short as possible. As a consequence, the *fix* and *unfix* operations are among the most frequently used primitives in a database system, and thus the efficiency of these operations is extremely important.

As we have mentioned in the previous section, the higher module which interacts directly with the buffer cannot issue write operations, so it is responsible for informing the buffer manager if the page it has access was updated. The buffer manager offers an interface for the caller to set a specific flag in the page to indicate if it is dirty, i.e, modified. When the buffer manager picks a page for replacement, it checks the dirty flag and writes the page back to disk if necessary.

3.3 Frame Pool Structure

When designing the buffer manager we should consider as the main data structure an array of frames, called here a frame pool. The frame pool has an initial fixed size, which can be modified later on for performance tuning purposes. When initializing the buffer, all frames in the pool must be made consistent by being marked as free, i.e, not holding a page.

When all frames in the pool are occupied and a request for a new page is made, the buffer manager acts directly in the pool in order to define a page to be replaced and give space for the requested new page. A procedure is invoked by the buffer manager to choose a page that is not being currently used as the victim to be replaced, based on a replacement policy.

Since the buffer manager can receive different requests from multiple callers, it must offer consistency guarantees for concurrent access. These guarantees are usually achieved by serializing access to the frame pool and other buffer structures via locking mechanisms, such as mutexes and semaphores. However, to design a lock-free buffer it is required the use of non-blocking synchronization mechanisms instead of acquiring locks. In the context of the frame pool structure, it is desired to have the replacement policy implemented as an algorithm based on lock-free programming.

3.4 Lookup Structure

When a page request is made it is required from the buffer manager a fast associative access to the frame pool to determine if the requested page is in the buffer. Since the request for a page requires that the caller informs only the page identifier, to achieve the mentioned behavior it is necessary to have an auxiliary structure to the frame pool that is associatively addressable via the page identifier. This structure usually uses a hash function and since the main purpose is to verify if the buffer contains a certain page, it is here referred as a lookup structure.

Similar to the frame pool previously described, multiple callers share the same buffer lookup structure and thus the operations on this structure must be synchronized to protected against concurrent page requests from the callers. To maintain a consistent non-blocking synchronization buffer management scheme, a lock-free data structure should be used to implement the lookup structure.

Lock-free data structures implement concurrent objects without the use of lock mechanisms. In multithreaded environments with high concurrency, these structures usually provide a more robust performance and reliability than conventional lock-based implementations. However, a common problem to lock-free programming, known as the ABA problem, is usually present in lock-free data structures algorithm and it must be handled.

3.4.1 The ABA Problem

As previously discussed in other sections, the Compare-And-Swap operation is often used in lock-free programming. Its use pattern consists of making a local copy of a shared resource, update this copy and attempting to publish the copy using the CAS operation to verify if the value of the shared resource was not modified by another thread at the same time. However, to exemplify the ABA problem let's imagine an environment with two different threads attempting to modify the same shared resource. The following scheme shows an execution path that reproduces the error.

1. The current value of the shared resource is A. The first thread is running and it makes a local copy of the shared data structure.

2. The first thread is preempted and the seconds thread starts to execute. The second thread updates the value of the shared resource firstly from A to B and then it updates the value again from B back to A before being preempted.

3. The first thread is now executing again, it attempts to publish the changes to the shared resource and succeeds because the current value is A, but it does not see the changes made by the second thread meanwhile.

Even if the first thread will continue to execute normally, it does not notice the hidden update made by the second thread. Depending on the algorithm being implemented, the ABA problem can lead to wrong assumptions and inconsistent states. It is also possible that an algorithm does not care about hidden updates, and thus it can simply ignore the ABA problem.

A common scenario of the ABA problem in the context of lock-free data structures is the implementation of a lock-free linked-list as in (Valois 1995). A thread can delete a certain entry from the list and insert a new one in the same location of the deleted entry. If we are dealing with pointers, a pointer to this new entry has the same value of a pointer to the old entry and there is no way to differentiate.

Solutions exist to work around the ABA problem. In the lock-free linked-list example, one might simply prohibit the re-use of the memory of a deleted node. However this is not a practical approach, since memory is a finite resource, no matter how big it is.

The use of garbage collection mechanisms would prevent the ABA problem, since it traces and manages the use of dynamically allocated objects by reference counting. Yet, garbage collection mechanisms are not universal and are not present and portable to multiple systems that do not offer support. Also, these mechanisms are usually not lock-free and, considering the case of failure or delay, they could prevent threads from making progress indefinitely, violating the lock-freedom property.

The use of a different version of the Compare-And-Swap operation is also a candidate for solution, as presented in (Detlefs et al. 2001). The Double-Compare-And-Swap (DCAS) operation takes two address and compares each one of them with two expected values, updating the address values with two pre-supplied expected values if the comparison matches. In addition to not being supported in many hardware architectures, (Doherty et al. 2004) demonstrates that DCAS does not provide more significant programming power to solve non-blocking synchronization problem than the simple CAS operation.

The lock-free safe memory reclamation method presented in (Michael 2004), use hazard pointers to allows the reuse of the memory of deleted nodes and provides a solution to the ABA problem for pointers to dynamic memory nodes. Each thread keeps a list of hazard pointers that are pointers to indicate which memory nodes a certain thread is currently accessing. By checking the hazard pointer list of the other threads, it is possible to determine if a certain shared resource can be safely modified without provoking the ABA problem, or not.

There are other different workarounds to the ABA problem, but the focus of this section was to present the problem and give an overview of the most known approaches.

4 IMPLEMENTATION OF THE LOCK-FREE BUFFER

This chapter discusses the implementation of buffer using non-blocking synchronization. We are going to start by presenting a typical buffer based on locking synchronization and then we are going to introduce modifications to the blocking scheme in order to make it a non-blocking scheme.

4.1 Implementation Overview

First of all it is important to note that the following implementation has some requirements. The programming language used is C++ with Gnu Compiler Collection (GCC) (GNU Project 2013) and the implementation takes place in a Linux system and x86_64 instruction set architecture. To implement non-blocking algorithms it is required a hardware architecture that offers atomic primitives like Test-And-Set and Compare-And-Swap. It is also highly desirable to have atomic load and store operations for primitive data types. The x86_64 architecture offers these guarantees. GCC also offers built-in functions for atomic memory access in order to facilitate the usage of atomic primitives, usually programmed in assembly code.

In order to demonstrate the steps taken to implement a database lock-free buffer based on non-blocking synchronization we are going to start by giving an overview of the buffer main structures, classes and methods considering the usage of classic synchronization mechanisms, as mutexes. The Figure 4.1 below shows an UML-like diagram of the classes, its attributes, methods and relations. For now, some aspects like constructors, destructors, method parameters were omitted in the diagram for simplicity.

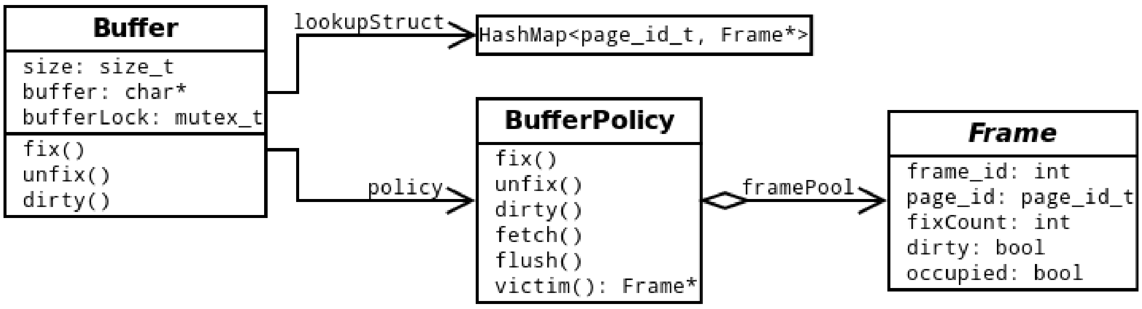

Figure 4.1: Overview of buffer diagram.

The *Buffer* class is the only entry point for a client to request the buffer services. The *Buffer* class has as attributes its in bytes, a pointer *buffer* to the beginning of a reserved portion of memory space of size *size* and the mutex *bufferLock* to synchronize access to shared portions of code during concurrent access. The portion of memory pointed by *buffer* is where the buffer is going to keep pages loaded from a disk device.

The *Buffer* class also contains an associative container as the lookup structure, which is implemented here as a hash map. Access to the lookup structure must be also synchronized, meaning that the Buffer class must lock its mutex when accessing the structure in the case that the structure does not offer this behaviour intrinsically.

The last *Buffer* class attribute to be considered is the *policy*. The *policy* attribute is here defined by the *BufferPolicy* class, which is in fact implemented as an abstract class. In order to implement a replacement policy, a new class must be define to extend the *BufferPolicy* class and overwrite its virtual methods. Thus, the abstract methods defined by the *BufferPolicy* class must be implemented by a concrete class that defines a replacement policy, since different replacement policies may have different behaviours on operations like *fix*, *unfix*, *dirty*, etc.

The *BufferPolicy* class also defines the *victim* method. This method is called everytime a new page request is made to the buffer manager, there is no available frame and the buffer manager must choose a page that is not being used to be removed from the buffer memory, in order to create space in the memory for the requested page. This method acts directly on the frame pool structure which must be defined by the class implementing a replacement policy. As mentioned in the previous chapter, the frame pool is a shared structure in the sense that multiple concurrent requests may result in a call to the *victim* method. To handle the concurrent access, the buffer must synchronize calls to the *victim* method using its mutex.

The *Frame* class has the basic structure to represent important elements of a frame. In addition to uniquely identify a frame, the *frame_id* attribute is used to access the portion of memory in which a page is stored by indexing the *buffer* attribute of the *Buffer* class which points to the beginning of the buffer memory. The *page_id* attribute is used simply to identify which page the frame currently holds. The *fixCount* attribute indicates whether the page is being used, is eligible for replacement or is being brought into the buffer by another concurrent thread. The Figure 4.2 shows the possible states of a frame concerning the *fixCount* attribute. The *dirty* and *occupied* flags are used to indicate if the page was modified while in the buffer or if the frame is currently free, i.e, not holding a page, respectively.

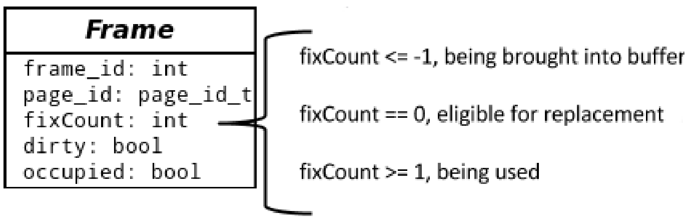
Frame
frame_id: int
page_id: page_id_t
fixCount: int
dirty: bool
occupied: bool
fixCount <= -1, being brought into buffer
fixCount == 0, eligible for replacement
fixCount >= 1, being used

Figure 4.2: Frame possible states.

Finally, the buffer manager offers its services to the clients by the *fix*, *unfix* and *dirty* methods of the *Buffer* class. A client willing to access a page must make a call to the *fix* method passing as arguments the *page_id* and a pointer which is going to be used for the output of the requested page address in the buffer memory space. After using the page via the address returned in the pointer parameter, the client must call the *unfix* method to inform it is no long using the page. If the client modified the page, it must call the *dirty* method before calling the *unfix* method in order to let the buffer manager know the page was modified and take the required measures.

The diagram in Figure 4.3 shows a more detailed version of the previous diagram, implementing the policy Least Recently Used.

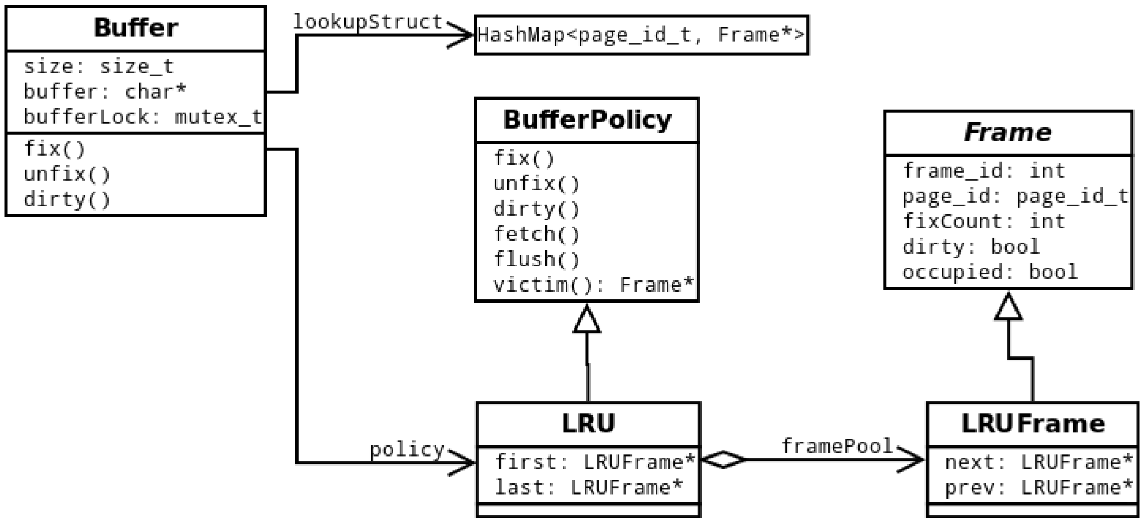

Figure 4.3: Overview of buffer diagram with LRU replacement policy.

The *LRU* class defines the *framePool* attribute as a linked-list of *LRUFrame*. The *first* attribute is a pointer to the head of the list, which has the most recently used frame. Analogously, the *last* attribute is a pointer to the tail of the list, which has the least recently used frame. The *LRU* class must overwrite the methods of the *BufferPolicy* class in order to move a page to the head of the list every time it is accessed. It must also implement the *victim* method to determine a page to be replaced, returning its frame. The *victim* method has to search the linked-list backwards, checking for eligible frames and return the first found. If no eligible frame is found, it returns a NULL pointer which should be interpreted by the caller of the victim method. Since a mutex is used, search and modification operations to the linked-list can be issued without worrying with further synchronization problems. Code 4.1 shows the pseudocode for the behaviour of moving a frame to the head of the list when it is used and Code 4.2 shows the pseudocode for the *victim* method.

```
void LRU::use(Frame *f) {
  lock(&bufferLock);
    LRUFrame *frame((LRUFrame *) f);
    if(frame == first) {
      unlock(&bufferLock);
      return;
    }
    if (frame->prev) {
      frame->prev->next = frame->next;
    }
    if (frame->next) {
      frame->next->prev = frame->prev;
    }
    if (last == frame) {
      last = last->prev;
    }
    frame->next = first;
    frame->prev = NULL;
    first->prev = frame;
    first = frame;
  unlock(&bufferLock);
  return;
}
```

Code 4.1: Method for moving the frame to the head of the list.

```
Frame* LRU::victim() {
  lock(bufferLock);
    for (LRUFrame *frame = last; frame != 0x0; frame = frame->prev) {
      if (frame->fixCount == 0) {
        frame->fixCount = -1; /* page is being brought */
        unlock(bufferLock);
        return frame;
      }
    }
  unlock(bufferLock);
  return NULL;
}
```

Code 4.2: Method victim() to determine a page to be replaced.

4.2 Nb-GCLOCK: A Non-blocking Replacement Policy

The implementation of a lock-free replacement policy is needed in order to provide concurrent access to the frame pool structure without locking the whole structure. We are going to implement a replacement policy based on the non-blocking version of the GCLOCK algorithm (Nb-GCLOCK) introduced by Makoto Yui in (Yui et al. 2010).

Since the *BufferPolicy* class *victim* method acts directly on the frame pool structure, it must handle the frame attributes. As seen in the previous section, the main attribute used by the replacement policy to determine a victim is the *fixCount* attribute. Once we do not want to locks, there is the need of basic non-blocking operations to manage the frame *fixCount* attribute state in an atomic way. We then modify the previously introduced *Frame* class to declare the *fixCount* attribute as volatile. In C++ the volatile keyword in a variable declaration introduces a guarantee that no reordering will occur between reads and writes of this variable. The effect is the same of introducing a memory barrier to indicate that no memory access will be reordered across the barrier point. We also introduce two methods to modify the *fixCount* attribute using the CAS operation supported by GCC atomic builtins. From now on all operations on the *fixCount* attribute must be through these methods. Figure 4.4 shows the redefined Frame class. The *pin* and *unpin* methods are used to increment and decrement the *fixCount* attribute, respectively. Because we are using the instruction set architecture `x86_64`, and since the *fixCount* attribute is represented as an integer, we assume that load and store operations are also atomic. Furthermore, it is required a *weightCount* attribute in the *Frame* class to store the weight of each frame needed by the GCLOCK algorithm. The code for the *pin* and *unpin* methods can be seen in Code 4.3.

```
bool Frame::pin() {
  int x;
  do {
    x = fixCount;
    if(x <= -1)
      return false;
  }while (!__sync_bool_compare_and_swap(&fixCount, x, x+1));
  return true;
}

void Frame::unpin() {
  __sync_fetch_and_sub(&fixCount, 1);
}
```

Code 4.3: Pin and Unpin methods to set fixCount value

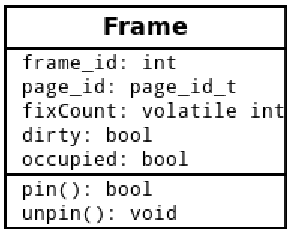

Figure 4.4: Redefined Frame class.

The *pin* method is used to indicate that the corresponding frame is being requested. It attempts to increment the frame *fixCount* attribute if its value is equal or greater to 0, meaning that the frame is available or another thread is also using it, respectively. If the current value of the *fixCount* attribute is -1 or less, it means that the frame is currently being set and there is a page being brought to buffer by another thread, so it is not available for use. Once the method is called, it attempts to atomically increment the frame *fixCount* attribute until it succeeds, returning true, or until another thread begin the process of setting a frame, making it unavailable and returning false.

Since the buffer must follow the FIX-USE-UNFIX protocol, we assume that every call to the *unpin* method is made after a corresponding call to the *pin* method, thus we can simply atomically decrement the value of the *fixCount* attribute without worrying with further guarantees.

By using the previous described structures and primitives, we can modify the *Buffer* class victim method to determine an eligible page to be replaced without the need of blocking synchronization. The frame pool structure here is seen as a circular linked-list. The replacement policy requires a *handIterator* attribute to mark the frame from which the algorithm is going to start to search for victims.

The idea is to iterate through the circular linked-list of frames, starting by the frame pointed by the *handIterator* attribute, decrement and check the *weightCount* attribute of each frame to verify if it is 0 or less. In this case, the page has been in the buffer memory long enough and if the page is not being used currently, it sets the *fixCount* attribute to -1 meaning that this frame is going to be used to load the new request page. Otherwise, if after decrementing the *weightCount* attribute the page is still being used, it simply steps to the next frame in the circular linked-list until it finds a suitable victim. In order to introduce a less drastic stop condition than simply looping through the circular linked-list until it finds a victim, we define that each call to the *victim* method will iterate through all the elements only once, returning a NULL pointer if no victim was found. This NULL return value should be interpreted by the caller of the *victim* method as there is no eligible victim at the current time, and it can take actions like thrown an exception or move the current thread to the end of the execution queue in order to give other threads a chance.

The Code 4.4 illustrates the pseudocode of the non-blocking *victim* method as well as the auxiliary *moveClockHand* method.

```
Frame* NbGCLOCK::victim() {
  unsigned int numpinning = 0;
  const int start = this->hand;

  for(unsigned int i=start%size;;i=(i+1)%size) {
    NbGCLOCKFrame* const frame = &(this->frames[i]);
    const int pincount = frame->fixCount;

    if(pincount != 0) {
      if(++numpinning >= size) {
        return 0x0;
      }
      continue;
    }

    if(__sync_sub_and_fetch(&(frame->weightCount), 1) <= 0) {
      if(__sync_bool_compare_and_swap(&(frame->fixCount), 0, -1)) {
        frame->weightCount = 0;
        this->moveClockHand(i, start);
        return frame;
      }
    }
  }
}

void NbGCLOCK::moveClockHand(int curr, int start) {
  int delta;
  if(curr < start) {
    delta = curr + size - start + 1;
    }
  else {
    delta = curr - start + 1;
  }
  __sync_fetch_and_add(&(this->hand), delta);
}
```

Code 4.4: Victim and moveClockHand methods.

By using an algorithm based in GCLOCK, every time a frame is used, all we have to do is to atomically increment the *weightCount* attribute of this frame, in contrast with LRU where it would be necessary to move the frame to the head of the linked-list which could provoke a reordering of other frames. The Code 4.5 shows the method to be called everytime a page is accessed.

```
void NbGCLOCK::use(Frame *f) {
  NbGCLOCKFrame *frame((NbGCLOCKFrame *) f);
  __sync_fetch_and_add(&(frame->weighCount), 1);
}
```

Code 4.5: Method for incrementing the weightCount

The following Figure 4.5 illustrates the buffer diagram with the *Frame* class modifications and the *NbGCLOCK* policy class.

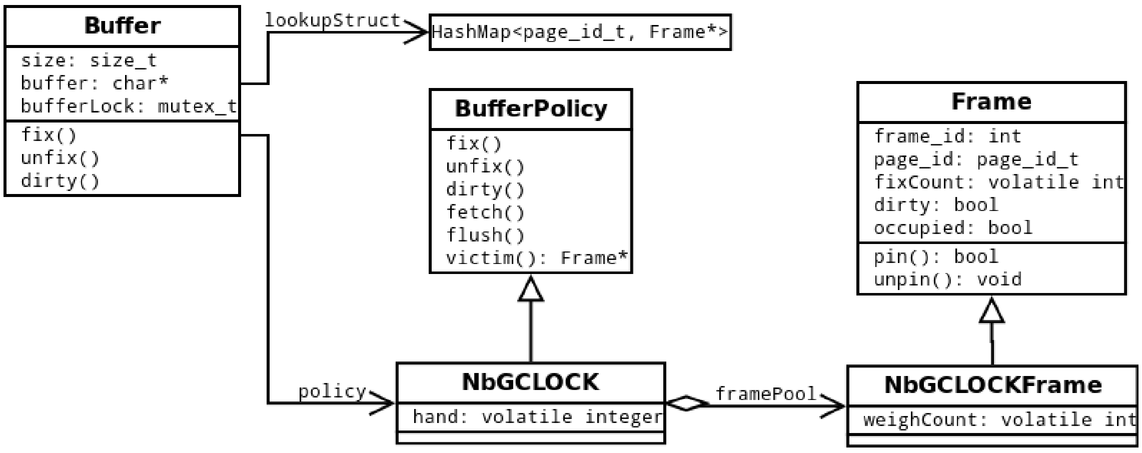

Figure 4.5: Overview of buffer diagram with Nb-GCLOCK replacement policy.

4.3 A Lock-free Hash Map

As mentioned in the previous chapter, the buffer manager needs a lookup structure with associative access to the frame pool to quickly determine if a requested page is in the buffer memory space. A good choice is a hash map structure that maps a *page_id* to the address of the frame holding this page in the frame pool. By having the address of the frame it is possible to make any operations necessary by using its pointer. The hash map is implemented as a container and it should offer the basic operations of searching, inserting and deleting an entry.

Usually generic containers are already implemented by external sources like the C++ Standard Template Library(STL) or by the GCC compiler itself as part of the __gnu_cxx Namespace. Even if it is likely that concurrent operations on a hash map will access disjoint locations, meaning that a situation where multiple threads are competing for the exactly same node is very rare, there is still need for synchronization. None of these two mentioned implementations are thread-safe and thus it is responsibility from the external agent accessing the hash map to synchronize the access.

The Intel Thread Building Blocks library (TBB) (Intel 2013) offers an implementation of a concurrent_hash_map. However this implementation still uses inner blocking synchronization mechanisms for each group of one or more hash map entries. Once an entry is retrieved from the structure by a thread, this thread holds and implicit lock in order to guarantee synchronized access. When a hash map entry needs to be accessed it must be specified if the access is going to be for a read-only or write operation. In the case of a read-only access, multiple threads can access the same entry simultaneously, but if it is a write access the execution of each thread is serialized. Furthermore, the TBB concurrent_hash_map does not allow the delete of an entry even if the current thread holds the write-lock of this entry. The delete operation is a “blind operation”, in the sense that it simply deletes an entry from the structure with no further guarantees. The caller can never be certain that erasing an entry from the structure is a good idea, because there is no guarantee that another thread did not add something important to that same entry in between the time the caller decided the entry was erasable and the time the caller actually erased it. Since it is not possible for a thread to erase an entry that it has access to (i.e. has a write-lock on), there is a race condition between the thread fetching the contents of the hash map entry and the thread removing that entry from the hash map.

Finally, an indeed lock-free hash map data structure is proposed in (Michael 2002). The drawback of this algorithm is that it requires a memory management method in order to avoid the ABA problem when the re-use of dynamic memory nodes is made necessary. Libcds (Khizsinsky 2013) implements this lock-free hash map algorithm and it also provides as memory management method a safe memory reclamation method based on hazard pointers. Although it is required that every thread accessing the hash map structure registers with the safe memory reclamation system, the overhead introduced by this should not be significant.

4.4 Buffer Fix Method Implementation

Assuming Nb-GCLOCK as replacement policy and the lock-free hash map structure provided by libcds, we are going to present the pseudocode for the *Buffer* class *fix* method. The code for the *unfix* method and *dirty* method are presented at the end as well. Since we defined that the buffer manager operates on a FIX-USE-UNFIX protocol, the *unfix* method and *dirty* method are trivial because we assume they always happen after a fix call and then we can safely operate on a frame without worrying so much about consistency of concurrent operations.

```
void Buffer::fix(const page_id_t& page_id, Page*& page) {
  Frame* frame;
  for (;;) {
    /* STEP 1: look for page in HashMap */
    functor f;
    const bool miss = !(HASH_MAP.find(page_id, cds::ref(f)));
    frame = miss ? 0x0 : f.value; /* f.value is the Frame* */

    /* STEP 2: Check if the page_id was found in the buffer.
     * Try to increment fixCount to see if the frame is
     * available and to guarantee that the frame wont be
     * picked by the victim() method in another thread. */
    if (miss || !(frame->pin())) {
      /* MISS */

      /* STEP 3: Get a free victim from the frame pool. */
      Frame* const victim = policy->victim();

      /* STEP 4: Verify if a free victim was found. */
      if (victim == NULL) {
        sched_yield(); /* If not found, yield the current thread */
        continue;
      }

      /* STEP 5: Verify if the victim was previously
```

```
 * occupied and if it needs write-back. */
if (victim->occupied) {
  if (victim->dirty) {
    write page back to disk
    victim->dirty = false;
  }
  victim->occupied = false;

  /* If the victim was occupied, remove the previous
   * association from the map. */
  HASH_MAP.erase(victim->page_id);
}

/* STEP 6: Try to insert the new association into the map. */
const bool inserted = HASH_MAP.insert(page_id, victim);

/* STEP 7: If it was not inserted this means that there was
 * a previous association. */
if (!inserted) {
  /* Get previous association */
  HASH_MAP.find(page_id, cds::ref(f));
  Frame* const prevFrame = f.value; /* f.value is the Frame* */

  /* I am not going to use this victim.
   * So make it available for other threads in the victim()
   * method. */
  victim->page_id.container = -1;
  victim->page_id.page = -1;
  victim->occupied = false;
  victim->dirty = false;
  victim->fixCount = 0; /* Until here victim->fixCount is -1. */

  /* Test pin() first to guarantee the prevFrame
   * wont be picked by victim(). */
  if(prevFrame->pin()) {
    /* Test if page_id is the same, in case it
     * was modified by another thread. */
    if(prevFrame->page_id == page_id) {
      frame = prevFrame; //Good, I use prevFrame
      page = (Page*) &buffer[frame->frame_id * PAGE_SIZE];
      page->getHeader().frame = frame;
    }
    else { /* page_id was modified meanwhile. */
      __sync_fetch_and_sub(&(prevFrame->fixCount), 1); //undo pin()
      continue; //Try again
    }
  } else { //pin() failed
    //Try again
    continue;
  }
```

```
      } else { /* Association was successfully inserted!*/
        /* I use the victim */
        frame = victim;
        frame->page_id = page_id;
        frame->occupied = true;

        page = (Page*)&buffer[frame->frame_id * PAGE_SIZE];
        page->getHeader().frame = frame;
        page = load page from disk

        frame->fixCount = 1; /* until here victim->fixCount is -1 */
      }
      break;
    } /* END if (miss || !(frame->pin())) */
    else {
      /* Possible HIT */
      if(frame->page_id != page_id) {
        /* but page_id was modified meanwhile */
        __sync_fetch_and_sub(&(frame->fixCount), 1);
        continue;
      }
      else {
        /* HIT indeed */
        page = (Page*)&buffer[frame->frame_id * PAGE_SIZE];
        page->getHeader().frame = frame;
        break;
      }
    }
  } /* END for(;;) */

  /* Call for policy fix event */
  policy->fix(frame);
}
```

Code 4.6: Buffer non-blocking fix method.

```
void Buffer::unfix(const page_id_t& page_id, Page*& page) {
  /* Page MUST BE in map, because fix() was
   * called before. */
  Frame* const frame = page->getHeader().frame;
  __sync_fetch_and_sub(&(frame->fixCount), 1);
  policy->unfix(frame);
}
```

Code 4.7: Buffer non-blocking unfix method.

```
void Buffer::dirty(const page_id_t& page_id, Page*& page) {
  Frame* const frame = page->getHeader().frame;
  frame->dirty = true;
  policy->dirty(frame);
}
```

Code 4.8: Buffer non-blocking dirty method.

5 LOCK-FREE BUFFER RESULTS

In this chapter, experiments and results achieved by different implementations of the buffer manager will be compared and discussed.

5.1 Experiments Workloads

At the time of this work, complex query capabilities were not yet completely implemented in WattDB and therefore typical benchmark tools of OLTP queries were not available. In order to run the experiments, we created artificially workloads generated to simulate page requests to the buffer manager. These workloads were based on the access pattern of the TPC-C benchmark (TPCC 2010) , which is used to measure the performance of online transaction processing (OLTP). The trace of pages requests issued during the TPC-C benchmark were used to generate the initial workload.

In order to create a more realistic scenario, this initial workload was modified to create a mixed workload. It was introduced artificial processing to each requested page, as well as a chance for a page being accessed to be modified, creating the need to write this page back to disk before it is removed from the buffer memory space. Furthermore, the initial workload was also modified to include a chance of occasionally start scans of sequential pages in order to simulate online analytical processing queries (OLAP).

Databases in general make use of indexes for improving the speed performance of data retrieval operations on tables. An index is a data structure that stores values of columns of a table, usually sorted, and is used for fast lookup of these values, since it reduces the number of records/rows in a table that need to be examined. Indexes use additional storage space, and so it is not advisable to have an index on every column. In order to provide access to other values of a certain row, an index stores a reference to the page address where this record is stored, in addition to the value of the selected column to create the index.

It is important to point that indexes have direct influence on the behaviour of the access patterns which are simulated by the created workload. Assuming that indexes are created, most of the queries will make use of them to search for the data page of a record rather than scanning each row in a table. Since indexes are data structures also stored in the database, this means that indexes pages tend to have an access frequency much higher than normal data pages, even if there are a lot less indexes pages when compared to data pages in a database. The Figure 5.1 illustrates this scenario where the “hot pages” at the top of the pyramid represent the most accessed pages, i.e, the indexes pages.

Finally, the goal of these experiments is to compare the algorithms and data structures proposed that use non-blocking synchronization mechanisms with classical approaches. As previously mentioned in this work, one of the advantages of lock-free programming

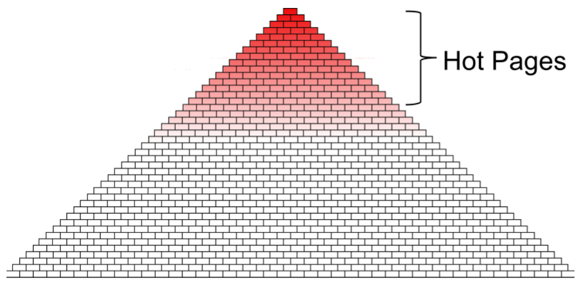

Figure 5.1: A small amount of pages are the most accessed pages.

is the improvement in scalability in multicore and high concurrent environments. To simulate this and to make the simulation more CPU-bound than I/O-bound, we introduced a set of threads, each one representing a database client running its own workloads.

5.2 Experiments Environment

The experiments were executed in the environment, which is described in the following Table 5.1.

Operating System	Ubuntu 10.04 LTS
Kernel Version	Linux 3.2.0-38-generic
Cores (#Threads/Core)	2(2)
CPU Frequency	2.13 GHz
Main Memory	4 GB
Disk	7200 RPM
L2 Cache Per Core	256 KB
Database Size	32 GB
Database Page Size	8 KB

Table 5.1: Description of environment used to run experiments.

5.3 Experiments Results

In this section we are going to show, compare and discuss results of experiments concerning implementations of a database buffer.

5.3.1 Hit Rate

The following experiment was executed to compare the buffer hit ratio of three different approaches. On the first one and second approach, the buffer uses LRU and GCLOCK as replacement policy, respectively. On both of them the lookup structure is a hash map and the access to it is synchronized by typical mutual exclusion locks. The third approach has the Nb-GCLOCK algorithm for replacement policy and uses a lock-free hash map implemented by libcds. The experiment used 32 threads concurrently executing their own workload. For each approach the experiment was executed 30 times and the average is compared in the Figure 5.2 below.

When comparing Nb-GCLOCK and the classical GCLOCK algorithm for replacement policy, we notice in the comparison chart that the hit rates show a similar tendency,

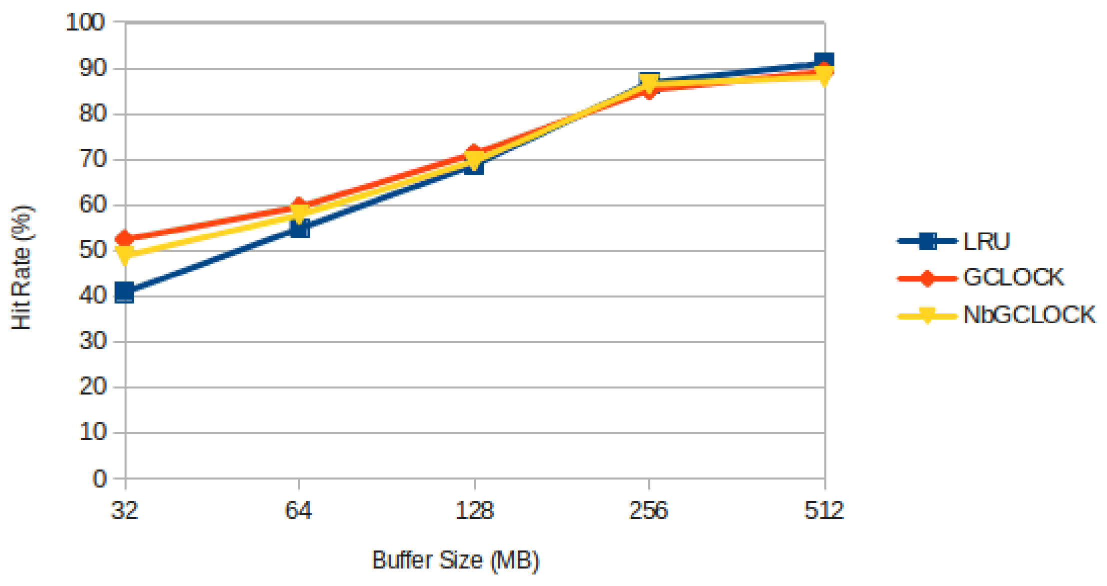

Figure 5.2: Comparison of Hit Rate when buffer size increases.

and this happens basically because Nb-GCLOCK is totally based on GLOCK. The small discrepancy between the hit rate of LRU and GCLOCK/Nb-GCLOCK at low buffer sizes happens probably because of the sequential scans that happen casually. Since LRU is based on the time frequency of pages, when a scan occurs it swaps a considerable amount of pages from the buffer, starting from the least recently used one, but affecting also pages that were recently used. GCLOCK and Nb-GCLOCK are tolerant to this behaviour because the weight counter of each page guarantees that pages being accessed more often do not get swapped out from buffer by pages that will be accessed only once, i.e, while LRU respects only recency of pages, the weight counter guarantees that GCLOCK and Nb-GCLOCK will take frequency into account, as well. However, we can notice that when the buffer size increases and it has enough capacity, this behaviour is less harmful to the hit rate and the differences are minimized.

5.3.2 Execution Time

The following experiment compares the execution time when we fix the buffer capacity and vary the number of threads, i.e, concurrent clients, executing the workloads. Again we compare the three different versions with LRU and GCLOCK using a hash map synchronized by mutual exclusion locks and Nb-GCLOCK using the lock-free hash map implemented by libcds. The buffer size was fixed in 32 Megabytes. For each approach the experiment was executed 30 times and the average is compared in the Figure 5.3 below.

Comparing LRU and GCLOCK, we can notice that from the beginning, with only 16 threads, there is no big difference in the execution time. As the number of thread increases, so does the difference between the execution time of LRU and GLOCK. GCLOCK performs better than LRU because the algorithm has a lower overhead and provides a lower probability of lock contention. Consequently it is expected to have a better performance in concurrent environments. Now analyzing Nb-GCLOCK combined with a lock-free hash map, we can verify that it scales much better than the other two approaches. Considering that a buffer manager is also I/O dependent, this behaviour is

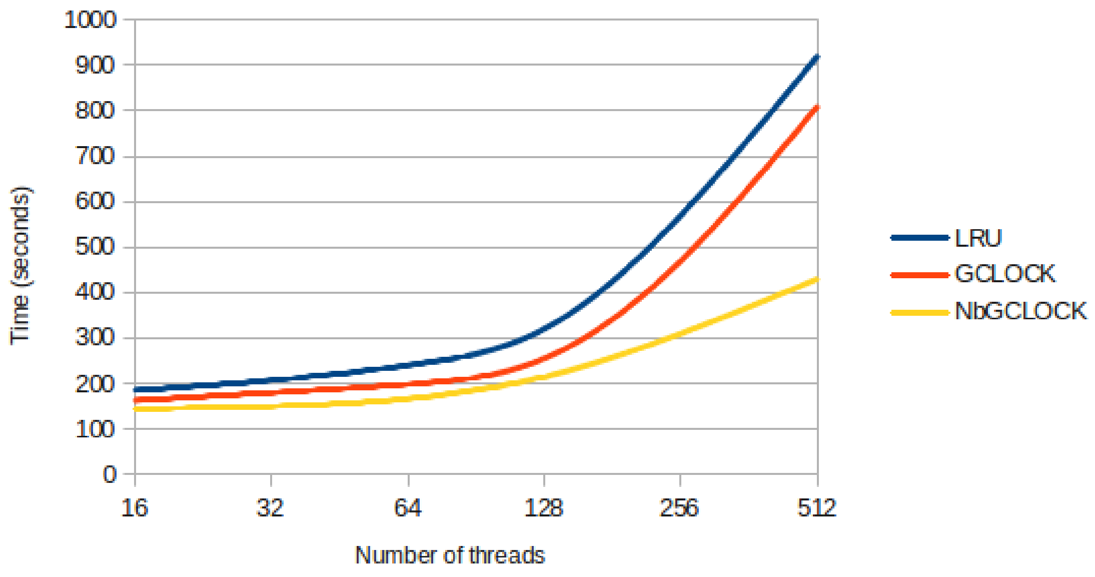

Figure 5.3: Comparison of execution time when number of threads increases.

justified because when we increase the concurrency level, it causes problems like priority inversion and convoying tend to happen more often and this impact directly on performance.

Figure 5.4 illustrates the comparison between buffer managers implementations with different lookup structures: hashmap using mutual exclusions, Intel Thread Building Blocks concurrent_ hash_ map and HashSet implemented by libcds. Since the goal is to compare the three data structures alone, all of the three implementations use NbGCLOCK as replacement policy. For the experiment the buffer size is kept constant at 32 Megabytes and the number of threads is increased. For each approach the experiment was executed 30 times and the average is compared.

Again, all of three implementations present the same behaviour in scenarios with low to moderate concurrency. It is important to note that in this case, the lock-free structure is not a dominant factor in terms of the scalability and other alternatives, even if they use blocking synchronization, can be used instead. This is justified because access to hash maps are unlikely to happen in the same portion of the structure implying a little need for synchronization. When the concurrency level is increased from a certain point, the difference of a lock-free implementation is notable because concurrent access to the same portion of the structure need to be more frequent. However, even if a lock-free hashmap is not dominant in terms of scalability, it is important to ensure the lock-freedom property of the whole buffer.

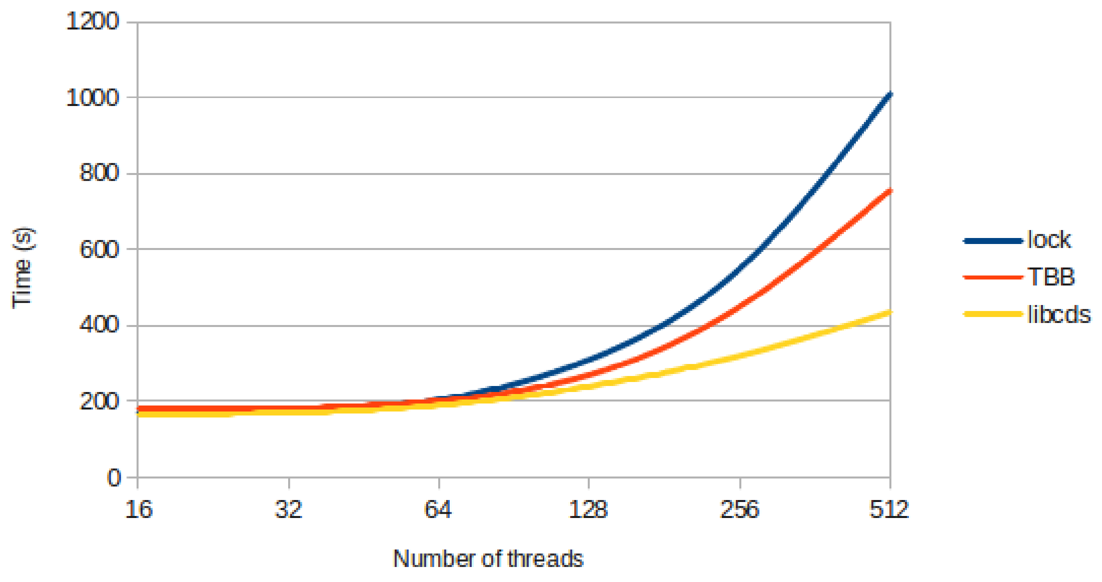

Figure 5.4: Comparison of execution time when number of threads increases.

6 CONCLUSION

This work presented the design and implementation of a lock-free database buffer in the context of the WattDB project based on non-blocking synchronization. Initially, all the important concepts required for the full understanding of this work were introduced. After, important aspects that should be considered when designing a lock-free buffer were presented and the implementation was discussed. Finally experiments and results were shown.

Lock-free programming is a very promising concept in the context of highly concurrent environments. Lock-free algorithms offer good properties and advantages over classical blocking programming, being freed from synchronization problems like deadlocks, priority inversion and convoying. However, these algorithms tend to be much more complex to implement and often impose environment restrictions. The design and understanding of algorithms using non-blocking synchronization require more time than those of mutual exclusion and this complexity is the main obstacle for programmers to accept non-blocking synchronization.

Nevertheless, new lock-free algorithms like concurrent data structures with non-blocking synchronization are being frequently proposed. These data structures can provide to programmers basic building blocks for their applications. Also, memory management schemes for non-blocking synchronization exist and are highly desirable to help the designing of these algorithms and overcome initial obstacles. These technologies are crucial and can lower the entry cost of using non-blocking synchronization.

In this work we presented and discussed the aspects and designing of a lock-free database buffer. We implemented a buffer based on a lock-free hash map structure and a non-blocking replacement algorithm for the buffer's frame pool. Since database buffer operations are the most frequent operations called in a database, the goal was to improve the performance of such operations in highly concurrent environments and CPU-bound scenarios.

REFERENCES

[Codd 1983]CODD, E. F. A relational model of data for large shared data banks. *Commun. ACM*, ACM, New York, NY, USA, v. 26, n. 1, p. 64–69, jan. 1983. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/357980.358007>.

[Detlefs et al. 2001]DETLEFS, D. L. et al. Lock-free reference counting. In: *in Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 2001. p. 190–199.

[Doherty et al. 2004]DOHERTY, S. et al. Dcas is not a silver bullet for nonblocking algorithm design. In: *In SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. [S.l.: s.n.], 2004. p. 216–224.

[Effelsberg e Haerder 1984]EFFELSBERG, W.; HAERDER, T. Principles of database buffer management. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 9, n. 4, p. 560–595, dez. 1984. ISSN 0362-5915. Available from Internet: <http://doi.acm.org/10.1145/1994.2022>.

[GNU Project 2013]GNU Project. *GNU Compiler Collection*. 2013. [Online; accessed May-2013]. Available from Internet: <http://gcc.gnu.org/>.

[Gray e Reuter 1992]GRAY, J.; REUTER, A. *Transaction Processing: Concepts and Techniques*. 1st. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN 1558601902.

[Haerder 2005]HAERDER, T. *DBMS Architecture – the Layer Model and its Evolution*. 2005.

[Haerder e Reuter 1983]HAERDER, T.; REUTER, A. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 15, n. 4, p. 287–317, dez. 1983. ISSN 0360-0300. Available from Internet: <http://doi.acm.org/10.1145/289.291>.

[Haerder e Schall 2012]HAERDER, T.; SCHALL, D. *WattDB – A Cluster of Wimpy Processing Nodes to Approximate Energy Proportionality*. 2012.

[Intel 2013]INTEL, I. C. *Intel Threading Building Blocks*. 2013. [Online; accessed May-2013]. Available from Internet: <http://threadingbuildingblocks.org/>.

[Khizsinsky 2013]KHIZSINSKY, M. *libcds, Concurrent Data Structures*. 2013. [Online; accessed May-2013]. Available from Internet: <http://libcds.sourceforge.net/>.

[Michael 2002]MICHAEL, M. M. High performance dynamic lock-free hash tables and list-based sets. In: *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2002. (SPAA '02), p. 73–82. ISBN 1-58113-529-7. Available from Internet: <http://doi.acm.org/10.1145/564870.564881>.

[Michael 2004]MICHAEL, M. M. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, IEEE Press, Piscataway, NJ, USA, v. 15, n. 6, p. 491–504, jun. 2004. ISSN 1045-9219. Available from Internet: <http://dx.doi.org/10.1109/TPDS.2004.8>.

[1]O'NEIL, E. J.; O'NEIL, P. E.; WEIKUM, G. The lru-k page replacement algorithm for database disk buffering. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 22, n. 2, p. 297–306, jun. 1993. ISSN 0163-5808. Available from Internet: <http://doi.acm.org/10.1145/170036.170081>.

[Smith 1978]SMITH, A. J. Sequentiality and prefetching in database systems. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 3, n. 3, p. 223–247, set. 1978. ISSN 0362-5915. Available from Internet: <http://doi.acm.org/10.1145/320263.320276>.

[Tanenbaum 2007]TANENBAUM, A. S. *Modern Operating Systems*. 3rd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007. ISBN 9780136006633.

[TPCC 2010]TPCC, T. P. P. C. *TPC Benchmark C, Standard Specification, Revision 5.11*. fev. 2010. Available from Internet: <http://www.tpc.org/tpcc/default.asp>.

[Valois 1995]VALOIS, J. D. Lock-free linked lists using compare-and-swap. In: *In Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*. [S.l.: s.n.], 1995. p. 214–222.

[Yui et al. 2010]YUI, M. et al. Nb-gclock: A non-blocking buffer management based on the generalized clock. *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, IEEE Computer Society, Los Alamitos, CA, USA, v. 0, p. 745–756, 2010.