Optimistic locking and epoch-based reclamation for efficient implementations

Maksim Pavlov

June 2024

Contents

Preface

This report delves into various locking techniques, exploring their applications, benefits, and drawbacks. It also provides a comparative analysis of these techniques, showcasing my insights and the knowledge I gained during the practical course "Introduction to C++" (IN0012).

Chapter 1

Introduction

1.1 Locks and latches

Locks are synchronization mechanisms used to control access to shared resources, ensuring that only one thread or process can access the resource at a time. They prevent concurrent threads or processes from interfering with each other's operations and maintain data integrity in multi-threaded or multi-process environments. Types of locks are:

- **Mutex (Mutual Exclusion)**: A mutex provides exclusive access to a resource. Threads attempting to acquire a mutex lock will block (or spin) until the lock becomes available. An example for mutex in C++ is std::mutex.
- **Semaphore**: A generalized version of a mutex that can allow multiple threads to access a resource concurrently, up to a specific limit controlled by the semaphore count. Unlike the mutex, the semaphore doesn't have an owner, meaning that the semaphore can be unlock by any thread, not only by the one who locked it. An example for semaphore in C++ is std::counting_semaphore.
- **Read-Write Locks**: Read-write locks are synchronization primitives that allow concurrent access for read operations while maintaining exclusive access for write operations. This means multiple threads can read from a shared resource simultaneously, but write access is granted to only one thread at a time, blocking other reads and writes until the write operation is complete. An example implementation in C++ is std::shared_mutex.

A way to classify locks is by what happens when the lock strategy prevents the progress of a thread. One way to go is to block the execution of the thread (sleeping) requesting the lock until it is allowed to access the resource. Another way is to use spin lock, which constantly tries to acquire the lock normally using

a loop, potentially consuming too many resources (e.g. CPU cycles) if it waits too long.

The locks may also rely to some extend on atomic operations like "test-and-set" or "fetch-and-add".

In database context, the term latch is defined similarly to the lock, but the latch don't hold until the entire logical transaction. It holds only on the operation and protects the critical sections.

1.1.1 Pessimistic locking

The pessimistic locking has two states:

- **Exclusive lock**: At most one thread or process can acquire the lock in this state. It's useful when one wants to modify a resource and ensures that no data races happen. While thread or process is in this state, no one (except the one holding the lock) can access the resource. A downside of the this state is that users can be locked out for a long time, thereby slowing the overall system progress.
- **Shared lock**: On the other hand, shared lock allows multiple threads or process to read from resource without having to exclude each other from accessing it. While a thread or process is in this state, no one can access the resource exclusively.

Implementation in C++ that has these characteristics is std::shared_mutex. This technique is mainly used in environments where data-contention is heavy, the cost of protecting the data with locks is less than the cost of rolling back transactions, if a conflict occur, or the next update can be delayed until the previous update is complete.

1.1.2 Optimistic locking

Optimistic locking is a mechanism that allows multiple threads or processes to complete without explicit locking of resources during their execution phase. We can divide the work of this mechanism in three phases:

- **Read phase**: The transaction reads the necessary data and performs its computations. It also records the version of the data or a timestamp to check for modifications later.
- **Validation phase**: Before committing the transaction, the system checks if the data has been modified by another transaction since it was read. This check typically involves comparing the current version with the recorded version or timestamp.
- **End phase**: If no modifications are detected, the transaction proceeds with the information it read from the resource. If a conflict is detected, the transaction is rolled back, and the changes are discarded. The transaction may then retry from beginning.

This technique is appropriate in environments where there is low contention for data, typically when we have few updates of the resource and mostly reading from it.

	Optimistic	**Shared**	**Exclusive**
Optimistic	✓	✓	✗
Shared	✓	✓	✗
Exclusive	✗	✗	✗

Table 1.1: Locking mode compatibility

1.2 Lock-free approach

The lock-free approaches don't rely on traditional locking mechanisms such as mutexes or semaphores. Instead, they are designed to operate with atomic operations and synchronization primitives provided by hardware and programming languages to ensure thread safety and progress in concurrent environments.

Examples of atomic operations in C++ are std::atomic<T>::fetch_add or std::atomic<T>::fetch_or, which perform atomic addition and atomic bitwise OR operation respectively between the value in the arguments and the atomic variable and saves the result in the atomic variable.

Another example is std::atomic<T>::compare_exchange_strong, which is a CAS (compare-and-swap) operation meaning it will compare the atomic variable with expected value and if they match, the value in the atomic variable will be replace with desired one.

Further, we can also use memory ordering, which specifies how memory accesses, including non-atomic memory accesses, are to be ordered around an atomic operation. In C++, std::memory_order provides such functionalities.

Two examples are memory_order_acquire and memory_order_release. The first one is usually used with load operation. It guarantees that no ordinary operations in the current thread are reordered before the load operation that uses memory order and that the current thread see all writes of other threads that release the same atomic variable.

The second one is used usually with store operations. It guarantees that no ordinary operation are reordered after the store operation that uses memory order and all writes of the current thread are seen by the others that acquire the same atomic variable.

1.3 Epoch-based reclamation

If we use a lock-free data structure, we can delete a resource that reader/s might be still accessing. Freeing or reusing that resource immediately may cause correctness problems and/or crashes e.g SEGFAULT. For this reason, we need some mechanism to ensure that one resource can be safely deleted. One

of the solutions (on which we will concentrate in this report) is the epoch-based memory reclamation. The main logic is to have a global epoch, which is incremented periodically, and local epoch counters for each thread. Before a thread accesses the data structure it sets its local epoch to the current state of the global epoch. When a thread exits the data structure, it sets its local epoch to a special value to show it doesn't access any resources. When a thread wants to defer a resource, it marks it with the global epoch and place it in so called 'limbo bag' (list or a vector, depending on the implementation). When it comes time to clean the limbo bag (e.g. if the limbo bag is implemented as FIFO queue, this moment may be when the resource reaches the end of the queue), we take the minimum of all thread local epochs. If the resource was marked with epoch less than the minimum of all local epochs, it can be safely deleted. Otherwise, it should remain in the limbo bag.

The idea is relatively easy to understand and implement. It provides fairly low overhead, because we use simple operations and there is no need to change the data structure code, but simply wrap the operations. On the other hand, incrementing epochs too frequently incurs overhead, but if they are not updated often enough, the limbo bag may grow large. One further drawback may be that a single slow thread may hold an epoch for very long time and prevent any memory reclamation.

1.4 Implementation

For the discussion and evaluation of pessimistic and optimistic locking, the HybridLock, HybridGuard, EpochHandler and EpochGuard classes were developed.

- **HybridLock**: The HybridLock class implements the necessary logic behind the locking mechanisms. It provides the functions to control the version and the state of the lock, try_to_lock and unlock functions and upgrade and downgrade functions to change the state of the lock from exclusive to shared and vice versa, using atomic operations.
- **HybridGuard**: The HybridGuard class implements the API that controls the HybridLock. It's hybrid, because it both supports Pessimistic and Optimistic locking. Setting the mode_ attribute, we can easily control what lock we want to acquire.
- **EpochHandler**: The EpochHandler class controls the global epoch as well as the outdated resources in the 'limbo bag' mentioned previously (the place where we store the resources that should be deleted in the future).
- **EpochGuard**: The EpochGuard class manages the transitions of the single threads between the different local epochs. When it accesses the data structure, it sets its local epoch to the current global epoch and when it leaves it set it to infinity (UINT64's max value).

Additionally, two sorted singly-linked lists were developed to compare the two locking practices.

- **MutexSortedList**: MutexSortedList is a classical implementation of a multi-threaded sorted list with pessimistic lock. Insertion in the list would mean holding the lock in exclusive mode and finding the right place to insert the element. Searching for an element means locking in shared mode, so elements can't be modified, and finding out if the element is in the list. Deletion is holding the lock in exclusive mode and removing the element.

- **OptimisticSortedList**: OptimisticSortedList uses the implemented methods and classes above to lock optimistically and implements also a multi-threaded data structure. Insertion and deletion acts the same as MutexSortedList; it just uses our implementation of exclusive lock. Deletion puts the deleted elements in retired list, so they can be deleted later, and uses exclusive mode of the lock. Searching involves holding the lock in optimistic mode. Creating the EpochGuard class we set the local epoch to the value of the global epoch. Then, we try to find the element. If it a conflict is detected, we restart the process. It repeats until we managed to complete the searching without being interrupted by write access. When it succeeds, we set the thread's local epoch to infinity.

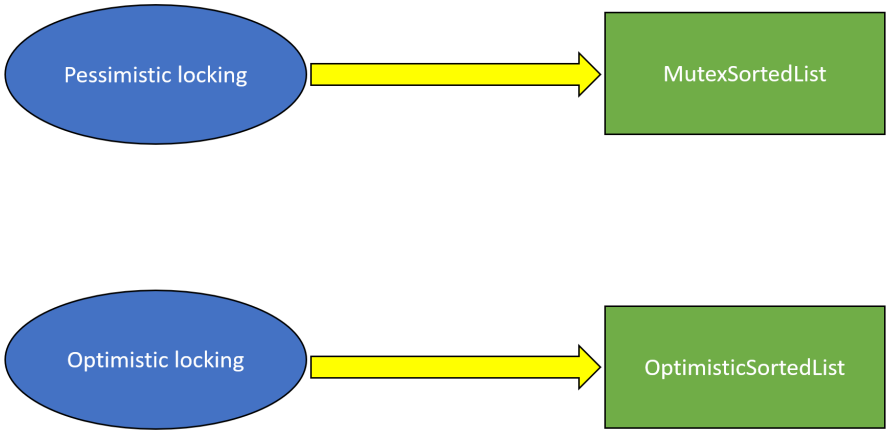

Figure 1.1: Lock to data structure mapping

Further, an example with transactions and multiple users was developed using PostgreSQL and C++ to show pessimistic and optimistic locking in another light.

Chapter 2

Benchmarks

To compare pessimistic and optimistic locking simple benchmarks were made. The benchmarks were made using machine with following parameters:

- **OS**: Windows
- **CPU**: Intel(R) Core(TM) i7-865U CPU @ 1.90GHz 2.11 GHz
- **Number of cores**: 4
- **RAM**: 32 GB
- **L1d cache size**: 128 KiB (32 KiB per core)
- **L1i cache size**: 128 KiB (32 KiB per core)
- **L2 cache size**: 1 MiB (256 KiB per core)
- **L3 (LLC) cache size**: 8 MiB
- **Compiler**: g++ 11.4.0

Each test was performed 5 times and the mean value of the results was taken into account to ensure accurate end results. To acquire the data in the diagrams, perfevent was used, which provides access to a wealth of detailed performance information related to CPU's functional units, caches, main memory etc.

For this benchmark, one shouldn't concentrate that much on the exact numbers, but on the proportions, because the program doesn't use all available resources for itself.

2.1 Benchmark 1 : Basic test without data structure

This test is made using no data structures, but pure pessimistic and optimistic locks updating and reading a counter. The shared lock is compared with optimistic lock. In the diagrams about L1 and LLC cache misses, we see that in the

most cases the optimistic locking has true advantage over pessimistic locking. In the diagram involving the CPU cycles, the optimistic approach performs also better than the pessimistic one.

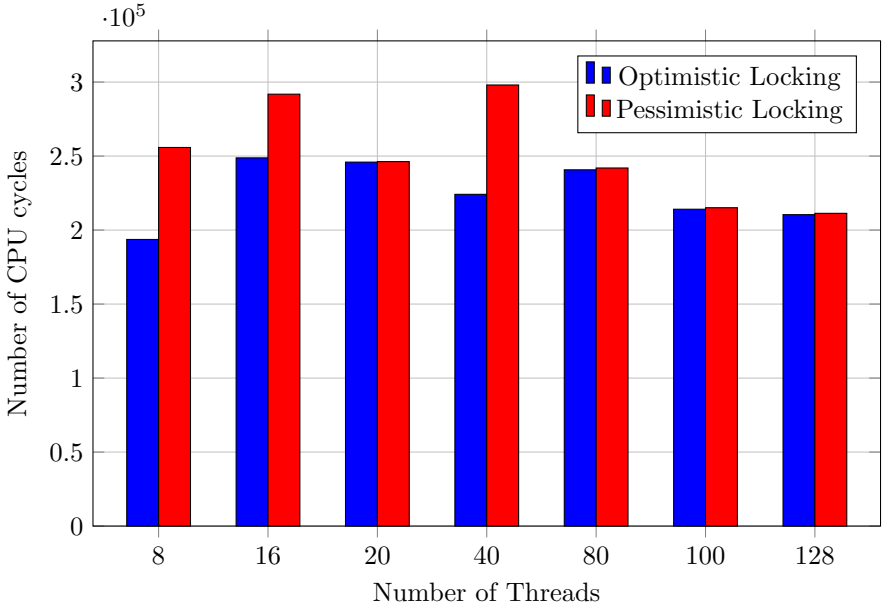

Figure 2.1: Comparison of CPU cycles (Benchmark 1)

2.2 Benchmark 2 : Basic Node

This test is made using the following basic node structure. First, we add elements in the list with a single thread, then we look up (calling LookUp method of each of the two classes) for the nodes using multiple threads. We observe the same result as in the previous benchmark. Optimistic locking is better in CPU cycles test and truly better in L1 and LLC cache misses tests than pessimistic locking.

```
struct Node {
  int key;
  int value;
  Node *next;

  Node(int key, int value, Node *next)
  : key(key), value(value), next(next) {}

  ~Node() = default;
};
```

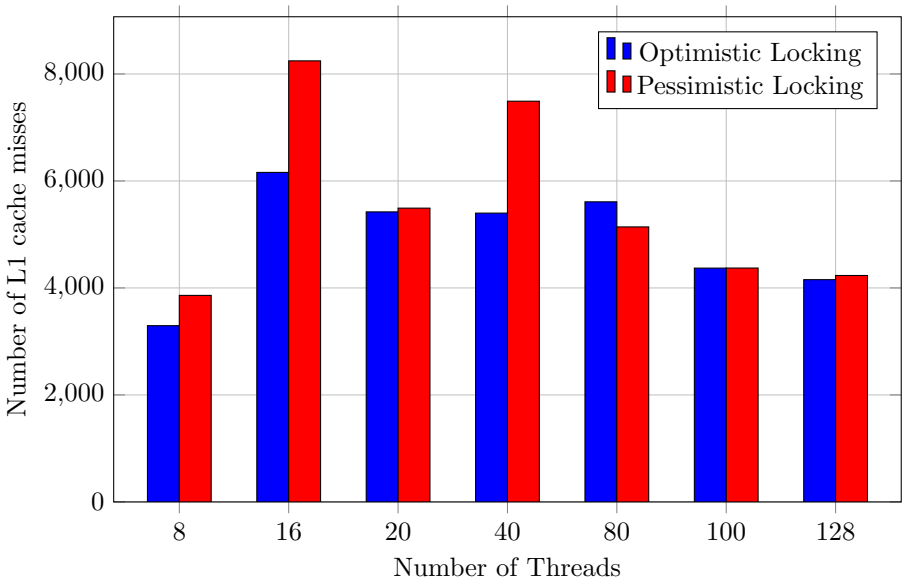

Figure 2.2: Comparison of L1 cache misses (Benchmark 1)

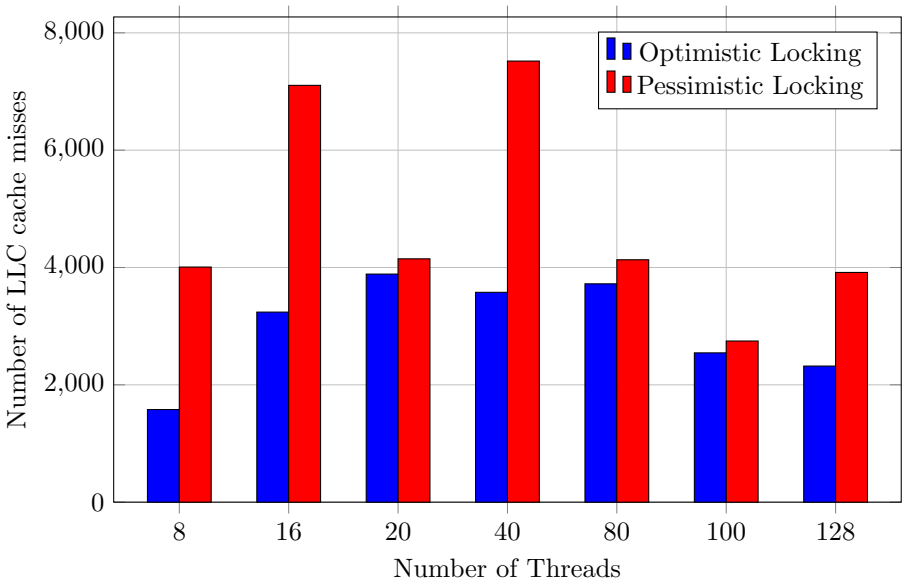

Figure 2.3: Comparison of LLC cache misses (Benchmark 1)

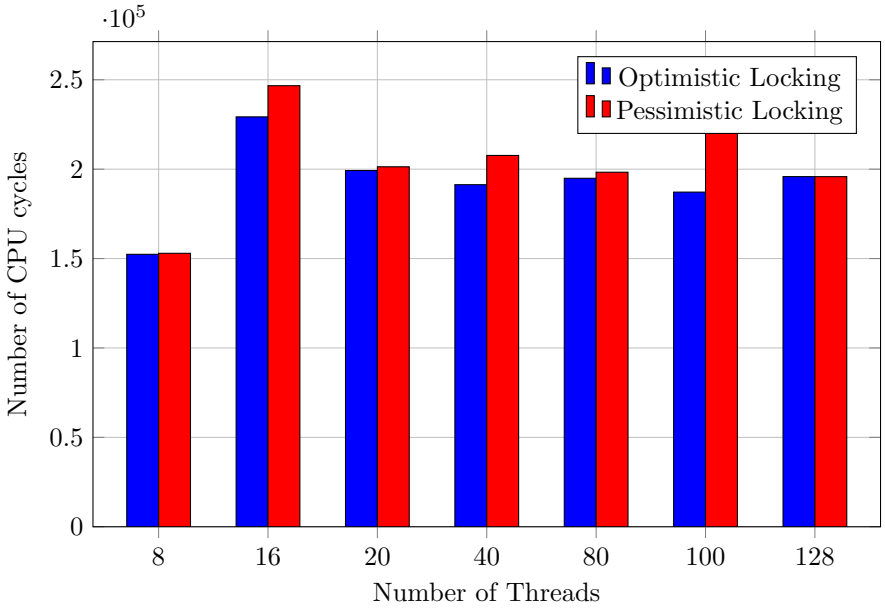

Figure 2.4: Comparison of CPU cycles (Benchmark 2)

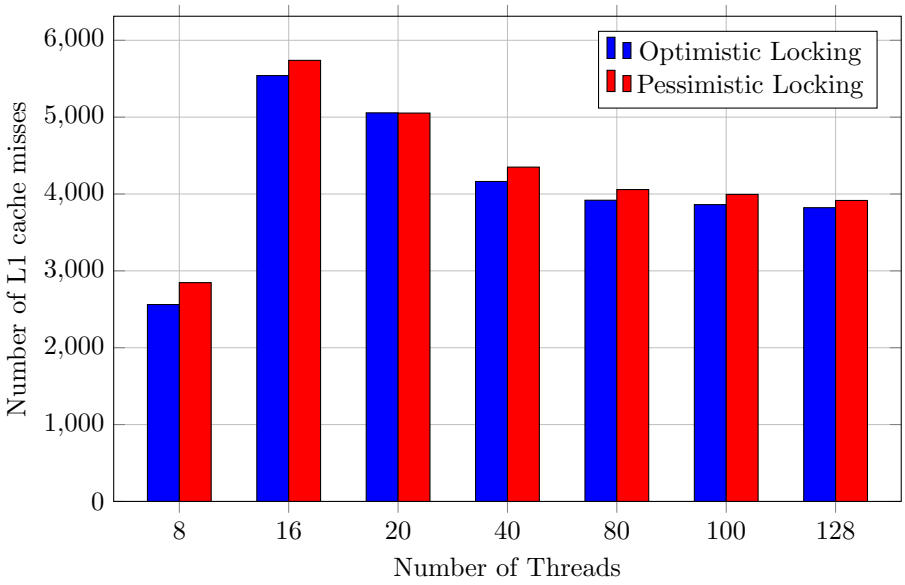

Figure 2.5: Comparison of L1-Cache Misses (Benchmark 2)

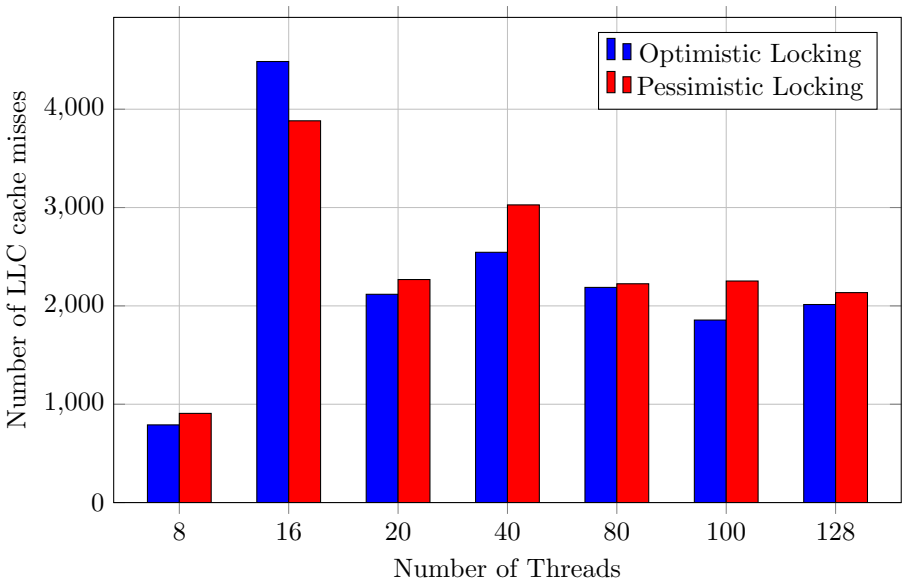

Figure 2.6: Comparison of LLC cache misses (Benchmark 2)

2.3 Benchmark 3 : Student Node with 32 byte alignment

The test is made using the following structure as a node and compares the std::shared_lock with the optimistic lock in the way they make the searching. It shows a more real-world example - entries of a example database (although for databases, B-Trees are preferred). We align each entry on 32 byte boundary, so we don't have any elements split on two cache lines. Again the optimistic technique proves to be better than the pessimistic one.

```
struct alignas(32) Student {
private:
    unsigned matrID;
    std::string name;
    uint8_t currentSemester;

public:
Student(int matrID, const std::string& name,
                                        uint8_t currentSemester)
   : matrID(matrID), name(name), currentSemester(currentSemester)
   {}

    auto operator<=>(const Student& other) const {
        return matrID <=> other.matrID;
    }
};
```

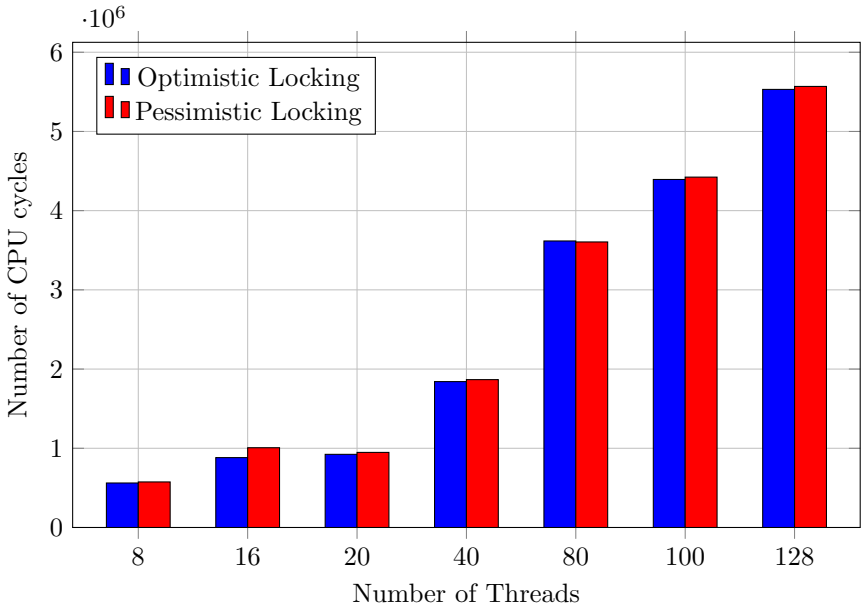

Figure 2.7: Comparison of CPU cycles (Benchmark 3)

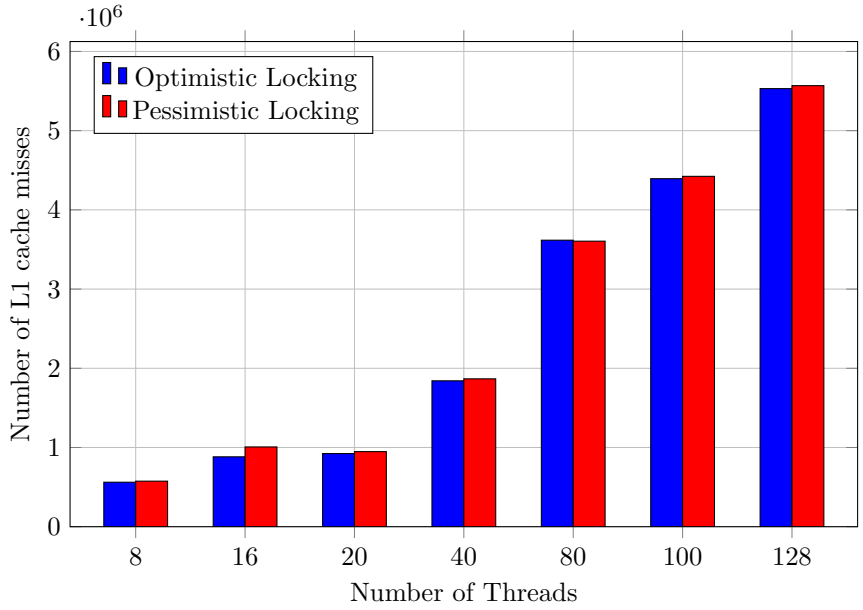

Figure 2.8: Comparison of L1 cache misses (Benchmark 3)

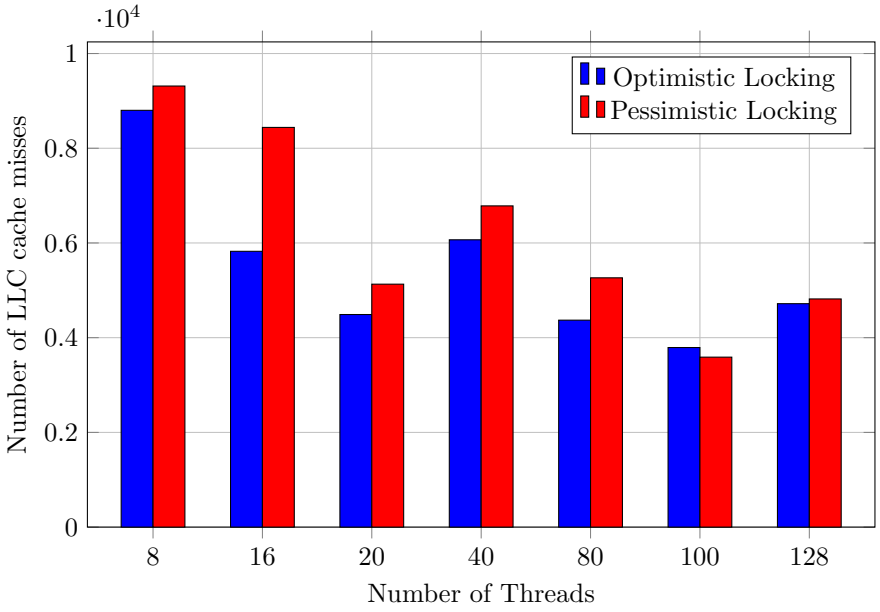

Figure 2.9: Comparison of LLC cache misses (Benchmark 3)

Chapter 3

Pessimistic locking versus Optimistic locking

In the introduction, the two locking techniques were presented. In this chapter, they will be compared.

Complexity

Pessimistic locking, dating back to the 1960s-1970s, is a straightforward approach where exclusive locks are acquired for writing and shared locks for reading. This way it provides very clear coordination to the threads.

In contrast, optimistic locking poses greater challenges and complexity. It assumes operations can proceed without holding locks initially, checking for conflicts only during validation. However, this approach can fail if some thread attempts to write the resource, while we are reading it, necessitating retry mechanisms. Optimistic locking is suitable in scenarios where occasional failures can be tolerated, allowing operations to be retried until successful.

Managing deleted nodes in memory is critical to prevent crashes when using optimistic locking. Techniques like epoch-based reclamation or hazard pointers ensure memory remains valid even after deletion of the element from the data structure, so it can be still accessed by the threads who already read it.

Moreover, all operations under optimistic locking must be restartable without adverse effects, applying changes only after successful validation. For instance, data read from a buffer should only be used if we read it without encountering conflict.

Our implementation of pessimistic locking and optimistic locking also proves this. Exclusive and shared locking methods are a lot simpler, but if we use optimistic locking, we have to think about how should we handle the RestartException, when to advance the global epoch etc.

Scalability and Concurrency

In systems with multi-core processors, the issue with traditional locking mechanisms arises from the need to acquire and release locks by writing to shared variables, such as std::atomic<uint64_t> state_and_version_. Due to the cache coherence protocols, like MESIF used by Intel processors, which manage the consistency of cached data across cores, when a core modifies a shared variable, it gains exclusive ownership of the cache line containing that variable. This action invalidates cached copies of the variable on all other cores. The true sharing of the cache line leads to what's known as the "cache line ping pong effect". This frequent invalidation forces other cores to fetch the updated variable from main memory. Consequently, this results in higher cache misses and increased latency, impacting overall system performance.

In contrast, optimistic locking avoids this problem because it operates without modifying shared variables during normal operations. This leads to less L1 and LCC cache misses, proven by the benchmark in Chapter 2. Instead of holding locks, it allows multiple threads to proceed with their operations concurrently, assuming no conflicts will occur, allowing the programs to complete their jobs (usually) in less cycles according to the benchmark. Important to mention is that if we use optimistic locking to update a resource concurrently the program would use more cycles to complete its job because of the higher contention and more retries needed to complete.

Deadlock

One of the primary concerns for programmers dealing with concurrent programming is the occurrence of deadlocks, especially prevalent when using pessimistic locking strategies. Deadlocks can be challenging to debug and mitigate once they arise. In pessimistic locking, threads acquire locks on resources before performing operations, which can lead to situations where one thread holds a lock needed by another thread, causing both threads to wait indefinitely for each other to release their respective locks.

However, with optimistic locking, deadlocks are not a concern because this approach avoids locking resources preemptively. Instead of locking resources before performing operations, optimistic locking allows threads to proceed with their operations concurrently, assuming no conflicts will occur. Only during a validation phase, typically at the end of a transaction or critical section, does optimistic locking check for conflicts. If conflicts are detected, the transaction can be retried or rolled back.

Starvation

In environments with high write contention, optimistic locking can indeed face issues such as starvation, where repeated conflicts delay transactions from completing successfully. To mitigate this, it may be necessary to incorporate a fallback mechanism to pessimistic locking after a certain number of retries (see

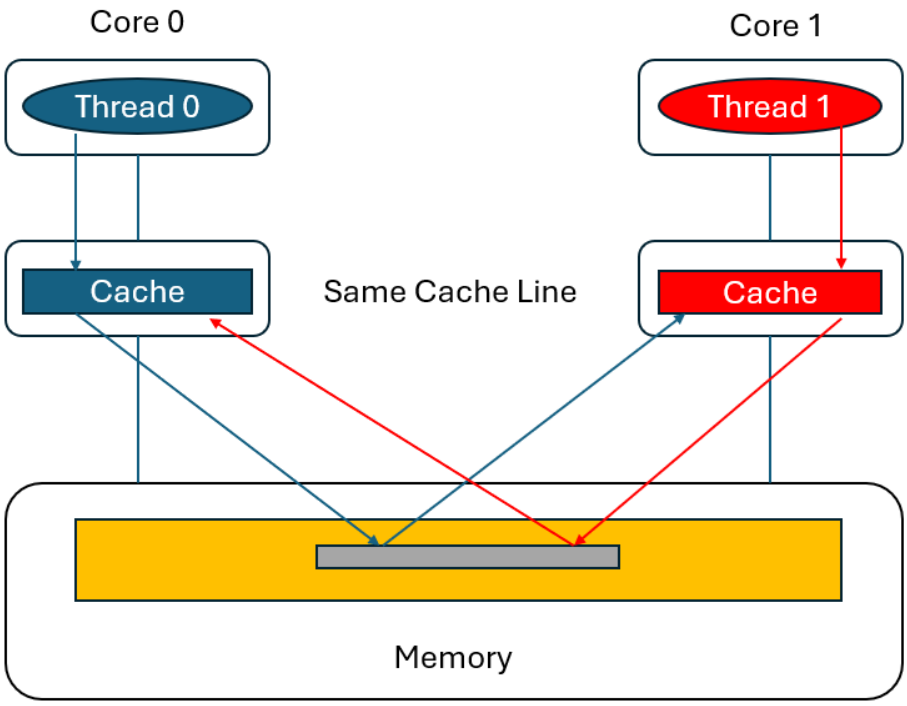

Figure 3.1: CPU Cache Line Ping Pong

example implementation using our functions down). This ensures that the resource can eventually be accessed, even if conflicts persist initially. This approach can be a good idea if one doesn't know exactly if high or low contention is expected, that's why one decides to use a mixture of pessimistic and optimistic locking.

```
void OptimisticLock(some args)
{
    int max_tries = 10;
    int try = 0
    while(try < max_tries)
    {
        if(lock_->LockState() == HybridLock::EXCLUSIVE){
          try++;
          continue;
        }

        state_ = lock_->StateAndVersion().load();

        // Here we can do the job we want;
        // It can be read, but also write

        try{
```

```
            ValidateOptimisticLock();
            return; // or commit the changes or return value
        } catch (const RestartException &) {
            try++;
            // Potentially undo some changes
        }
    }

    // Fallback to exclusive or shared lock
    // In this case let's say we want to lock exclusively
    while(!lock_->TryLockExclusive(lock_->StateAndVersion().load())
        )
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(10));
    }
    state_ = lock_->StateAndVersion().load();

    // Do the job we want

    Unlock();

    return; // or commit the changes or return value
}
```

Conversely, pessimistic locking involves acquiring locks, which can lead to increased waiting times and potential for starvation if locks are held for extended periods. Here starvation is from another kind. It wouldn't be for the current thread that isn't able to acquire the lock, but for the other threads that are waiting.

Inaccessible data

With optimistic locking, data can't be left inaccessible to other users as a result of a user taking a break or being excessively slow in responding to prompts. Locking systems leave locks set in these circumstances denying other users access to the data. Further, data cannot be left inaccessible as a result of client processes failing or losing their connections to the server. Many of the modern databases handle this problem by detecting the disconnection and aborting the transaction. However, this problem is handled by default with optimistic locking.

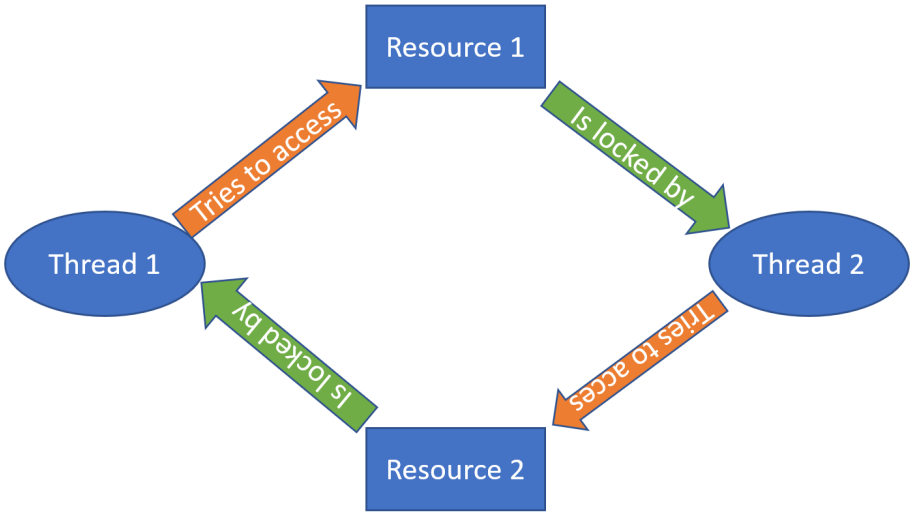

Figure 3.2: Deadlock

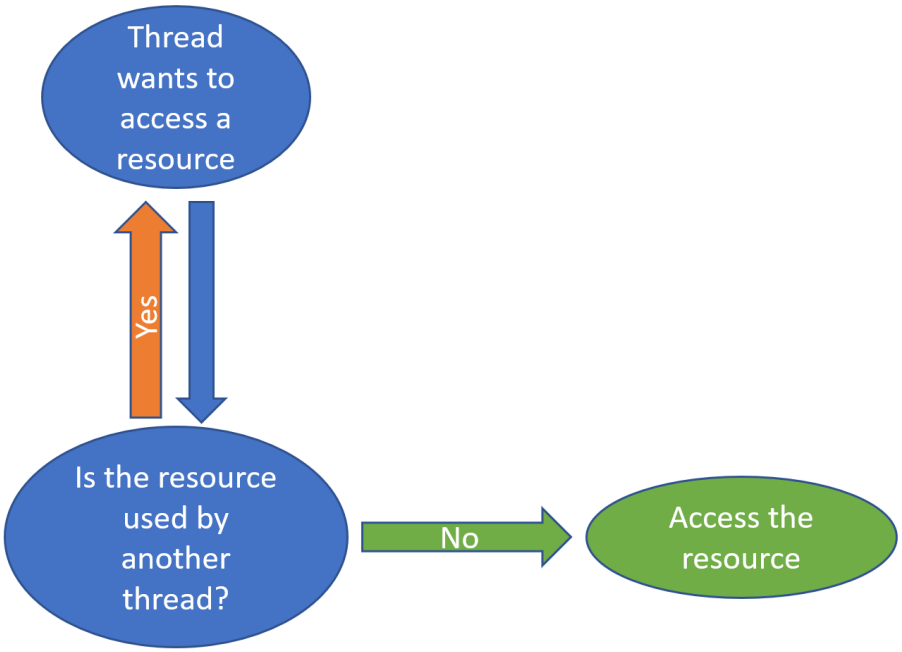

Figure 3.3: When we answer to the question "Is the resource used by another thread?" too many times with yes, the thread starves

Chapter 4

Deallocation of outdated resources

In epoch-based reclamation, the safe deletion of the resources is crucial to ensure that no thread accesses a freed memory region, which would result in undefined behaviour and potential program crashes. The general principle is to delay the deletion of resources until it is guaranteed that no thread can access the memory being reclaimed. This involves balancing between minimizing the overhead of reclamation operations and ensuring timely freeing of memory to prevent excessive memory usage. This can highly depend on the individual system specifics like the size of the main memory and available computational resources. Here are three proposes when the outdated resources to be deallocated (in our case, EpochHandler::FreeOutDatedPtr() to be called):

Batch processing

Here we have two cases: We may check periodically and conditionally. When we check periodically, we perform reclamation at regular intervals. This can be useful when we know that after some interval of time the retired list (also called limbo bag in this report) may contain outdated resources. We may also interpret the periodical deallocation of resource as deallocation during specific windows. We can perform cleanup during scheduled maintenance windows or periods of low activity to minimize the impact on users and ensure that the outdated memory is deallocated properly. On the other hand, we can check also conditionally. Based on specific conditions such as the number of retired objects exceeding a certain threshold or memory usage reaching a certain level. This one would be very successful on system with little main memory or in environments that produce a lot of outdated resources in short period of time. This technique could be also used after large transactions, such bulk inserts, updates, or deletes, which can temporarily consume a significant amount of memory. Another condition that we could use is the change of the global epoch.

For example, if we have some expected value for deferred resources per epoch, we could use it to say that every two (or every three and so on) increments of the global epoch would mean deallocation of outdated memory.

Piggybacking with other operations

Scheduling the deallocation of the outdated resources with other activities would be also useful and would take advantage of natural synchronization points to minimize additional overhead. Using this technique, we can check on each user that leaves the data structure if there is any memory to be deallocated. Example for such operation in operating systems is the software-based timer, which gets active only when the system is in Kernel mode (the system gets in Kernel mode at least once every 10 us).

Hybrid approach

A hybrid approach would mean a combination of piggybacking and batching. Example for such implementation would be every tenth thread that leaves the data structure to free the outdated pointers and when this is too slow, start also periodically deallocating outdated resources.

The deallocation of the outdated resources is also connected with incrementing the global epoch. If we advance the global epoch variable too frequently, this may make the copies of the cache line in all cores invalid and produce cache coherency traffic, whereas infrequent increments may prevent outdated resources being deallocated or reused in a reasonable time frame. Therefore the increments of the global epoch should consider the number of deferred resources (because the limbo bag may grow large) and a constant value as a lowerbound, which depends on the hardware, the application etc. One can also use approaches like batch processing or piggybacking as described above to increment the global epoch.

Chapter 5

Real world implementations

Multiversion concurrency control

Multiversion Concurrency Control (MVCC) is a non-locking concurrency control method used by Database Management Systems (DBMS) to provide concurrent access to the database. Many major products, such as Microsoft SQL Server, Oracle, PostgreSQL, and SAP HANA, implement MVCC under different names, but the underlying concept remains the same.

The main idea behind MVCC is to create multiple versions of a piece of data when it is modified. Each transaction views a snapshot of the database at a specific point in time, determined by the timestamp or version number associated with each data version. This allows transactions to read consistent snapshots of the database, even while other transactions are modifying data. As a result, queries within a transaction reflect a consistent state of the database from the start of the transaction.

In MVCC, writes create a newer version of the data, while concurrent reads access an older version. This approach effectively avoids the need for pessimistic locking, offering a non-locking solution to concurrency control.

5.1 Case Study : PostgreSQL and C++

PostgreSQL is a well-known free and open-source relational database management system, developed by PostgreSQL Global Development Group. In this example I use their database (through libpqxx) and showed another implementation of optimistic and pessimistic locking. Here is the implementation of the optimistic lock:

```
// Optimistic Locking Implementation
class AccountRepositoryOpt {
public:
    AccountRepositoryOpt(pqxx::connection& conn) : conn(conn) {}
```

```
    // Make deposit.
    // The struct DepositRequest contains
    // std::string 'username' and double 'amount' members
    // The struct Account has an attribute
    // for each column of the table
    void depositOpt(const DepositRequest& request) {
       while(true){
        pqxx::work txn(conn);
        try {
            // Select account
            pqxx::result res =
            txn.exec_prepared("select_account_by_username"
                                              , request.username);
            if (res.empty()) {
                throw std::runtime_error("Account not found");
            }

            Account account;
            for (const auto& row : res) {
              account.id = row["id"].as<int>();
              account.username = row["username"].as<std::string>();
              account.balance = row["balance"].as<double>();
              account.version = row["version"].as<int>();
            }

            // Modify balance
            account.balance += request.amount;

            // Update account with optimistic lock
            pqxx::result update_res =
            txn.exec_prepared("deposit_account_opt"
                    , account.balance, account.id, account.version);
            // If no rows are affected from the change,
            // abort the transaction and try again
            if (update_res.affected_rows() == 0) {
                txn.abort();
            } else {
                txn.commit();
                return;
            }
        } catch (const std::exception& e) {
            txn.abort();
            std::cerr << "Error: " << e.what() << std::endl;
        }
    }
    }
private:
    // Connection to the database
    pqxx::connection& conn;
};
```

We assume that the table looks like:

```
CREATE TABLE account (
    id SERIAL PRIMARY KEY,
    username VARCHAR(100) UNIQUE NOT NULL,
    balance NUMERIC(10, 2) DEFAULT 0.0,
    version INT DEFAULT 0
```

```
)
```

and that the prepared statements are:

- select_account_by_username = SELECT * FROM account WHERE username = $1 LIMIT 1;
- deposit_account_opt = UPDATE account SET balance = $1, version = version+1 WHERE id = $2 AND version = $3;

The characters of the type '$(number)' are replaced with values specific for the situation. This example can be implemented with most of the big databases like Oracle, MySQL, Microsoft SQL Server. Here we could also use our implementations of pessimistic and optimistic locks, but the aim here was to use another feature implemented by the provider. It shows optimistic locking not as a database feature, but as a practice that is applied using the database with standard instructions.

5.2 Case Study : Redis

Redis is an open source in-memory datastore that can be used as a database, cache, or message broker. It has some features that may make it attractive for use as a database, such as support for transactions and publish/subscribe messaging. Redis introduces the keyword WATCH, which provides CAS behavior to the transactions. The 'watched' keys are tracked in order to detect changes. If one of the keys is modified before the EXEC command (for commiting), the whole transaction aborts and returns a Null reply to notify the failure. Simple example:

```
WATCH mykey
val = GET mykey
INCR val
MULTI
SET mykey $val
EXEC
```

Here we start WATCHing the key 'mykey', get it from the datastore, increment it and if no conflict, update it in the datastore. The WATCH keyword is idempotent and Redis offers the opposite keyword UNWATCH to stop the tracking of all keys.

5.3 Case Study : Jakarta Persistence

Jakarta Persistence, known as JPA, is a Jakarta EE application programming interface specification that describes the management of relational data in Java applications. It introduces the 'version' attribute marked with @Version. The attribute should have the type int, Integer, long, Long, short, Short or java.sql.Timestamp

and its value is controlled by the provider, so it shouldn't be changed manually. JPA provides 2 different optimistic lock modes and two aliases for them:

- **OPTIMISTIC**: OPTIMISTIC is used to obtain an optimistic read lock for all entities containing a version attribute (@Version)
- **OPTIMISTIC_FORCE_INCREMENT**: This option obtains an optimistic lock and additionally increments the version attribute (@Version). However, it's not specified whether the version should be incremented immediately or may be put off until commit or flush.
- **READ**: READ is synonym for OPTIMISTIC
- **WRITE**: WRITE is synonym for OPTIMISTIC_FORCE_INCREMENT

Here is a small example in Java. Let's say we have the following entity:

```
@Entity
public class Person {
    @Id int id;
    @Version int version;
    String name;
    String label;
    @OneToMany(mappedBy = "person", fetch = FetchType.EAGER)
    // One person may have many cars => one to many mapping
    List<Car> cars;
    // getters & setters
}
```

We assume that we have one person in the database called Henry. We make two transactions.

```
//T1 reads Person("Henry")
Person person1 = em1.find(Person.class, id, LockModeType.OPTIMISTIC
    );

//T2 reads Person("Henry")
Person person2 = em2.find(Person.class, id);
//Changing name to "Mikele" within T2
person2.setName("Mikele");
// T2 commits
em2.getTransaction().commit();

// Try to find Henry in the database.
// Throws OptimisticLockException
System.out.println(em1
 .createQuery("SELECT count(p) From Person p where p.name='Henry'")
 .getSingleResult());
```

5.4 Case Study : AWS DynamoDB

DynamoDB is a NoSQL database offered by Amazon as part of the Amazon Web Services portfolio. Similarly to JPA, the SDK for Java (and not for C++) pro-

vides the @DynamoDBVersionAttribute annotation. In a mapping class (concept introduced by DynamoDB, which allows the user to access the data in various tables, using functions like create, read, save etc) for a table, one can designate one property to store the version number. The DynamoDBMapper assigns a version number when you first save the object, and it automatically increments the version number each time you update the item. Your update or delete requests succeed only if the client-side object version matches the corresponding version number of the item in the DynamoDB table. The difference between JPA and DynamoDB is that DynamoDB always increment the version.

5.5 Case Study : Kubernetes

Kubernetes is a popular open source project to deploy, scale, manage and in general operate containerized applications. It uses optimistic locking to manage concurrent updates to resources, such as Pods, Deployment, ConfigMaps etc. within the cluster. Each Kubernetes resource object (like CofigMap) includes a metadata.resouceVersion field. This field contains a numeric identifier that gets updated every time the resource is modified. The Kubernetes API server uses a CAS mechanism to ensure optimistic locking. When a client send an update or delete request with the resourceVersion, the API server checks if the current resourceVersion matches the one provided by the client. If the resourceVersion matches, the update or delete operation proceeds. If it doesn't match, it means that the resource has been modified since the client retrieved it. The API server returns a 409 Conflict error. In this case the client typically needs to retrieve the resource again to obtain the lasest resourceVersion and then reapply the desired changes. Example of resource object:

```
kind: ConfigMap
metadata:
  creationTimestamp: 2024-06-29T21:44:10Z
  name: my-config
  namespace: default
  resourceVersion: "147"
```

5.6 Case Study : Microsoft SQL Server

Microsoft SQL Server is a well-known relational database management system used in variety of software products. It introduces the rowversion datatype and with its help row versioning can be implemented and also optimistic concurrency control. The rowversion data type is unique within the table and can't be manually modified. It replaces the formerly known as timestamp or versioning column, which were depricated. Every time the row is inserted or modified, the row version value is automatically incremented, and the rowversion is replaced with the new one. Using a for loop we can retrieve the row, modify it and try to commit it until the commit succeeds. This is very similar to the example with PostgreSQL and C++ above. Example table with the rowversion data type:

```
CREATE TABLE MyTable (
ID INT PRIMARY KEY,
Name NVARCHAR(50),
RowVersion ROWVERSION
);
```

Other products like MimerSQL, YugabyteDB and GAE (Google App Engine) also claim to use optimistic locking.

Chapter 6

Conclusion

To conclude, optimistic locking achieves great results with read-heavy workloads, because reads typically do not interfere with each other, therefore produces no conflicts, and because we don't have to update a common variable. This allows us to get closer to the best-case, where we don't implement actual mechanisms to protect the threads from one another (which is the most efficient, but error-prone). However, in situations, where more conflicts (e.g write contention) are expected, pessimistic locking remains a very valuable and frequently chosen approach.

We also see that there is not a single solution. It's not just pessimistic or optimistic locking, but sometimes combination of them or even neither of them (like MVCC). Knowledge in advance about what our environment's requirements and limitations are will cause us select optimistic or pessimistic locking for the implementation of a mechanism to control the access to a resource. Every decision in some of the two directions comes with its price.

When simplicity is aimed and higher latency is not a drawback or when we have highly contended resources, then pessimistic locking would be suitable and maybe enough. In other cases, where we need efficiency and we have infrequent conflicts or retry logic is possible and acceptable, we can look at more advanced techniques like the optimistic lock and the ideas it offers.

Many modern database systems and products implement optimistic locking internally or allow to be implemented by the users, giving them flexibility options.

When one chooses the optimistic locking, then he also should think about the additional data structures to handle deferred resources (like the vector used in our epoch-based memory reclamation implementation) and their efficiency. Combinations and variations are also possible around the implementation and usage of these structures.

Literature

1. Haubenschild, Michael; Leis, Viktor: "Lock-Free Buffer Managers Do Not Require Delayed Memory Reclamation"

2. Neumann, Thomas; Freitag, Michael: "Umbra: A Disk-Based System with In-Memory Performance"

3. Leis, Viktor; Haubenschild, Michael; Kemper, Alfons; Neumann, Thomas: "LeanStore: In-Memory Data Management Beyond Main Memory"

4. Leis, Viktor; Haubenschild, Michael; Neumann, Thomas: "Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method"

5. Böttcher, Jan; Leis, Viktor; Giceva, Jana; Neumann, Thomas; Kemper, Alfons: "Scalable and Robust Latches for Database Systems"

6. Brown, Threvor: "Reclaiming Memory for Lock-Free Data Structures: There has to be a Better Way"

7. Leis, Viktor; Scheibner, FLorian; Kemper, Alfons; Neumann, Thomas: "The ART of Practical Synchronization"

8. Fraser, Keir: "Practical lock-freedom"

9. Neumann, Thomas; Mühlbauer, Tobias; Kemper, Alfons: "Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems"