Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Cooperative Memory and Database Transactions

Ricardo Jorge Freire Dias (26579)

Lisboa
(2008)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Cooperative Memory and Database Transactions

Ricardo Jorge Freire Dias (26579)

Orientador: Prof. Doutor João Manuel Santos Lourenço

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática.

Lisboa
(2008)

To my beloved wife Ana

Acknowledgements

First I would like to thank my supervisor Professor João Lourenço for his support and friendship along this year, and mostly important for believe in me.

To Nuno Preguiça for his critiques and excellent tips which help to improve the quality of my work.

To my wife Ana who always stood by my side and gave me the motivation I needed to finish this work.

To my parents who always did everything they could to help me achieving my objectives.

To my colleagues and friends Ricardo Marques, Ricardo Cardoso, Daniel Martins, Francisco Castanheiro, Miguel Figueiredo, Bruno Rodrigues, Vitor Gouveia, Nuno Rocha, Liliana Pratt, Filipa Silva and Alexandra Cabrita for their support.

This work was partially supported by Sun Microsystems and Sun Microsystems Portugal under the "Sun Worldwide Marketing Loaner Agreement #11497".

Summary

Since the introduction of Software Transactional Memory (STM), this topic has received a strong interest by the scientific community, as it has the potential of greatly facilitating concurrent programming by hiding many of the concurrency issues under the transactional layer, being in this way a potential alternative to the lock based constructs, such as mutexes and semaphores. The current practice of STM is based on keeping track of changes made to the memory and, if needed, restoring previous states in case of transaction rollbacks. The operations in a program that can be reversible, by restoring the memory state, are called *transactional* operations. The way that this reversibility necessary to transactional operations is achieved is implementation dependent on the STM libraries being used. Operations that cannot be reversed, such as I/O to external data repositories (e.g., disks) or to the console, are called *non-transactional* operations. Non-transactional operations are usually disallowed inside a memory transaction, because if the transaction aborts their effects cannot be undone. In transactional databases, operations like inserting, removing or transforming data in the database can be undone if executed in the context of a transaction. Since database I/O operations can be reversed, it should be possible to execute those operations in the context of a memory transaction.

To achieve such purpose, a new transactional model unifying memory and database transactions into a single one was defined, implemented, and evaluated. This new transactional model satisfies the properties from both the memory and database transactional models. Programmers can now execute memory and database operations in the same transaction and in case of a transaction rollback, the transaction effects in both the memory and the database are reverted.

Keywords: Transaction, Database Systems, Software Transactional Memory, Concurrency Control.

Sumário

Desde a introdução da Memória Transaccional por Software (STM), este tópico tem recebido grande interesse por parte da comunidade científica, pois a memória transaccional por software tem o potencial de facilitar bastante o problema da programação concorrente através de um aumento do nível de abstracção das primitivas de sincronização, como os semáforos ou *mutexes*. A prática corrente das STMs baseia-se em manter um histórico das modificações da memória durante uma transacção e, se necessário, restaurar o estado anterior caso a transacção aborte. As operações de um programa que são reversíveis, através do restauro de um estado da memória, são chamadas operações transaccionais. A maneira como esta reversibilidade necessária às operações transaccionais é atingida depende da implementação da biblioteca de STM usada. Operações que não são reversíveis, tais como I/O para repositório externo de dados ou para a consola, são chamadas operações não transaccionais. As operações não transaccionais não são usualmente permitidas no contexto de uma transacção em memória, pois os seus efeitos não podem ser revertidos no caso da mesma abortar. Nas bases de dados transaccionais, operações como inserir, remover ou actualizar dados, podem ser revertidas se executadas no contexto de uma transacção. Assim, sendo as operações de I/O para uma base de dados revertíveis, deveria ser possível poder realizar essas mesmas operações no contexto de uma transacção em memória.

Para alcançar tal objectivo, um novo modelo transaccional que unifica as transacções em memória e em base de dados numa única transacção foi desenvolvido, implementado e avaliado. Este novo modelo satisfaz as propriedades garantidas por ambos os modelos transaccionais em memória e em base de dados. Os programadores podem agora executar operações na memória e na base de dados na mesma transacção, e no caso de a transacção abortar, então todos os seus efeitos são revertidos quer na memória como na base de dados.

Palavras-chave: Transacção, Bases de Dados, Memória Transaccional por Software, Controlo de Concorrência.

Contents

List of Figures

List of Tables

1

Introduction

Concurrent programming has been a laborious experience over the past decades, not only due to the appropriate concurrent programming abstractions available but also due to the lack of hardware support. Until recently, computers with more than one processor were only a mirage to the home user and, hence, the development of concurrent applications were not a major requirement to software companies. With the advent of multicore processors, concurrent programming is becoming widespread, and the need of suitable synchronization primitives led to the development of Software Transactional Memory.

1.1 Overview and Motivation

Gordon Moore, back in 1965, has observed that the number of transistors per square inch doubles every 18 months and the rate of growth has been relatively stable since then (Figure 1.1). As the number of transistors was growing, the processor clock frequency also grew as well, but recently, since the advent of CPUs with GHz clock frequency, the growing of clock frequency has been slowing down, even though the number of transistors is still rising.

CPU manufacturers started to incorporate more than one core inside a unique processor, and this trend lead to a situation where there is more processing power than what can be used by most applications. The solution is to explore concurrency in applications whereas sequential processing was used. Now that there is a real demand on exploring concurrency, concurrent programming needs to become a friendly and

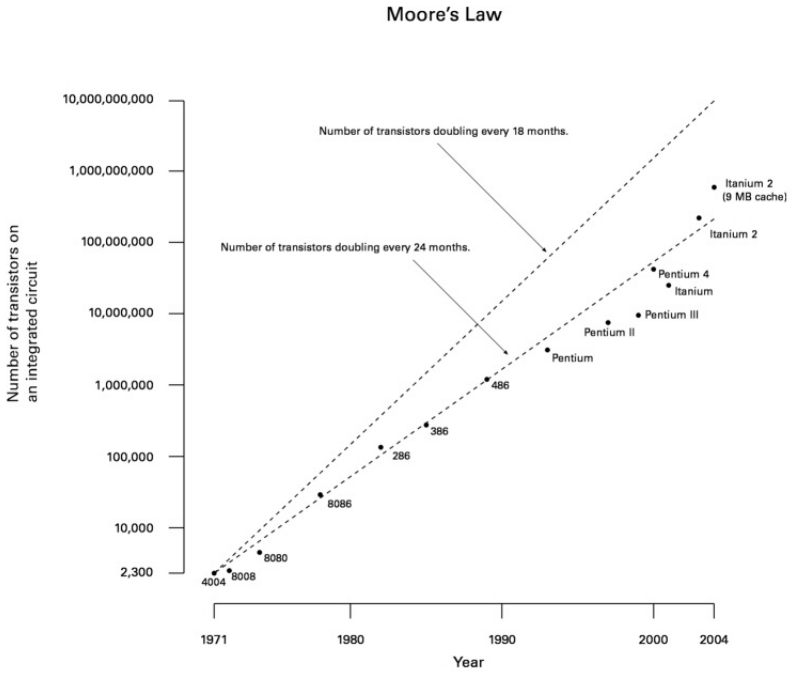

Figure 1.1: Moore's Law Diagram.

easier programming abstraction.

Until now, the practice of multithreaded processing on multiprocessor machines was, in most cases, done in the academic research environments. The reason for this fact is the inherent difficulty of designing and programming highly concurrent programs and data structures. This difficulty is also due to lack of a suitable framework to deal with concurrency: the usual synchronization constructs (locks and condition variables), while simple on the paper, may become unpredictable and error prone in complex systems. Additionally there is a problem with scaling: the use of coarse grained locks in large data structures do not scale. Fine grained locks scale well and but are prone to many concurrent problems, such as deadlocks and priority inversion.

With the increasing number of cores in a single CPU chip, multiprocessor machines are widespreading through home desktop computers and laptops. It is time to start taking advantage of their computational power by developing highly concurrent programs. To succeed in this task it is, however, necessary to rise the abstraction level of synchronization constructs currently available. Software Transactional Memory (STM) is a good approach towards such goal.

In sequential programs, the programmer only had to keep in mind a single control flow. In concurrent programming this single control flow switched into a set of multiple concurrent flows, which are impossible to the common human mind to keep up with. Software transactional memory aims at softening this problem by serializing the

multiple control flows into a single logical one. When using the STM programming model, the programmer is given much of the feeling of the sequential model, which greatly facilitates concurrent programming.

STM has its roots on database transactions which are based in the transactional model. In the transactional model the unit of execution is the atomic code block, enclosing one or more operations. This code block will execute as a single operation and, therefore, will not interleave with other code blocks executing concurrently. If any of the blocks' operation fails, then none will be executed.

Using some properties of database transactions, STM provides the means to synchronize threads efficiently and to avoid lock based problems, such as deadlocks and priority inversion. The current best practice of STM is based on keeping track of changes made to the memory and, if needed, restoring the previous state in case of a transaction rollback. The operations in a program that can be reversed, by restoring the memory state, are called transactional memory operations. The way that this reversibility is achieved is dependent of the implementation of each STM framework being used. Some operations, such as I/O to external data repositories or to console buffers, however, cannot be reversed. Since these types of operations are not *directly* reversible, they should not be used inside memory transactions. To widespread the adoption of the STM programming model it would be desirable to have a simple methodology to adapt the current applications to use this new programming model, and this restriction is a major drawback.

Some applications make use of a data repository to store their permanent data, and for this purpose some use transactional databases. Databases have been well studied in the past decades and they do support all the properties of the transactional model. Databases use transactions to control concurrency accesses and to maintain consistency. If every database access is done in the context of a transaction there is the guarantee that the access will never corrupt the database state and, mostly important, that it can be reverted. To widely accept software transactional memory as programming abstraction, it is important to support database accesses within memory transactions.

1.2 Problem Statement and Work Goals

The current practice of STM is based on keeping track of changes made to the main memory and, if needed, restoring previous states in case of transaction rollbacks. The way that this reversibility is achieved is implementation dependent on the STM frameworks being used. Operations that cannot be reversed, such as I/O to external data repositories (e.g., disks) or to the console buffers, are usually not allowed inside a

memory transaction, as if the transaction aborts their effects cannot be undone.

Another class of I/O operations that are used frequently is database access. Most applications use databases to store data which needs to persist by different instances of the application. Such applications usually depend on the concurrency control mechanisms provided by the database system to synchronize concurrent threads which may exist in the application. In some applications this synchronization is not enough as, sometimes, applications need to synchronize memory data structures together with the database. In these cases, the applications mainly use lock based synchronization mechanisms to prevent concurrent accesses to shared memory data structures and have to, simultaneously, be aware of database transactions when accessing data in the database.

Since STM engines implement a high level abstraction layer to synchronize memory data structures, it would be useful to use them in the context of the applications just described. But first, there is the need to solve the problem of reverting operations on a database in the scope of a memory transaction. In transactional databases, operations like inserting, removing or transforming data in the database can be undone within the context of a transaction. Since database I/O operations can be reversed, it should be possible to execute those operations in the context of a memory transaction.

The purpose of this work is to study the relationships between memory and database transactions, in order to understand how do they affect each other, and how do their coexistence and cooperation be made sound.

Figure 1.2 shows examples of some of the different interleavings between memory and database transactions that will be studied in this work (the memory transaction is delimited by the bracketed `atomic` construct).

```
1 atomic {
2     //...
3     beginDBTransaction();
4     //...
5     endDBTransaction();
6     //...
7 }
```

```
1 beginDBTransaction();
2 //...
3 atomic {
4     //...
5 }
6 //...
7 endDBTransaction();
```

```
1 atomic {
2     //...
3     beginDBTransaction();
4     //...
5 }
6 //...
7 endDBTransaction();
```

```
1 beginDBTransaction();
2 //...
3 atomic {
4     //...
5     endDBTransaction();
6     //...
7 }
```

Figure 1.2: Example of memory and database transactions interleavings.

As depicted in Figure 1.2, the plan is to allow total freedom in intertwining memory and database transactions. To achieve such goal, the cooperation semantics must be defined in order to fully specify a program behavior in such conditions.

1.3 Contributions Of This Thesis

The focus of this dissertion is the cooperation of database transactions with memory transactions. To achieve this, the CTL [Cun07] STM engine was adapted to be able to support apparently irreversible operations, by defining compensating actions, executed in key moments of a memory transaction. This support was then used to support the sound cooperation of database and memory transactions.

The main contributions of this thesis are:

- *Handler system* — The handler system provides mechanisms to define compensation actions to revert the effects of operations that cannot be reversed directly by the STM engine ([DLC08]).
- *Database transactions support* — We present a unified model between database and memory transactions which allows the access to a database repository from within a memory transaction.
- *Identification and resolution of isolation anomalies* — The integration between database transactions and memory transaction introduced some non expected problems related to the *Isolation* property of both memory and database transactions, which were identified and solved.

1.4 Document Outline

This document is divided in the following chapters:

- *Chapter 1*. This Chapter introduced the motivations, the problem under study and highlighted the achieved contributions.
- *Chapter 2*. This Chapter reports on the state of the art in software transactional memory and its foundations on the transactional model implemented in database systems.
- *Chapter 3*. This Chapter covers the specification and implementation of the handler system, and the evaluation of the overhead introduced by this new feature.

- *Chapter 4*. This Chapter covers the integration of database transactions with memory transactions by unification of the two models. The semantics of the new unified model is described as well as its implementation. The serialization problems found are also analyzed and its solution described.
- *Chapter 5*. This Chapter presents a use case example, that will be used to functionally validate the main aim of this work. This same use case example will be used as a benchmark to evaluate the performance of the proposed model against a more conventional lock based approach.
- *Chapter 6*. This Chapter summarizes the results of this investigation and brings out some pointers for future research directions.

2

Related Work

To study the cooperation between memory and database transactions, both models must be studied. This chapter describes both models based on the state of the art literature.

2.1 Transaction Model

Transactions are known for a long time on the database community. They have been very well studied and support a set of standard features with standard semantics.

The transactional model is a high level abstraction to deal with concurrency, as it hides most of its complexity. With the transactional model, a programmer only has to group sequences of operations into transactions. This will ensure that all of the operations in the sequence will execute or that none will and, from a concurrency perspective, that all operations will execute as an unique atomic operation. If there are many processes executing the same set of operations, the transactional layer will execute each set of operations as if it were executing alone in the system as a sequential program.

The transactional model defines four properties known as ACID properties [SKS06]:

- *Atomicity*. Either all operations of the transaction are reflected properly in the system, or none are.

- *Consistency*. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency, meaning that the global

state moves from one consistent state to another.

- *Isolation*. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished its execution before T_i started or T_j started its execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

- *Durability*. After a transaction completes successfully, the changes made to the system will persist, even in the presence of system failures.

A transactional system with these properties allows programmers to develop concurrent programs with minimal changes while using the sequential programming mental scheme. Programs basically become more simple dealing with concurrency control as the transactional model deals with it.

2.1.1 Transaction Life-Cycle

The life-cycle of a transaction is composed by several states as described in [EN00]:

- *Active*. The transaction sequence is currently executing.

- *Partially committed*. The sequence of operations has been concluded and the concurrency controller checks whether there is an interference with other transactions executing concurrently. Furthermore, integrity constraints are also checked by the transactional engine.

- *Committed*. The transaction has been successfully completed.

- *Failed*. The effects of the transaction on the system must be undone.

- *Terminated*. The transaction has been successfully completed or has failed. If the transaction has failed, no effect on the system can be observed.

The state diagram corresponding to a transaction is shown in Figure 2.1. We say that a transaction has committed only if it has entered the *committed* state. Similarly, we say that a transaction has aborted if it has entered the *failed* state. A transaction is said to have *terminated* if it has either committed or failed. Before entering the **active** state a transaction is said to be in a state called **Begin of Transaction (BOT)**.

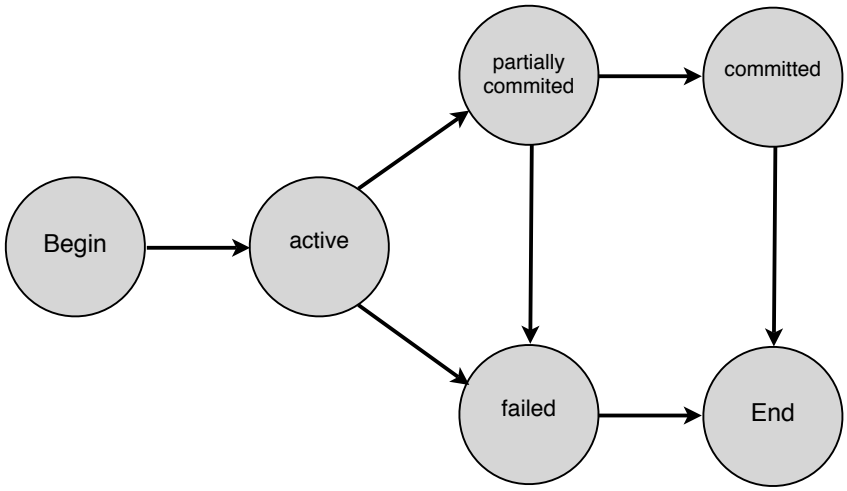

Figure 2.1: State diagram of a transaction.

2.1.2 Nested Transactions

Nested transactions allow to compose two or more previously written transactions into a single new transaction (also referred to as *subtransactions*) [Mos81].

The composition of transactions is not a difficult task if assuming that subtransactions will be executed sequentially but, if we specify that the transactions are to be executed concurrently, then many problems may arise [Mos81]. To solve these problems the composition should be run as a transaction in its own right, but also provide concurrency control within the transaction [Mos81]. The natural method is to consider the whole transaction as a microcosm and synchronize its components (called its subtransactions) with respect to each other in the way whole transactions are synchronized. This results in two levels of transactions: *top-level* transactions, the kind of transactions discussed in the previous section; and subtransactions, the separate pieces composed to make new composite top-level transactions.

A transaction may contain any number of subtransactions, and every subtransaction, in turn, may be composed of any number of subtransactions, resulting in an arbitrary deep hierarchy of nested transactions.

As the top-level transactions guarantee the ACID properties, for subtransactions it is sufficient to provide weaker properties [HR87]. A subtransaction appears atomic to the surrounding transaction and may commit or abort independently. It may abort without affecting the outcome of the surrounding transaction. However, the commit of a subtransaction is totally dependent of its superiors implementation, even if it commits, aborting one of its superiors will undo its effects. All updates of a subtransaction become permanent only when the enclosing top-level transaction commits. Furthermore, it would be unnecessarily restrictive to require consistency to be preserved by

a subtransaction. Therefore, subtransactions only comply to the *atomicity* and *isolation* properties.

There are three main semantic models for nested transactions [ALS06]: *flat nested transactions*, *close nested transactions* and *open nested transactions*.

Flat Nested Transactions

When flat nested transactions semantic model is used, each subtransaction is *inlined* within the parent transaction. If a subtransaction aborts the parent transaction will abort as well, since the subtransactions operations (reads and writes) are also part of the parent transaction. When the subtransaction commits the changes are only visible to the parent transaction and only when the top-level transaction commits the changes become visible to other top-level transactions.

Close Nested Transactions

This semantic model is the most intuitive, and usually, when we refer to nested transaction only, we are referring to close nested transactions. As described in [MH06], in this model, only transactions with no current children can access data; such accesses are either reads or writes. Logically, the (globally committed) data are not updated until a top-level transactions commits. When a transaction reads a value, it sees the value in the current context, otherwise the value seen by its parent. In this model, two transactions conflict if both access the same data and at least one of the accesses is a write, and neither transaction is an ancestor of the other.

When a subtransaction commits, then all of its writes become permanent in the context of its parent, when a top-level transaction commits then all its writes become permanent in the (global) system. When a transaction aborts, either for a subtransaction or a top-level transaction, its reads and writes are simply discarded. Importantly, a child transaction abort does not abort its parent, though the parent may be notified of the abort and take alternative action of its choosing, including aborting itself. One interesting property of this model is that a child's operation will never conflict with its parent (or any other ancestor).

Open Nested Transactions

Open nested transaction and closed nested transaction models are quite similar. However, in the former case, the parent and child execute at *different levels of abstraction* [Mos06]. Sometimes many conflicts between transactions at the level of memory words are what is called *false conflicts*: conflicts that occur at low level of abstraction, but which

are not essential to the semantics of the operations. The subtransactions work with a low level of abstraction instead of the parent transactions that work at a high level of abstraction.

When a subtransaction aborts it simply discards the changes made, as in the case of close nested transactions, and does not abort its parent transaction.

The changes committed by a subtransaction immediately become visible to other top-level transactions. If the parent transaction decides to abort after a committed subtransaction, then the parent has to undo the subtransactions effects at a high level of abstraction, using compensating transactions, and not just simple return to the state prior to their changes. Thus, from the standpoint of the lower level, the upper level undo is *forward progress*.

Linear Nesting

Linear nesting [MH06] is a model for nesting transactions in which the subtransactions of one transaction do not execute concurrently between each others. The semantic model could be one of the three semantics described in previous sections. In this model, top-level transactions continue to execute concurrently. Another issue is that, when a subtransaction starts its execution, the parent transaction stops and waits for the finish of the subtransaction, and after that continues its execution.

2.2 Concurrency Control

One of the goals of the concurrency control is to prevent that concurrent execution cause unpredicted interleavings in the application programs which lead to malfunction. This objective addresses the consistency (C) and isolation (I) of the ACID properties.

The consistency property can be seen as a set of object invariants that define the conditions that an operation has to satisfy when modifying the system state. To modify the system state, often the state must be made temporarily inconsistent while it is being transformed into a new consistent state. A transaction that is started with a consistent system state may make the state temporarily inconsistent, but will always terminate by producing a new consistent state. If a set of transactions are executed one at a time (sequentially), no transaction sees the inconsistent state of any other, this property may not hold if multiple transactions are executed concurrently. To prevent this situation, each transaction has to execute *isolated* from the others in order to give each one a permanent consistent view of the system state.

The isolation can be achieved by applying the serializability theory, formalized by

Eswaran et al. [EGLT76], which defines how to schedule the actions of two concurrent transactions in order to appear that they were executed one after another.

The example in Figure 2.2 shows three schedules (S_1, S_2, and S_3) of two transactions (T_1 and T_2) where the invariant $A = B$ should hold. The schedule S_1 is the serial schedule in which T_1 executes before T_2. The schedule S_2 is equivalent to S_1 because, the invariant is preserved at the end of the execution. The schedule S_3 does not preserve the invariant, therefore is not a valid schedule.

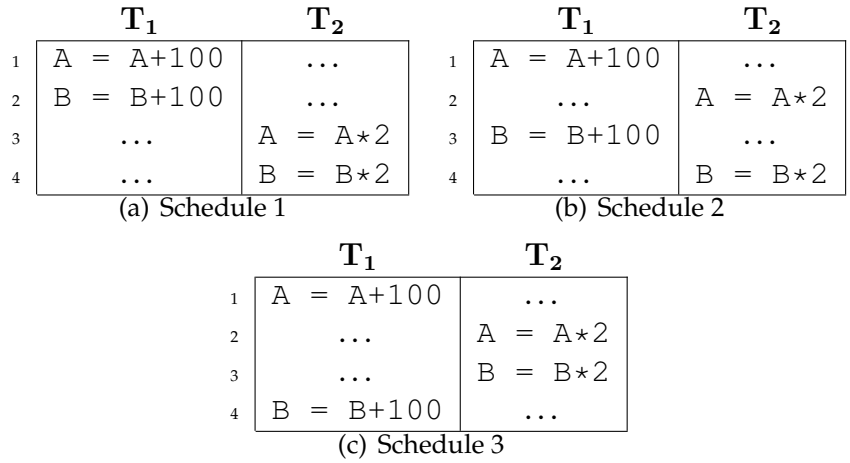

Figure 2.2: Example of three different schedules for **T1** and **T2**.

It is very difficult to know *a priori* which data will be accessed by a transaction, hence, generating schedules before execution is difficult as well [GR92]. The schedules must be dynamically generated as the execution progresses. Consistent dynamically generated schedules can be achieved with two main synchronization approaches: blocking and non-blocking synchronization.

2.2.1 Blocking Synchronization

In a concurrent environment, a blocking synchronization approach guarantees mutual exclusion access, to the same data, by different processes. If a process tries to access shared data which is already being accessed by another process, then it has to wait for the data to be freed. This type of approach uses lock based techniques to prevent other processes from accessing the same data. Locks are acquired by a process before accessing the shared data and released after the operation is complete.

Blocking synchronization may suffer from problems of deadlocking and priority inversion, inherited from lock based techniques.

2.2.2 Non-Blocking Synchronization

Non-blocking synchronization guarantees progress of the system as a whole independently of the interleavings or any process failures. Non-blocking algorithms are designed under the assumption that synchronization conflicts are rare and should be handled only as exceptions. When a synchronization conflict is found then the operation/transaction has to be restarted.

Non-blocking algorithms can be classified according to the kind of progress guarantee that they make [FH07, HLMWNS03]:

- *Obstruction freedom*: "is the weakest guarantee: a thread performing an operation on the data structure is only guaranteed to make progress as long as it does not contend with other threads for access to any location (...). This requires an out-of-band mechanism to avoid livelock; exponential backoff is one option."

- *Lock freedom*: ensures that at least one thread always makes progress hence "the system as a whole makes progress, even if there is contention. (...) This is sufficient to prevent livelock, although it does not offer any guarantee of per-thread fairness."

- *Wait freedom*: "adds the requirement that every thread makes progress, even if it experiences contention." "It ensures that every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads."

A wait free algorithm is always lock free and a lock free algorithm is always obstruction free, but not the other way around. Non-blocking synchronization mechanisms avoid liveness problems such as deadlock and priority inversion [HM93], but do not necessarily avoid other problems like *livelock*.

2.2.3 Read Write Locks

In this lock based technique there are two types of lock acquisition: one for read access and another for write access. Read (or shared) locks can be acquired, over the same object, by different processes in a shared mode. In opposition to write (or exclusive) locks which can only be acquired, over the same object, by different processes in an exclusive mode. A process can acquire a read lock, over an object, if there is not any type of lock already acquired by other process, or if the lock acquired by other process is also a read lock. A process can acquire a write lock only if there is not any type of lock already acquired by other process. The table 2.2.3 summarizes the acquire policy.

	Read Lock	**Write Lock**
Read Lock	Yes	No
Write Lock	No	No

Table 2.1: Table with the lock acquisition possible combinations.

When a process has a read lock over an object, it can upgrade it to a write lock only if no other process has also a read lock over the same object. If there are other processes that have read locks over the same object, then the process has to wait for the other processes to release the read locks. This operation must not be used frequently because it is very expensive in terms of performance [SATH+06]. Another negative hit on performance is the fact that for each read, it is necessary to acquire a lock. Other disadvantage of this technique is that it is prone to the *starvation*, *priority inversion* and *deadlock* phenomenas.

Frequently, when this technique is used, it is used in conjunction with the *two-phase locking* protocol. The two-phase locking protocol is a necessary and sufficient condition to generate serializable schedules [EGLT76]. This protocol states that there are two phases: the *growing* phase and the *shrinking* phase. In the growing phase the process only acquires locks, but may not release any lock. In the shrinking phase the process only releases locks, but may not acquire any new locks. The problem with *two-phase locking* is that it lowers the level of concurrency. To increase the concurrency level different techniques have to be used in trade off of getting inconsistent schedules.

This type of technique is also called pessimistic concurrency control, as it always blocks if a concurrency conflict arises.

2.2.4 Timestamp Ordering

This technique is an approach to resolve the deadlock problem of lock based techniques. Each transaction T is assigned with a timestamp $TS(T)$. For each transaction T_i with timestamp $TS(T_i)$ it is possible to create a total order of requests from transactions according to transactions' timestamps [RSPML78]. If two transactions (T_1 and T_2) execute concurrently, and T_1 requests access to a data item x that is already being accessed by T_2, then transaction T_1 has three options: T_1 waits for T_2 to finish its access to x, T_1 aborts itself and restarts or T_2 is preempted and T_1 gets hold of x. The action taken by T_1 is decided by a schedule protocol by comparing the timestamps of the two transactions. Two possible scheduling protocols are [BK91]:

- *WAIT-DIE*: "which forces a transaction to wait if it conflicts with a running transaction whose timestamp is more recent or to die (abort and restart) if the running transaction's timestamp is older (...)."

- *WOUND-WAIT*: "which allows a transaction to wound (preempt by suspending) a running one with a more recent timestamp or forces the requesting transaction to wait otherwise (...)."

Both protocols use implicit locking when forcing transactions to wait for one another and also both guarantee that a deadlock situation will not arise.

The timestamp ordering technique ensures freedom from deadlock, since no transaction waits forever. However, it does not solve the problem of starvation.

2.2.5 Multiversion Timestamp Ordering

The main idea of this technique is that every object in the system is considered to have a time history and object addresses become <*name*, *time*> rather than simply < *name* >. In such a system, an object is never updated or deleted, instead is created an history of versions of the same object in which each version is indexed with the timestamp of the transaction that produced that version [Gra81, BK91].

When a transaction issues a read operation over an object, the timestamp of the transaction is added to a readset of the object. The readset of an object has all the timestamps of the transaction that read the object. When a transaction successfully writes a new value to an object, then a new version of the object is created with the same timestamp as that of the transaction requesting the write operation.

This concurrency control technique operates as follows:

Let T_i be a transaction with timestamp $TS(T_i)$. If T_i issues a read operation over an object $Read(x)$, the value returned by the read operation is the version of the object x whose timestamp is the largest timestamp smaller than $TS(T_i)$ (i.e., the latest value written before T_i started). $TS(T_i)$ is then added to the set of read timestamps of the object. The read operation is always permitted. If T_i issues a write operation over an object $Write(x, v)$ that assigns a value v to the object x, the write operation will be permitted only if other transactions with a more recent timestamp than $TS(T_i)$ have not read a version of the object x whose timestamp is greater than $TS(T_i)$. If this condition is not satisfied then T_i has to be aborted.

The multiversion timestamp ordering technique has the desirable property that a read request never fails and is never made to wait. However suffers from two undesirable properties: first, the reading of an object also requires an update of the object read timestamp set; and, second, the conflicts between transactions are resolved with rollbacks which may be an expensive alternative.

2.2.6 Optimistic methods

Optimistic methods are *optimistic* in the sense that for efficiency, they rely on the hope that conflicts between transactions will not occur [KR81].

These methods have three phases:

- *Read phase*: During this phase, a transaction issues read operations storing the read values in the readset. And all write operations are done in a private shadow copy of the objects to be written.
- *Validation phase*: Using the readset of the transaction the concurrency control validates if there is a serializability conflict with respect to all committed transactions.
- *Write phase*: If the validation phase was successful then the objects are written with the values stored in the writeset.

Each transaction is assigned with a timestamp once the read phase has concluded. Each transaction also has a *readset* and a *writeset*. The readset is used to record the timestamps of every data item it reads. The writeset is used to record the value which is intended to write in the write phase. In the validation phase a timestamp ordering based protocol is used to verify the serializability conflicts using the readset of the transaction.

One of the methods in this category is the versioned write locks. In versioned write locks technique, each data item has a timestamp number that is updated for every committed transaction which updated it. When a transaction issues a read operation then the data item timestamp is added to its readset. On commit, it is verified if the timestamps of the data item accessed by the transaction were changed by comparing the actual timestamp with the one recorded in the transaction's readset. If any of the timestamps have changed, the transaction must be rolled back, otherwise it may commit. This prevents non-serializable orderings from committing.

The usage of versioned write locks makes reads invisible to writing transactions, as a consequence, writes may conflict with reads, however the transaction that made the conflicting read will abort.

2.3 Transaction Recovery

To achieve the atomicity property, and besides the use of concurrency controls mechanisms to guarantee serializable schedules, there is the need to provide recovery mechanisms to transactions which suffer from conflicts and need to abort and roll back the

execution. Logging techniques are used to keep track of changes made by the transaction during its execution. When a transaction needs to roll back, then these logs are used to recover the system to the state prior to the transaction execution. There are two types of logging techniques: the *undo log* and the *redo log*. Each one of these logging techniques keep track of the values that are written to the system and the previous values that were substituted by the new ones. These changes are recorded in a write log or update log. The difference between the two techniques is mainly in the way that this write log is managed.

2.3.1 Undo Log

In the undo log technique, every write operation, made by a transaction, records the current value of the data item in an update log and then substitutes it by the new tentative value. This strategy is also called *in-place update* or *direct-update*.

When a transaction aborts and it is rolled back, then with the use of the update log, each data item value is rewritten with the previous value in backwards order of their update. When all the update log entries were processed then the data is restored to the state it had before the transaction began. When the transaction commits, the update log is simply discarded.

2.3.2 Redo Log

In the redo log technique, every write operation, made by a transaction, writes the new tentative value in the update log and the data item value remains unchanged. This strategy is also called *out-of-place update* or *deferred-update*.

When a transaction wants to read a data item value, first it has to look at the update log to check weather there is an entry for such data item. If that is the case, it reads the value in the update log, otherwise it reads directly from the data item.

When a transaction aborts and it is rolled back, the transaction only has to discard the update log, since it did not change any data item value and the system is still in the state it had before the transaction began. When the transaction commits, it has to read the tentative values from the update log in forward order and write these values into the data items.

2.4 Database Transactions

Currently, there are several Database Management Systems (DBMSs) that implement the transactional model. The main difference between them is in terms of the concur-

rency controls used.

The PostgreSQL [Pos] is a database management system which uses multiversion concurrency control (MVCC) and two phase locking as for base techniques. The MVCC is based on the multiversion timestamp ordering described in Section 2.2.5. The Oracle [Ora] also uses a MVCC based concurrency control. In Oracle concurrency model, read operations do not block write operations and write operations do not block read operations, this property allows a high degree of concurrency. The DB2 [DB2] uses lock based concurrency control techniques. DB2 implements strict two-phase locking for all update transactions, hence write locks are held until commit or rollback time. MySQL [MyS] also uses a lock base concurrency control methodology.

2.4.1 Isolation Levels

Database systems implement the transactional model, the concurrency control and the recovery strategies. The purpose of the concurrency control is to achieve the atomicity, isolation and consistency properties of transactions. But a concurrency control that always generates serializable transaction schedules is very restrictive in terms of throughput. Therefore the `ANSI SQL-92` defines four isolation levels to be implemented by the database systems [BBG+95].

The isolation levels are defined by the `ANSI SQL-92` in terms of three *phenomena* that must be prevented between concurrent transactions: *dirty read*, *non-repeatable read* and *phantom read*.

The dirty read occurs when a transaction reads data written by a concurrent uncommitted transaction. The non-repeatable read occurs when a transaction rereads data it has previously read and finds that another committed transaction modified or deleted the data. The phantom read occurs when a transaction re-executes a query returning a set of rows that satisfies a search condition and finds that another committed transaction has inserted additional rows that satisfy the condition.

Given these three types of phenomena, four isolation levels were defined, which are: *read uncommitted*, *read committed*, *repeatable read* and *serializable*. The table 2.2 describes the relation between the isolation levels and the phenomena.

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read
Read Uncommitted	possible	possible	possible
Read Committed	not possible	possible	possible
Repeatable Read	not possible	not possible	possible
Serializable	not possible	not possible	not possible

Table 2.2: Transaction isolation levels

Each database system chooses which isolation levels to implement, and not all databases implement all the four isolation levels.

PostgreSQL [Pos] and Oracle [Ora] implements only the read committed and serializable isolation levels. MySQL [MyS] implements all isolation levels. DB2 [DB2] implements different alternative isolation levels which will not be discussed in this work.

2.4.2 Snapshot Isolation

In fact, PostgreSQL and Oracle implement a relaxed version of the *serializable* isolation level. The isolation level implemented is provided by a multiversion concurrency control mechanism called *Snapshot Isolation* [BBG+95]. The idea of *snapshot isolation* is to take a consistent snapshot of the data at the time the transaction starts, and all read and write accesses, made by the transactions, are done in this snapshot.

When an update transaction tries to commit, it has to get a unique timestamp that is larger than any existing start or commit timestamp. In this way, when two update transactions try to commit at the same time, the first to get the commit timestamp wins and commits the updated data, this protocol is based on the *first-committer-wins* principle [BBG+95].

All the read accesses made by transactions are made in the snapshot, hence they will never conflict with concurrent update transactions. This raises the problem that read-/write conflicts are never detected, and in particular the generated schedules which are not serializable.

Snapshot isolation does not guarantee serializability but avoids the common isolation anomalies described in the previous section.

2.4.3 Database and Distributed Transactions

Cooperation between database transactions and distributed transactions is an already known problem in applications servers. The need to coordinate the data flow between different databases atomically raised the concept of transaction managers which operate on an external area from databases.

The cooperation between a database transaction and a distributed transaction is done by the transaction manager which controls the *begin*, *commit* or *abort* of both transactions. When a distributed transaction begins, automatically is initiated a database transaction, and every operation to the database is done within the context of such a transaction.

To commit a distributed transaction, which must commit several database transactions atomically: either every transaction commits or every transaction aborts. The *two-phase commit* protocol [Gra78] provides a way to achieve this purpose.

In the *two-phase-commit protocol* , all transactions must first agree whether they will commit or abort. This phase is called the preparation phase. After this phase all the transactions must perform the decision agreed in the previous phase. If the decision was to commit then the transaction manager will request that all the transactions commit, otherwise if the decision was to abort, then the transaction manager will request that all the transactions abort. This phase is the commit phase.

With the use of this protocol it is possible to commit several transactions atomically creating the notion of distributed transaction. This model of integration is an inspiration source to the integration and cooperation problems between memory and database transactions planned for this work.

2.5 Transactional Memory

Concurrent programming requires constructs to control accesses to critical sections which may originate data races. Typically these constructs are lock based. Locks provide mutual exclusion to critical sections and are easy to use when programming small concurrent applications. When dealing with bigger concurrent applications, locks may become difficult to use because of their inherent problems like deadlock and lack of composability.

If a concurrent linked list implemented with locks is used in an operation to remove an item from one list and insert that item into another list, without exposing the intermediate state where the item is in neither of the lists, then there is the need to access the list's concurrency control implementation and break the abstraction of this data structure.

Lock free implementations, although being deadlock free, do not compose well and are very difficult to implement. Therefore, there is space for an alternative concurrency control technique to ease the programming of concurrent applications.

Transactional Memory (TM) was first introduced by Herlihy *et al.* [HM93] as a hardware architecture. Transactional memory differs from database transactions in the way that data items are stored. Instead of being in a secondary memory persistent storage, data items are stored in primary memory. Another difference is the requirement of a much higher performance in executing memory transactions.

Transactional memory does not ensure all the ACID properties [HMPJH05]. Since the operations over data items are intentionally stored on volatile storage, there is no

need to ensure the durability property. Relational database systems have a very strict data organization—tables, rows, etc. and have standard integrity rules, like referential integrity. In this scheme consistency property is very important but in TM, data structures are more flexible and cannot have consistency verification in the same way as database transactions do. Therefore consistency property is more relaxed in TM than in database transactions. TM requires three properties: atomicity, consistency (relaxed) and isolation.

TM operates on memory words, controlling only two operations, read and write, and uses the same techniques as database transactions to achieve the atomicity and isolation properties. Earlier TMs were created using non-blocking synchronization techniques based on atomic hardware operations [HM93, ST95] but the architectural support for this model remains subject of ongoing research [HMPJH05].

Software Transactional Memory, based on TM, controls concurrency by using only software constructs, and aims at overcoming the inflexibility of hardware operations.

2.6 Software Transactional Memory

Software Transactional Memory (STM) was first introduced by Nir Shavit and Dan Touitou in 1995 paper [ST95]. The methodology is the same of transactional memory but applied in software. Although the performance is smaller than in Hardware TM, it has the advantages of applicability and portability among today's machines.

Using syntactical constructs such as library calls or language keywords, STM provides the means to drop the complexity of developing concurrent programs. There are three types of syntactical constructs used by most STMs to define atomic blocks of code: library calls, language keywords and function wrapping. Figure 2.3(a) shows an example of a transaction defined by a library call, first introduced by [HLMWNS03]. Figure 2.3(b) illustrates the same content but defined by the `atomic` programming language keyword. The code inside the `atomic` block is going to execute as a transaction. This approach was introduced by [HMPJH05]. Figure 2.3(c) shows an example of a transaction defined within the scope of a function. The execution of the transaction, defined by this way, is controlled via a transaction manager. This approach was introduced by [HLM06].

Independently of the construct type used, the code block must execute within a transaction and must hold the three ACI properties. If the transaction returns successfully, then all the actions of the transaction were executed and the transaction did not interfere with any other transaction. Otherwise, if the transaction did not return successfully, then none of its actions did take effect. These constructs rise the abstrac-

```
1 BeginTransaction();
2  action1();
3  action2();
4  action3();
5 EndTransaction();
```

(a) Transaction defined by library call.

```
1 atomic {
2  action1();
3  action2();
4  action3();
5 }
```

(b) Transaction defined by an atomic block.

```
1 int TransactionXPTO {
2  action1();
3  action2();
4  action3();
5  return 1;
6 }
7
8 //...
9 CallTransaction(TransactionXPTO);
```

(c) Transaction defined by a function.

Figure 2.3: Examples of STM constructs.

tion level of concurrent programming as they hide the implementation issues of thread synchronization.

The semantics of an atomic block is a more complex subject and it has not yet been formally specified, although some attempts have been made [Sco06]. Database transactions execute in a closed environment, and applications do all the computation and invoke database transactions to store or retrieve data. There is no connection between the application environment and the database transactions environment. In contrast, memory transactions are strongly coupled to the application environment and, therefore, its semantics should consider other issues. Examples are the language features like `retry` and `orElse` (see Sections 2.6.2), the interaction between code inside atomic blocks and the non-transactional code outside atomic blocks, or irrevocable actions (see Section 2.6.5).

The semantics of a transaction can be easily described with an existing global lock which every transaction has to acquire when started, hindering in this way other transactions from executing. A transaction then releases the lock once the last operation is concluded. Using this global lock, transactions are serialized one after another. Of course this is just an abstraction to describe the transaction semantics, as it would eliminate any form of concurrency and destroy the purpose of transactions in the first place.

Another behavior that must be identified and described relates to a transaction which is aborted explicitly or by means of a conflict with other transaction. There are two possible semantics: the *exactly-once* semantics and the *at-most-once* semantics. The exactly-once semantics implies that, if a transaction is aborted, it must be restarted,

until it commits. The at-most-once semantics implies that, if a transaction is aborted, it does not restart again unless the application code explicitly restarts it.

2.6.1 Non-Transactional Accesses

Non-transactional accesses happen when a shared variable is concurrently accessed by code inside of an atomic block (transactional access) and also by code outside of an atomic block, and they both modify the shared variable originating a data race. The behavior of such code containing data races is undefined and depends on the implementation of a computer's memory consistency model. To understand more clearly this phenomenon take the example with locks in Figure 2.4. $Thread_1$ increments variable x inside an atomic block, and $Thread_2$ change variable x value without any transactional protection. Since the behavior is unpredictable, some STM engines decided to not permit such accesses [HLMWNS03, HLM06], others give the programmer the responsibility of avoiding non-transactional accesses, others will execute all the operations in the context of a transaction.

$Thread_1$

```
1 atomic {
2     y = x;
3     x++;
4     if (x != y+1) fireMissiles();
5 }
```

$Thread_2$

```
1 // ...
2 x = 2;
3 // ...
```

Figure 2.4: Example of non-transactional access with locks.

Another issue presented in [BLM05, MBL06] shows that different semantics have to be taken into account when dealing with non-transactional code. The author defines two transactional semantics: strong and weak atomicity. Strong atomicity is defined as a transactional semantics that gives the guarantee of atomicity between transactional and non-transactional code. Weak atomicity, is a transactional semantics in which transactions are atomic only with respect to other transactions (i.e., their execution may be interleaved with non-transactional code). When a program is executing transactional code, it is important to know which semantics is implemented because a program may show different behavior between the two semantics and it has been shown that in some cases it could deadlock.

2.6.2 Software Transactional Memory Features

Conditional Waiting

Conditional waiting is a programming technique widely used to coordinate the execution of multiple threads. It uses conditional variables and allows a thread to wait until a certain condition becomes true. Harris *et al.* [HF03] have introduced the *conditional critical region* (CCR), this kind of functionality was not available to plain transactions, and no transaction could be coordinated in respect to some shared condition/variable. A CCR is a block of code in which a transaction must wait for a single condition to be true, before starting executing the block. Later, another approach was used by the same authors [HMPJH05] which introduced the `retry` statement to coordinate transactions. Figure 2.5 illustrates a shared buffer implemented with the two approaches.

Conditional Critical Region

```
1  int stack[MAX_STACK];
2  int items = 0;
3
4  void push(int d) {
5      atomic {
6          stack[items] = d;
7          items++;
8      }
9  }
10
11 int pop() {
12     int d;
13     atomic(items > 0) {
14         d = stack[items];
15         items--;
16     }
17     return d;
18 }
```

`retry` Statement

```
1  int stack[MAX_STACK];
2  int items = 0;
3
4  void push(int d) {
5      atomic {
6          stack[items] = d;
7          items++;
8      }
9  }
10
11 int pop() {
12     int d;
13     atomic {
14         if (items > 0)
15             d = stack[items];
16         else
17             retry;
18         items--;
19     }
20     return d;
21 }
```

Figure 2.5: Example of two approaches for transaction coordination.

The CCR semantics is simple: the transaction has to wait while the condition is false. Once the condition becomes true, the transaction may execute. The semantics of the `retry` statement is more complex: the transaction executes normally, but when the `retry` statement is called, the transaction is rolled back and waits until any of the variables read in the previous execution is modified. Using the previous example in

Figure 2.5, if a transaction retries, it only restarts when at least one of the variables `items` or `stack` changes. Since the variable `items` and `stack` are always modified at the same time when some thread pushes an element on to the stack, the waiting transaction will restart only after the returning of the `push` function. The `retry` statement is more flexible than the CCR approach as the logic before a `retry` is issued can be more complex than a simple expression, e.g., it is not easy with the CCR approach to wait for all the elements of an array to have a specific value.

Another STM engine named Atomos [CMC+06] also has the `retry` statement, but it allows to specify which locations (*watch set*) that trigger the re-execution of an atomic block. With a watch set is possible to reduce the locations space search, thus improving the performance.

Composability

One important feature of STM is *composability*. Composition is the process of creating software from components—abstractions whose internal details are hidden. Defining a block of code to execute as a transaction, ignoring the implementation details, allows the ability of transaction composability. Lacking of such a feature, programming is far more difficult and error-prone as the programmer cannot rely on the specified interface of an object and instead must understand its implementation. Take as an example the problem of transferring one item from one linked list into another, which was not possible with lock based synchronization without exposing the data structure concurrency control implementation. Using STM composability feature, it is possible to execute this operation considering that each linked list operation (insert, remove, etc...) is implemented as a transaction, and the transferring operation as an enclosing transaction as well. Figure 2.6 shows an example of such item transfer operation using STM methodology. This examples shows that the intermediate state, where the item is in neither of the lists, is not seen by any other transaction because it is enclosed in an atomic block.

```
1 void switchItem(List l1, List l2, int pos) {
2     atomic {
3         Item i;
4         i = remove(l1,pos);
5         if (i != null) //if item exists
6             add(l2, i);
7     }
8 }
```

Figure 2.6: Linked list item transfer operation.

Composable Alternatives

Harris *et al.* [HMPJH05] introduced the `orElse` statement. This statement is another mechanism to coordinate transactions, in conjunction with the `retry` statement, and allows to compose alternatives. In the following example, in Figure 2.7, illustrates the use of the `orElse` primitive. The example shows a transaction waiting for data from two different stacks using the `orElse` statement. The `pop` function is the same used in Figure 2.5 with the `retry` statement. Since `b1.pop()` and `b2.pop()` are both transactions, then `b1.pop()` **`orElse`** `b2.pop()` starts by executing `b1.pop()`. If `b1.pop()` returns by committing or aborting the transaction then the `orElse` finishes without executing `b2.pop()`. If `b1.pop()` blocks by executing a `retry` statement, then the `orElse` starts executing the transaction `b2.pop()`. If `b2.pop()` returns by committing or aborting the transaction then the `orElse` finishes execution. If `b2.pop()` blocks by executing a `retry` statement, then the `orElse` waits for a location read by either `b1.pop()` or `b2.pop()` changes, and then re-executes itself in the same manner.

`orElse` Statement

```
1  void waitForStacks(Stack b1, Stack b2) {
2      atomic {
3          int d;
4          do {
5              d = b1.pop();
6              orElse
7              d = b2.pop();
8          } while(...);
9      }
10 }
```

Figure 2.7: Example of the usage of the `orElse` statement.

2.6.3 Data Granularity

STM engines must define which storage unit will be used to detect conflicts between transactions. There are two main types of granularity: *word-based*, which detect conflicting accesses to a memory word or adjacent, fixed-size group of words; and *object-based*, which detects a conflicting access to an object even if the transactions referenced different fields. Both of them must store metadata associated with each grain unit.

Word-Based

Word-based or block-based STMs have been first proposed by Harris *et al.* in [HF03]. This type of granularity has the size of a memory word or the size of a cache line. These STMs have more room for concurrency as the granularity is finer. Since the granularity is finer, it allows more concurrency between transactions and conflicts are less frequent. The metadata associated with each word must be stored in an auxiliary table, such as a hash table. In this case, mapping from a memory address to metadata can be expensive, reducing the overall performance. This type of granularity is particularly important for aggregate data structures, such as arrays, which many transactions may concurrently access if they can be logically partitioned.

Object-Based

Object-based STMs [HLMWNS03] have a coarser granularity than word-based STMs. The size of object-based granularity is the size of an object. The metadata is stored adjacent to the object in the form of an hidden field of the object. Simultaneous accesses from different transactions to different fields of the same objects may conflict. This type of granularity gives more performance than word-based, since it does not need to call the transactional engine every time that a transaction accesses some item, but it also lowers the concurrency level between transactions.

2.6.4 Nested Transactions

There are three semantic models of nested transactions implemented in STM engines. They are as those described in Section 2.1.2 and, to our knowledge, all of them follow the linear nesting model which is easier to implement in a programing language environment.

2.6.5 Irrevocable Actions

STM transactions execute an atomic code block and if everything went well they commit; if something went wrong they abort. The abort of a transaction implies that all the changes made by the transaction must be undone, hence every operation inside an atomic code block must be reversible. However, some operations are not reversible, like I/O to external data repositories (e.g., disks) or operating system calls which change the state of kernel data structures. Consequently they should not be allowed. Figure 2.8 shows an example of an irrevocable operation that cannot be undone if the

transaction aborts. Until now, there is no general solution to all these types of irrevocable operations, solutions exist on a case-by-case basis.

```
1 void irrevocablePrint() {
2     atomic {
3         //...
4         System.out.println("hello world!")
5         //...
6     }
7 }
```

Figure 2.8: Example of an irrevocable operation.

Preventing I/O operations from occurring inside transactions is not an easy task for the STM engines and, therefore, this responsibility is frequently left to the programmer. One exception is the STM Haskell [HMPJH05] which has a very expressive type system and divides all operations into either transactional operations or I/O operations. The compiler statically prevents I/O operations from occurring inside memory transactions.

Another solution is to allow I/O operations in transactions only if the I/O supports transactional semantics, so the STM system can rely on another abstraction to revert changes. Databases and some file systems are transactional. However, the granularity of these systems' transactions may not match the requirements of an atomic block.

In transactional databases, operations like inserting, removing or transforming data in the database can be undone if done in the context of a transaction. Since database I/O operations can be reversed, it should be possible to execute those operations in the context of a memory transaction[1].

Another approach is to use compensating operations, in order to undo the changes made by transactions. But such compensating operations are very difficult to write because of some operations' complex semantics, and some cannot be compensated. Chapter 3 reports on supporting compensating actions for an STM framework.

2.6.6 Case Studies

This section presents a brief description of some STM engines and their characteristics in terms of the properties described in previous sections.

[1] This is the main theme of this dissertation.

TL2

The TL2 STM engine was presented in [DSS06]. Table 2.3 summarizes the properties of this engine.

Strong or Weak Atomicity	Weak
Granularity	Word-based
Recovery Strategy	Deferred Update
Concurrency Control	Optimistic (Versioned Write Locks)
Synchronization	Blocking
Inconsistent Reads	Validation
Nested Transaction	Not supported

Table 2.3: TL2 engine properties.

This STM engine has word-based granularity and does not prevent non-transactional accesses. It uses a deferred update method as recovery strategy and, for concurrency control, uses a variant of versioned write locks (see Section 2.2.6). This engine was implemented as a library for C programming language and has the advantage of having transactional memory blocks stored at the same memory space as normal blocks allocated by non-transactional code. This fact allows transactional blocks to be recycled and to be used by non-transactional code with the help of a *quiesce* algorithm, which waits for all activity of some transactional data block to be ceased before deallocation. The implementation of the timestamp, used by the versioned write locks, is based in the *global version-clock* algorithm [DSS06]. The idea of the global version-clock algorithm is to have a global variable that is incremented once by each transaction that writes to memory, and is read by all transactions. The clock increment operation must be atomic to prevent data races. Another advantage of this engine is the ease of integration with legacy code, because it requires no modifications to the system libraries used by most applications. For every read operation, the timestamp of the word read is validated against the timestamp of the transaction which is reading it. At commit time the readset is also entirely validated, in order to detect if another transaction wrote a value to the same data item after the read that is being validated.

CTL

CTL, based in the TL2 engine, was developed by Cunha [Cun07]. It implements some new features and also solves some bugs found in the original TL2 engine [LC07]. Table 2.4 summarizes the properties of the modified engine.

This new engine supports object-based granularity, direct-update as a recovery strategy, user explicit aborts and automatic transaction re-execution in case of abortion.

Strong or Weak Atomicity	Weak
Granularity	Word-based/Object-based
Recovery Strategy	Deferred Update/Direct Update
Concurrency Control	Optimistic (Versioned Write Locks)
Synchronization	Blocking
Inconsistent Reads	Validation
Nested Transaction	Partially Closed Nested

Table 2.4: CTL engine properties.

The TL2 engine was implemented in a `SPARC` architecture and CTL was implemented to the `x86` architecture. Another improvement was transaction nesting support, which uses a mixed semantics between closed and flat nested transactions. If a subtransaction aborts explicitly, it rolls back without aborting the parent transaction (as in closed nesting) but if it aborts because of a conflict with another transaction, it rolls back the entire parent transaction (as in flat nesting). The bugs found in the TL2 engine were mainly in validation of memory inconsistencies. The *quiesce* algorithm for recycling transactional memory data blocks was converted, and also the performance of the read operations on transactional data blocks was improved. The automatic transaction re-execution after an abort makes use of an exponential backoff algorithm which forces a transaction to wait for some time before its re-execution. Figure 2.9 illustrates an example of the CTL (and TL2) API's use.

```
1  long *counter = (long *)malloc(sizeof(long));
2  long get() {
3      return counter;
4  }
5  void inc() {
6      Thread *self = TxNewThread();
7      intptr_t addr;
8      long value;
9      TxStart(self);
10         addr = TxLoad(self, &counter);
11         value = (*(long *)addr)+1;
12         TxStore(self,&counter,&value);
13     TxCommit(self);
14 }
```

Figure 2.9: Example of the TL2 API's use.

JVSTM

This STM engine was developed in the context of Cachopo's PhD thesis [Cac07]. This engine is better suited for long-running read-only transactions than most of the STM engines [CRS06]. Table 2.5 summarizes the properties of this engine.

Strong or Weak Atomicity	Weak
Granularity	Object-based
Recovery Strategy	Deferred Update
Concurrency Control	Multiversion Concurrency Control
Synchronization	Non-Blocking (lock-free)
Inconsistent Reads	Do Not Exist
Nested Transaction	Closed Nested

Table 2.5: JVSTM engine properties.

This engine implements a concept of *versioned boxes* which is based in the history objects created by the multiversion concurrency control (see Section 2.2.5). Hence, this engine's concurrency control is optimistic and its synchronization is non-blocking, providing lock-free guarantees. JVSTM also supports closed nested transactions. Versioned boxes can be seen as a replacement for memory locations or transactional variables. During the execution of a transaction, the read and write operations are done in versioned boxes which hold the data values. For each write operation another version is created and tagged with the transaction timestamp. For read operations, the version box returns the version with the highest timestamp less than the current transactions timestamp. A particularity of this engine is that read-only transactions never abort because of concurrency conflicts, as well as write-only transactions. Only read-write transactions can conflict, thus aborting. This STM engine has deferred update as recovery strategy, since every version created by a write operation is stored privately to the transaction and only on commit time it is stored as a public version box. In Figure 2.10 is an example of the JVSTM API's use.

DSTM

DSTM was introduced in Herlihy's paper [HLMWNS03] and was the first dynamic STM engine that did not require a programmer to specify a transaction's memory usage in advance. Table 2.6 summarizes the properties of this engine.

This engine uses non-blocking synchronizing techniques which guarantee obstruction-freedom. DSTM was implemented as a Java programming language library. This engine is object-based and uses deferred update as recovery strategy. The deferred update is implemented using an object cloning technique. Each transaction writes into

```
1 public class Counter {
2     private VBox<Long> count = new VBox<Long>(0L);
3     public long getCount() {
4         return count.get();
5     }
6     public @Atomic void inc() {
7         count.put(getCount() + 1);
8     }
9 }
```

Figure 2.10: Example of the JVSTM API's use.

Strong or Weak Atomicity	Weak
Granularity	Object-based
Recovery Strategy	Deferred Update
Concurrency Control	Optimistic
Synchronization	Non-Blocking
Inconsistent Reads	Validation
Nested Transaction	Flat Nested

Table 2.6: DSTM engine properties.

a private clone of the object and, at commit time, the object is replaced by its clone. A conflict can occur when validating the readset at commit time or if a transactions tries to write on an object which was already written by an uncommitted transaction. This engine also implements flat nested transactions and does not prevent non-transactional accesses. Figure 2.11 shows an example of the DSTM API's use.

More recently Herlihy presented a new version of the DSTM engine which, instead of focusing on the basic model of computation and on run-time techniques as the previous version, intends to provide a safe, convenient and flexible API for application programmers. DSTM2 [HLM06] is a flexible STM engine which permits the use of different synchronization techniques and recovery strategies, as framework *plug-ins*. DSTM2 creates transactional objects using the factory pattern. A programmer can implement his own factory in order to test different properties of STM engines.

```
1  public class Counter {
2      private long counter = 0;
3      void inc() { counter++; }
4      long get() { return counter; }
5  }
6  //...
7  long incrementAndGet() {
8      long value;
9      Counter counter = new Counter;
10     TMObject tmObject = new TMObject(counter);
11     TMThread thread = (TMThread)Thread.currentThread();
12     thread.beginTransaction();
13         Counter c = (Counter)tmObject.open(WRITE);
14         c.inc(); // increment the counter
15         value = c.get(); // get the counter value
16     thread.commitTransaction();
17     return value;
18 }
```

Figure 2.11: Example of the DSTM API's use.

3

CTL Handling System

The CTL handling system aims at allowing the library programmers to specify a set of compensating operations/actions. These compensating actions may revert the effects of some operations that would, otherwise, be irreversible.

3.1 Motivation

Using the transactional memory model in every day programming may seem to be a simple idea, but it may turn out to be a nuisance. When developing a library, the programmer aims at creating a *black box* behind a well defined interface, hiding all the implementation details from the library user. If such libraries are to be used in concurrent environments, programmers must protect them against concurrency hazards, and Software Transactional Memory can be an approach towards such a goal. However, this approach can raise problems as illustrated in Fig. 3.1.

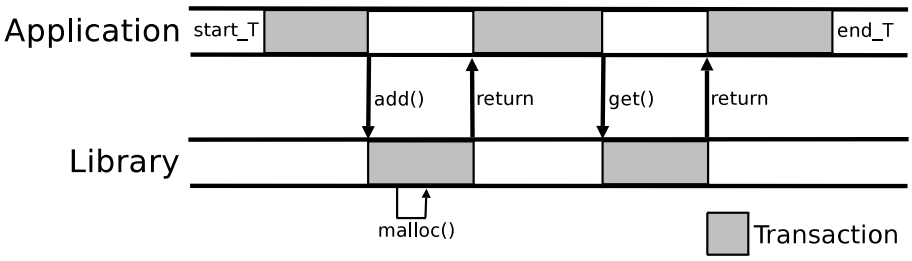

Figure 3.1: Calling a transaction code block in a library.

The fact is that an application is not solely made of memory changes and, typically,

need to perform other operations that are not reversible by the STM libraries, such as write data into a file, read data from a socket, and memory management (allocation/deallocation).

If a library code block is to be executed within a memory transaction whose boundaries (start and end of the transaction) are defined by the library user at the application level, then the library developer has control over neither the start nor the end of the transaction.

Allocating and freeing memory are irrevocable operations and, thus, cannot be executed freely inside a memory transaction. Although these operations are non-transactional, some of them are compensable. This means that the operations can be undone not by reverting their effects but by executing a second operation that will compensate for the effects of the first one. However, not all irrevocable operations are compensable and those that are compensable must obey to an isolation restriction: the effects of an irrevocable, but compensable, operation may only be compensated if no other concurrent transaction depends on (the effects of) the operation to be compensated.

Memory management (allocation and deallocation of memory) falls into the class of irrevocable but compensable operations. Allocating a memory block may be compensated by freeing that memory block. But freeing a memory block cannot be always compensated by allocating another memory block, as the initial memory contents may have been lost. Assuming that this irrevocable operations may be compensated, one must define when to execute the compensating operations.

We propose a solution that is, simultaneously, generic and elegant. Generic because it can be used to solve this and many other problems of the same type, that may arise when using the STM programming model. Elegant because allows the software library to execute compensating operations without the intervention and/or knowledge from the library user, respecting the philosophy of the *black box* model.

3.2 Concept and Model

Our proposal allows the programmer to create inverse functions and to decide when such inverse functions must (and will) be executed. Such a functionality is accomplished by the use of handlers. Handlers will be executed at important moments in the life-time of a transaction, as illustrated in Figure 3.2.

These important moments are:

- *Pre-commit handlers*: These handlers are executed in the context of the transaction to be committed. The memory validation step was done prior to the execution of

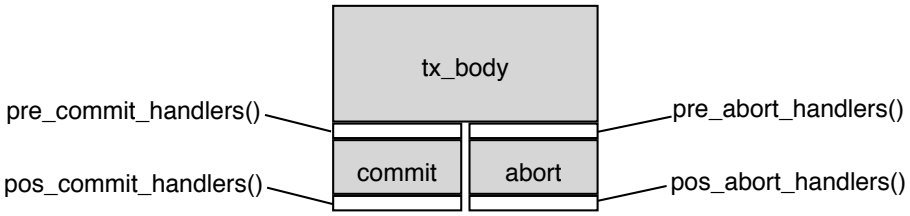

Figure 3.2: Handlers for STM compensation actions.

the pre-commit handlers, thus, they execute knowing that the memory transaction may commit;

- *Post-commit handlers*: These handlers are executed after committing the transaction and, therefore, are outside the transactional context;

- *Pre-abort handlers*: These handlers are executed just before aborting, therefore in the context of the transaction to be aborted. If the STM engine supports automatic retry of a transaction, these handlers are executed just before retrying;

- *Post-abort handlers*: These handlers are executed right after aborting the transaction, therefore outside the scope of a transaction.

All these type of handlers must be registered within the context of a transaction (inside the bounds of a transaction). The life-time of any type of handler is determined by the time of registration until the end of a transaction, either by committing or aborting/retrying. On registering a handler, the programmer may, optionally, pass some data to the handler. This data will be considered later when the handler is executed.

Pre-commit handlers are divided into two categories: *prepare-commit handlers* and *commit handlers*. The former may decide to allow (or not) the memory transaction to commit. The latter are still executed before committing the transaction, but the transaction will irreversibly commit. These handlers are executed sequentially: all *prepare-commit handlers* are executed in first place, followed by all of the *commit handlers*.

These two type of handlers permit to execute a *two-phase-commit* protocol (see Section 2.4.3) between several transactional subsystems in which one of them is the memory transaction.

Figure 3.3 represents the execution of each type/category of handler as a state in a transaction life-time state diagram.

Although pre-commit and pre-abort handlers are executed within the context of the transaction, they cannot make use of transactional memory accesses, as the memory transaction has already been validated and new transactional accesses to memory

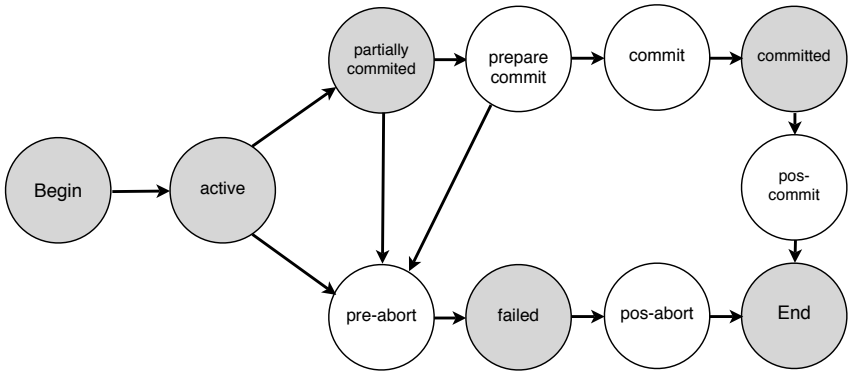

Figure 3.3: Transaction life-time state diagram with handler support.

could require new validations to be carried out. Thus, the programmer has the responsibility of preventing and managing any data-races that may arise when processing/executing handlers.

3.3 Implementation

The model described in the previous section was implemented as an extension to CTL [Cun07], a Software Transactional Memory library for the C programming language.

Each handler is identified as a function pointer that is registered in the handler system. There are two different types of function pointers:

```
1 typedef int (*ctl_prepare_handler_t)(void *);
2 typedef void (*ctl_handler_t)(void *);
```

The type `ctl_prepare_handler_t` declares a pointer to a function to be executed as a *prepare-commit handler*.

This function has one parameter that will point to a user-defined data structure to be passed as an argument to the handler when it is executed, and returns a `boolean` (*true=1* or *false=0*) indicating respectively that the overall transaction can proceed, or that it must abort. If all of the *prepare-commit handlers* return *true*, the *commit handlers* will be executed and the memory transaction will commit, otherwise none of the *commit handlers* will be executed and the transaction will abort.

Each transaction has a list for each type of handlers. Each handler has an integer priority attribute, with small numbers corresponding to a low priority and big numbers corresponding to a higher priority. The priority controls the order in which the handlers are executed. Handlers with a higher priority are executed before handlers

with a lower priority. Within the same priority, handlers are executed by registering order.

The API contains two functions for each type of handler: one requires the programmer to explicitly specify the priority attribute while the other does not, assuming a default priority.

Figure 3.4 shows the registering function for the *prepare-commit handlers*. When registering the handler, it is possible to pass user defined data to be used inside the handler. This data must be passed as a **void** pointer.

```
1 void ctl_register_prepare_handler_priority (
2     ctl_prepare_handler_t handler, void *args, int priority);
3
4 void ctl_register_prepare_handler (
5     ctl_prepare_handler_t handler, void *args);
```

Figure 3.4: Handler System API: *prepare-commit handler* registering functions.

Handlers are eliminated once the associated transactions commit or abort. In CTL, transactions aborting due to a concurrency conflict are automatically restarted. In this case, the handlers are eliminated after executing the last *pre-abort handler* and before the transaction is restarted. Also, *pos-abort handlers* will only be executed for user-aborted transactions, otherwise transactions are restarted automatically and always succeed with a commit state.

3.4 Using the Handlers

We will illustrate how to use the handler system, as described above, to solve the problem introduced in Section 3.1, where a library needs to manage memory inside memory transactions.

When implementing the `add` operation of a linked list, this operation needs to allocate memory for a new list node. If this memory allocation is executed inside a memory transaction and the transaction aborts, it is automatically restarted and, a new list node will be allocated. The previous list node will be dangling and will originate a memory leak in the program. In this case the library developer could register a *pre-abort handler* to free the just allocated memory in case of the abort/restart of the transaction.

Figure 3.5 illustrates the use of a *pre-abort handler* to compensate for the operation of memory allocation, when the transaction aborts while adding a new node to a linked list.

```
1  void freevar (void *args) {
2      free (args);
3  }
4  void add (List *list, void *item) {
5      Node *node;
6      node = malloc (sizeof (*node));
7      ctl_register_pre_abort_handler (freevar, (void *)node);
8      node->next = NULL;
9      node->value = item;
10     TxStore (&(list->tail->next), node);
11     TxStore (&(list->tail), node);
12 }
```

Figure 3.5: Linked list `add` operation with handler system support.

The inverse problem of compensating (postponing) for a memory deallocation problem is also easy to solve with a *pos-commit handler*. As an example, we will consider the removing of the head node of the linked list, as illustrated in Fig. 3.6.

```
1  void *removehead (List *list) {
2      Node *node;
3      void *value;
4      node = (Node *)TxLoad (&(list->head);
5      TxStore (&(list->head), node->next);
6      value = (void *)TxLoad (&(node->value));
7      free (node);
8      return value;
9  }
```

Figure 3.6: Linked list `removehead` operation.

If this `removehead()` function is called inside a memory transaction, and the transaction aborts after the `free()` operation, the transactions will be restarted and the function will be called once again, but now the head pointer `list->head` is pointing to an invalid memory block, because it was already released in the call to `free` in the previous execution. To solve this problem, one must delay the memory deallocation until the transaction commits. This can be achieved by registering a *pos-commit handler* to free the respective node as depicted in Fig. 3.7.

This solution could be supported at either programming language/compiler or library level. The library based solution was just described. A compiler based solution would have the compiler to transparently generate all the necessary code for registering the handlers and calling the replacement front-ends instead of the original functions.

```
1  void freevar (void *args) {
2    free(args);
3  }
4  void *removehead(List *list) {
5      Node *node;
6      void *value;
7      node = (Node *)TxLoad (&(list->head);
8      TxStore (&(list->head), node->next);
9      value = (void *)TxLoad (&(node->value));
10     ctl_register_pos_commit_handler (freevar, node);
11     return value;
12 }
```

Figure 3.7: Linked list `removehead` operation with handler system support.

In terms of general library development, each library can register the appropriate handlers to delay or reverse its effects to the commit/abort time, without the library user being aware of such handlers.

3.5 Performance Evaluation

We performed a set of simple tests to evaluate the overhead introduced by the handler system into the CTL engine. The tests consist on series of operations on a set. The set is a Red Black Tree implementation. The implementation provides three functions — put, delete and get. The set elements have a key and a value and all functions are indexed by the key. Duplicate keys are not allowed and adding an element with an already existing key just updates its value. Only the put and delete operations make use of the handler system. In the put method, a *pre-abort* handler is registered to compensate for the creation of a new node inside of a transaction, and in the delete method, a *pos-commit* handler is registered to defer the free of a node memory until the commit of the transaction.

The tests are divided in two types characterized by two different load patterns. The test load pattern is composed by different probabilities for each of the three methods provided by the set implementation. The first load pattern, that is meant to simulate a read dominant context, is composed by 5% of puts, 5% of deletes and 90% of gets. The second load pattern, is meant to simulate a write dominant context, and is composed by 45% of puts, 45% of deletes, and 10% of gets.

The tests were performed on a Sun Fire X4600 M2 x64 server with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz with 1024 KB cache, and the obtained results are depicted in Figures 3.8 and 3.9. We ran tests with as much as 64 threads

competing for the 16 available processors.

The graphics show three different executions: one with no handlers support, another with the handler support activated but with empty handlers, and another with handler support activated and executing the appropriate code inside the handlers (in this case executing the free operation).

As depicted in Figure 3.8, all the three different executions are at the same level of performance, meaning that if we do not use the handlers (because most operations are read-only), there is no performance penalty from the handler system.

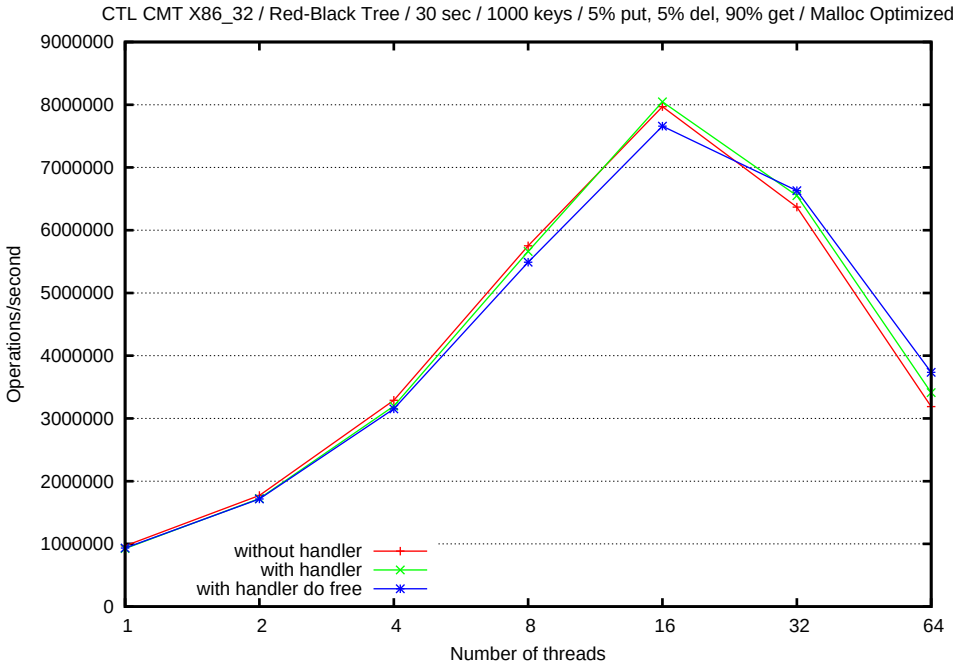

Figure 3.8: Overhead introduced by the handler support in read-dominated environment.

Figure 3.9 shows that the performance penalty inherent to the registration and execution of the handlers is not high besides that, using the handler system, the code is freeing the memory when necessary.

3.6 Related Work

Tim Harris in [Har05], also uses callback handlers in the form of *external actions* to provide support for operations with side-effects, such as console I/O, in the Java programming language. This work was derived from an earlier approach by the same author in [Har03]. These external actions are implemented using a copy of the heap in the moment of the invocation of an I/O operation inside a transaction. This invocation is delayed until the end of the transaction, and then executes the I/O operation in the

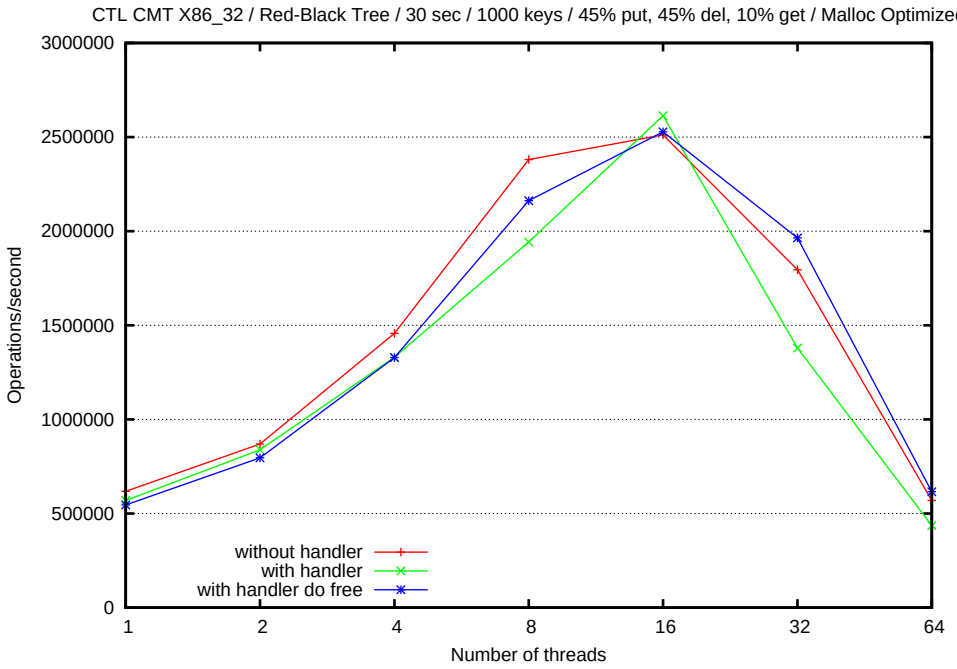

Figure 3.9: Overhead introduced by the handler support in write-dominated environment.

same context (using the heap copy) in which the invocation was made. A drawback of this approach is that, if used to implement libraries as described in this paper, the library would have to have control over start and end of the transaction.

Other related work is DSTM2 [HLM06], which provides the ability to register methods (in the form of `Callable<Boolean>` objects) to be called when important transactional events occur. The authors simply state that it is possible to register methods to be executed after a transaction commits or aborts.

4

Integration of Database and Memory Transactions

The main objective of this dissertation is to study the relationships between memory transactions and database transactions and, if possible, make them cooperate. In this chapter we will present our unified model between database and memory transactions and the respective implementation as well.

4.1 Unified Transactional Model

The integration between two transactional models could, conceptually, be quite easy if both satisfy the same semantics and the same properties. In this case, where we have memory and database transactions, memory transactions satisfy only a subset of the properties satisfied by database transactions. Memory transactions do not satisfy the *Durability* property and satisfy *Consistency* property only partially. Each transactional system could have different concurrency control implementations, or different recovery mechanisms, and therefore, their integration could be not so straight forward.

In the transactional model, each transaction is defined by its bounds. These bounds are defined, generally, by syntactical constructs. When using two transactional models, simultaneously, there are two types of transactions and, hence, the bounds for each type of transaction must be defined separately. Let `TX1_BEGIN` and `TX1_END` be the bounds delimiter for one transactional model, and let `TX2_BEGIN` and `TX2_END` be the bounds delimiter for the other transactional model. These four operations could be

interleaved in four different ways, as depicted in Figure 4.1.

```
TX1_BEGIN();
    //...
    TX2_BEGIN();
    //...
    TX2_END();
    //...
TX1_END();
```

```
TX2_BEGIN();
//...
    TX1_BEGIN();
    //...
    TX1_END();
//...
TX2_END();
```

```
    TX1_BEGIN();
    //...
TX2_BEGIN();
    //...
    TX1_END();
//...
TX2_END();
```

```
    TX2_BEGIN();
    //...
TX1_BEGIN();
    //...
    TX2_END();
    //...
TX1_END();
```

Figure 4.1: Example of two distinct transactions interleavings.

The semantics for each of the interleavings presented in Figure 4.1 is not intuitive. We could consider a nested transaction semantics for the interleavings where one transaction is embedded within the other transaction, but for the other two, it is not clear what is the behavior if one of the transactions commits or aborts.

In the unified model there will be only one transaction bounds delimiters. Considering the previously examples, the unified model will only consider the outer transaction delimiters and ignores the inner delimiters. The transaction will satisfy all the properties for both models. In this concrete case, where transactional memory will be unified with database transactions, the former will satisfy the ACI properties, and the latter will satisfy the ACID properties. Figure 4.2 presents an example of the use of the unified model. Be the `TX_BEGIN` and the `TX_END` the overall transaction bounds delimiters, any operation within this bounds will be committed only at the time of the `TX_END` execution. If the transaction aborts for any operations, independently of the transactional model to which they belong to, the overall transaction aborts.

```
TX_BEGIN();
    x = select x from table_1 where id=4;
    y = 3;
    insert into table_2 values (x, y);
TX_END();
```

Figure 4.2: Example of the unified model use.

The semantics of our unified model is very simple, actually it is like any other trans-

actional model. In the user perspective it is like that all the operations, in database or in memory, are at the same level, and the unified transactional system will take care of all the work.

The commit phase in this unified model will have to commit all the underlying transactions, belonging to the different transactional engines, in an atomic way. Based on the theory of distributed transactions, using the *two-phase-commit* protocol [Gra78], it is possible to commit N transactions, from N different transactional engines, in an atomic way.

In the *two-phase-commit* protocol, all transactions must first agree on whether they will commit or abort. This phase is called the *preparation phase*. After this phase all the transactions must perform the decision agreed in the previous phase. If the decision was to commit then all the transactions commit, otherwise, if the decision was to abort, then all the transactions abort. This phase is the *commit phase*.

In our unified model, we can unify several transactions into a single one and preform a *two-phase-commit* in order to commit all of them.

4.2 Implementation

The unified model described in the previous section will be implemented using the CTL handler system described in Chapter 3. The problem of dealing with two different transactional models at the same time is that the commit phase of both transactional models must be atomic: or both transactional models commit or both abort.

The handler system, provided by CTL, gives the possibility of making a *two-phase-commit* with respect to the memory transaction, by registering *prepare-commit handlers* and *commit handlers* associated with the respective memory transaction.

Since the memory transaction is already part of the *two-phase-commit* protocol, only the database transaction needs to be added to the protocol. The database operations are done using an ODBC interface for the C language. This allows to use any ODBC compliant database management system.

In order to add the database transaction to the *two-phase-commit* executed by CTL handler system, three handlers must be created: one for the prepare phase, other for the commit phase and another for the abort phase. But, not all the DBMS support the prepare phase, and the ODBC interface does not support as well. So, with the ODBC interface, it is only possible to commit the database transaction and it is not possible to reverse it once done. With this limitation, our implementation supports only one DBMS per memory transaction. Without the prepare phase, the commit of the database transaction will be done in the *prepare-commit handler* and, if the commit

is successful, the result of the *prepare-commit handler* will also be successful and the memory transaction will also commit. If the database commit is not successful, hence the database transaction has aborted, then the *prepare-commit handler* will also fail and the memory transaction will be rolled back executing the *pre-abort handler*. In case of any operation fails, during the execution of a transaction, the overall transaction will abort and the *pre-abort handler* will abort the database transaction.

To ease the work of registering all the necessary handlers to allow the usage of a database transaction inside a memory transaction, we created a front-end to the CTL `TxStart` function, in which all the necessary handlers are registered.

Figure 4.3 shows the definition of this front-end. The first and third parameters are the same as for the `TxStart` function. The second parameter is the database connection handler for the ODBC interface. First it will start the memory transaction calling the `TxStart` function, then it will register both the `do_commit` function as the *prepare-commit handler*, and the `do_abort` function as the *pre-abort handler*. The definition of the `do_commit` and the `do_abort` are similar, both only call the ODBC `SQLEndTran` function with the appropriate flag indicating whether to commit or abort. Figure 4.4 shows the `do_commit` function. The database transaction is implicitly started by ODBC upon the first operation over the database, and it will end upon the commit or abort of the transaction.

```
1 void TxDBStart(Thread *Self, HDBC dbc, int roflag) {
2     TxStart(Self,roflag);
3     _ctl_register_prepare_handler(Self, do_commit, dbc);
4     _ctl_register_pre_abort_handler(Self, do_abort, dbc);
5 }
```

Figure 4.3: `TxStart` front-end for using database transactions.

```
1 int do_commit(Thread *Self, void *args) {
2     SQLRETURN ret;
3     ret = SQLEndTran(SQL_HANDLE_DBC, (HDBC)args, SQL_COMMIT);
4     if (!SQL_SUCCEEDED(ret)) {
5         return 0;
6     }
7     return 1;
8 }
```

Figure 4.4: `do_commit` function handler definition.

The database connection handler passed to the `TxDBStart` function determines which database is going to be used in the current memory transaction. It is possible

in another memory transaction to use a different database, but not more than one per memory transaction. The user can still use the `TxStart` function if he is not going to operate with the database.

4.3 Using the Model

This unified model is very easy to use, even for programmers with little experience in using transactional memory engines, and database programming using ODBC interfaces. In practice, a programmer will create memory transactions, and inside it he can use the ODBC API to operate over the database.

Figure 4.5 shows the definition of a transaction which inserts an integer into a memory integer array and into a database table. The `runQuery` function is wrapping the ODBC API to make database queries. If this insert function is executed by several threads in parallel it will detect conflicts both at memory and database levels, and it is guaranteed that it will change memory and database atomically.

```
1  int array[MAX] = {0};
2  int curr = 0;
```

```
1  void insert(Thread *Self, HDBC dbc, int num) {
2      TxDBStart(Self,dbc,0);
3      int *pos = TxLoad(Self, &array[curr]);
4      TxStore(Self, *pos, num);
5      char buf[128];
6      sprintf(buf, "insert into int_table values
7              (%d);", num);
8      runQuery(dbc, buf);
9      TxCommit(Self);
10     curr++;
11 }
```

Figure 4.5: Example of inserting an integer into a memory array and into a database table.

4.4 Unified Model Problems

Some problems were found while testing the implementation of the work described in previous sections, using CTL engine in *deferred update* mode with a PostgreSQL database management system.

The test application was very simple. It implemented a memory cache of integer

numbers that were present in a database table. The test application has three operations:

- *Insert*: Inserts a number in the cache and inserts the same number in the database table. If the number already exists in database it does not do anything more.
- *Delete*: Deletes a specified number from the database table and from the memory cache.
- *Get*: Finds a specified number from the cache, if it does not find it, tries to find it in the database table. If it is in the database, then adds the number to the cache.

Each operation is defined as a transaction. Every time the memory cache is full, a random cache position is chosen to be freed. The delete operation in the memory cache reads all the elements in the cache until it finds what is looking for, if it does not find it, it does not do anything more.

The three operations were called randomly by more than one thread at a time. The definition of each operation is depicted in Figure 4.6. The operations that access the memory cache and the database are self described by their name.

Several tests were made to validate the coherence between the data present in memory and the data present in database. The assertion test was: every data that is in memory must be also in the database. An assertion failure meant that a coherency problem was found. To understand what caused the assertion failure, operations' traces were captured and analyzed to find out which operation interleavings were made.

4.4.1 Serialization Incompatibilities

Frequently the assertion defined in the previous section failed. We captured traces with the interleavings of the operations, to find out which sequence of operations was breaking the assertion. Figure 4.7 shows the interleaving of two operations executed by two threads. Thread **A** executes a `delete` operation for the number 10, and thread **B** executes a `get` operation also for the number 10. Dashed lines indicate thread context switching.

The problem depicted in the trace in Figure 4.7 is that thread **B** commits a transaction that stores in memory a value read from the database which was removed meanwhile by a committed transaction executed by thread **A**. In our perspective, the result of such interleaving must have been either thread **B** aborts the transaction when executes `TxCommit`, or the database system blocks the request or aborts the database transaction when thread **B** executes `select_from_db`.

```
1  void insert(Thread *Self, HDBC dbc, CACHE ch, int n) {
2      TxDBStart(Self, dbc, 0);
3      if (insert_in_db(dbc, n)) {
4          // successful inserted in db
5          insert_in_cache(ch, n);
6      }
7      TxCommit();
8  }
```

```
1  void delete(Thread *Self, HDBC dbc, CACHE ch, int n) {
2      TxDBStart(Self, dbc, 0);
3      if (delete_from_db(dbc, n)) {
4          // successful deleted from db
5          delete_from_cache(ch, n);
6      }
7      TxCommit();
8  }
```

```
1  int get(Thread *Self, HDBC dbc, CACHE ch, int n) {
2      int *res;
3      TxDBStart(Self, dbc, 0);
4      if ((res = get_from_cache(ch, n)) == NULL) {
5          // does not exists in cache
6          if (select_from_db(dbc, n)) {
7              // exists in database
8              insert_in_cache(ch, n);
9              *res = n;
10         }
11     }
12     TxCommit();
13     return *res;
14 }
```

Figure 4.6: Definition of test application operations.

Thread A `delete(10)`	**Thread B** `get(10)`
TxDBStart delete_from_db delete_from_cache *(read-only)*	
	TxDBStart get_from_cache select_from_db insert_in_cache
TxCommit	
	TxCommit

Figure 4.7: Trace of the execution of two operations by two threads.

The reason for the trace shown to happen, is that the memory transaction and the database transaction are running in different isolation levels. PostgreSQL implements *snapshot isolation* [BBG+95], in contrast to CTL implementation of *total serialization*. In the `select_from_db` operation, PostgreSQL is retrieving the last committed value present when the respective transaction started, and at time of commit it does not need to verify this read, because in terms of database transaction, it is like the database transaction of thread **B** occurred before the database transaction of thread **A**. As a result of the unification of the two transactional models, this type of serialization is not admissible, because the memory transaction of thread **B** is serialized as if it occurred after the memory transaction of thread **A**. Figure 4.8 shows the result serialization schedule for both memory transactions and database transactions in this particular case.

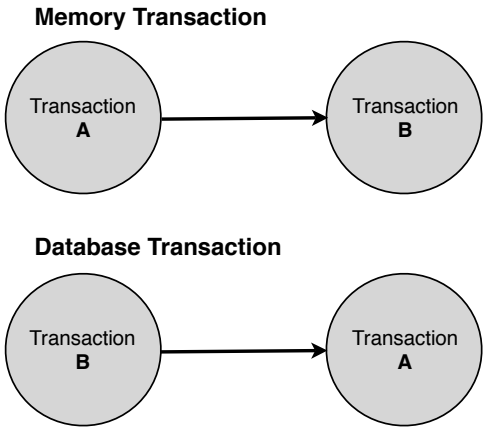

Figure 4.8: Serialization schedule of memory and database transactions.

In order to be able to unify the two models, the serialization schedules generated dynamically by both models must be the same, and this applies always to the use of more than one database engine in the same transaction.

4.4.2 Solution Approach

This problem only occurs when `SELECT` (read-only) statements are used inside transactions, and the respective DBMS does not support *total serialization* isolation level. One way of solving this problem is using a DBMS that implements *total serialization* isolation level. We have executed the same test with a DB2 database management system and no problems were found.

Another possible solution to the problem is to fully serialize *snapshot isolation* schedules. Since the problem is that *snapshot isolation* does not detect read-write conflicts, we must detect ourselves. There are two possibilities of treating the conflict: detect and

solve the conflict at the moment the read operation is done or log all transactions operations and detect and solve the conflict at commit time.

In the first approach, we will force the read operations to be treated as write operations by the DBMS [FLO+05]. In this way, read-write conflicts disappear because all operations turn into write operations. A simple way to do it, is for each `SELECT` statement in our program of a data item D_i, to update the value of the data item $V(D_i)$ to itself $V(D_i) = V(D_i)$. This update needs to be done before the `SELECT` statement, to acquire a write lock before reading the value. This solution will transform all read operations into write operation and therefore a performance overhead will exist. The PostgreSQL database [Pos] implements a SQL statement to address such problem, the statement `SELECT FOR UPDATE` or `SELECT FOR SHARE` can be used to force `SELECT` statements to be treated as updates. The difference between the two statements is that `FOR UPDATE` acquires an exclusive lock of the rows being accessed by the `SELECT` statement, and the `FOR SHARE` acquires a shared lock, thus permitting other `SELECT FOR SHARE` statements to not block. If any row is locked by an exclusive or shared lock, any write attempt to the locked row will block.

To implement the second approach, all read and write operations made by the database transactions must be registered in a log. When a database transaction tries to commit, its operations must be validated and if a read-write conflict is found the transaction must abort.

Each read operation will be identified as $R(T_i, x)$, meaning transaction T_i read data item x, and each write operation will be identified as $W(T_i, x)$ with similar meaning. Each transaction has also a $S(T_i)$ identifying the time that transaction T_i started and $E(T_i)$ identifying the time that transaction T_i ended (committed). To find if there was a conflict when committing a transaction T_i then, for each $R(T_i, x)$ in the readset of transaction T_i, there is a conflict if there exists a $W(T_j, x)$ in the writeset for some T_j where $S(T_i) < E(T_j) < E(T_i), i \neq j$. This means that if there was a write operation in data item x made by a transaction T_j that committed between the start and end of the transaction T_i, then there was a read-write conflict, and transaction T_i must abort. Figure 4.9 shows the pseudo-code of the validation algorithm.

The use of this approach for all the transactions has a strong impact in the performance. Another problem with this solution is that most DBMS do not provide the information necessary to apply the validation algorithm, such as transactions readsets and writesets. In this case, such information must be collected *manually* with even higher performance impact.

Applying the first solution described above to the application that generated the trace presented in the previous section, should result in a trace like the one depicted

```
1 foreach R(Ti, x)
2     foreach Tj %j!=i
3         if E(Tj) > S(Ti) and E(Tj) < E(Ti)
4             if exists W(Tj, x)
5                 ABORT TRANSACTION
6             endif
7         endif
8     endfor
9 endfor
```

Figure 4.9: Validation algorithm pseudo-code

in Figure 4.10. If we apply the second solution, then the trace should be like the one depicted in Figure 4.11.

Thread A `delete(10)`	**Thread B** `get(10)`
`TxDBStart`	
`delete_from_db`	
`delete_from_cache` *(read-only)*	
	`TxDBStart`
	`get_from_cache`
	`select_from_db` *(blocks)*
`TxCommit`	
	`TxCommit`

Figure 4.10: Trace of the execution using `SELECT` as a write.

Thread A `delete(10)`	**Thread B** `get(10)`
`TxDBStart`	
`delete_from_db`	
`delete_from_cache` *(read-only)*	
	`TxDBStart`
	`get_from_cache`
	`select_from_db`
	`insert_in_cache`
`TxCommit`	
	`TxAbort` *(and tries again)*

Figure 4.11: Trace of the execution using commit validation algorithm.

5

Use Case Example

In this chapter we will describe an application developed to validate the use of the unified model described in Chapter 4. This application makes use of both database and memory data structures to create a retrieval system.

5.1 Use Case Description

The application developed as a use case can store and retrieve scientific articles from a database. The articles can be indexed by author name, by keywords, or both. The article repository has an interface with four functions: insert an article, remove an article, find by author, and find by keyword. Each function accesses different shared data structures, as well as database tables in order to maintain consistency between memory and database.

The application is divided in two components: a server, and a client. The server will manage the repository allowing concurrent calls to the repository interface. The client is a single threaded component and its only purpose is to make calls to the server. The idea of using the unified model in this application is to have in memory the articles indexation structure which is also in database. If the application is closed then all information persists in database, when the application runs again then all indexation information is loaded into memory. Using our unified model is very simple to maintain the consistency between the database and its partial replica in memory with higher levels of performance.

First are described the database entity-relation model and the memory data struc-

tures used, and then each function of the repository interface will be described in detail.

5.1.1 Database model

The database scheme is very simple, it has three entities: articles, authors, and keywords. An article is represented by an internal *id*, a title, and the file path to the article document. An author is represented by its name (we assume that each author name is unique). A keyword is represented by the respective keyword.

Each article can have a number, higher than one, of authors and keywords. And there cannot exist an article in the database without authors and keywords. In Figure 5.1 presents the entity-relation model of the database scheme just described.

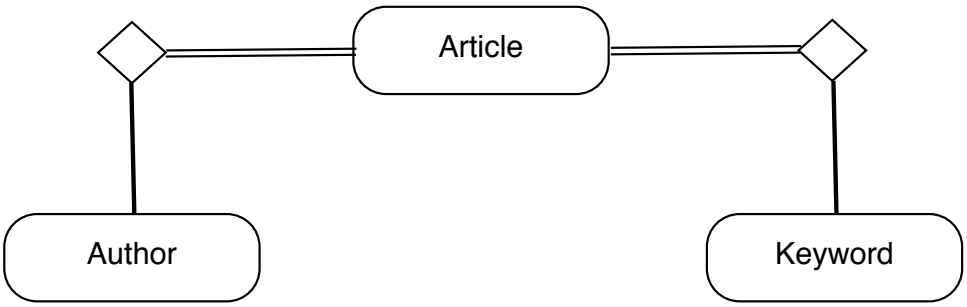

Figure 5.1: Application database entity-relation model.

The relation between authors and articles is supported by an association table where each row makes the link between an author and an article. The rows stored in this association table are the articles indexing structure and will be also replicated in memory. The same applies to the relation between keywords and articles.

5.1.2 Memory Data Structures

The indexing structure in memory is represented by a singly linked list and a hash table. The structure is simple: one hash table for indexing authors and another for indexing keywords, and each element of the hash table is a list of article identifiers which are associated with the author or the keyword.

The singly linked list was implemented using CTL to protect it from concurrency problems. This list implements three operations: add an element, remove an element, and get an element from the list. Each operation uses the handler system, described in Chapter 3, to manage list nodes memory allocation and deallocation. The `add` operation, adds an element to the head of the list. The `remove` operation removes the *nth* element from the list. The `get` operation, gets the *nth* element from the list.

The hash table is separate chaining and was also implemented using CTL to protect it from concurrency problems. It implements three operations: add an element, remove an element, and get an element from the hash table. Similar to the linked list, each hash table operation also uses the handler system support.

5.1.3 Description of Repository Operations

Inserting an article in the repository is an operation with several steps. First the article is inserted in the database. If the database insertion is succeeded, because the article is not yet in the database, then for each author of the article the association between the author and the article is inserted in the database and also in the list of articles stored in an element of the hash table indexed by the author's name.

The remove operation is implemented as follows: for each author and keyword, of the specified article, are removed all the respective associations between authors and article, and keywords and article from the memory hash tables and from the database.

Finding an article in the repository, by author or by keyword is implemented as follows: get the article list from the respective hash table, and for each article in the list retrieve it from database.

The operations of insert and remove from the database repository only use `INSERT` and `DELETE` SQL statements, and do not use any `SELECT` statement. In this particular application does not arise the problem identified in Section 4.4 and therefore, we can use any database implementing *snapshot isolation* to test this application.

5.2 Performance and Comparison Tests

To perform comparison tests, the same application was implemented using locks. Each repository operation was protected with a global lock. This approach was not the best option for performance but aimed at keeping the simplicity as it would be if using STM. We also implemented another approach, this one just using the database without memory data structures. In this approach all operations access only the database. The find operation which previously was implemented using only memory accesses, in this approach, is implemented querying the database with `SELECT` statements.

The tests preformed were analyzed in six different environments: a read dominant context, a write dominant context, and a read-write dominant context; all in both low and a high contention context. The operations of inserting and removing from the repository are *read-write* operations, and finding an article by author or by keyword are *read-only* operations. The tests are characterized by three types of operations: insertions, removals, and gets. Each operation has a predefined probability defined as

an application parameter. The number of articles to be inserted in the repository will control the contention level and will be also defined as an application parameter.

The tests were performed on a Sun Fire X4600 M2 x64 server with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz with 1024 KB cache. The database management system used was a PostgreSQL 8.3.

Figures 5.2, 5.3 and 5.4 show the results obtained for the different configurations.

The first remark is that in all tests the CTL implementation surpassed the lock implementation. This was the expected result as database access times are much higher than memory access times and, in the lock implementation, a thread executing an operation acquires an exclusive lock and since locks are held a long time, no operations can be executed concurrently. With CTL, if there are no conflicts, all operations execute concurrently, and no thread has to wait for others.

The contention level is also a determinant factor in the performance of the CTL implementation. With high contention, transactions tend to conflict more with each other, having a lower performance than in a low contention environments.

The load pattern also influences the three implementations. In a read dominant context the performance of all three implementations is higher, and lowers as the percentage of write operations increases.

Comparing the CTL implementation with a more realistic implementation using only the database to access data, we can see that CTL implementation is better in all low contention environments and it is not significantly worse in high contention environments. The technique of keeping some information in memory to boost performance of read operations is relatively simple to implement using our unified model and the performance gains over the database only approach are evident.

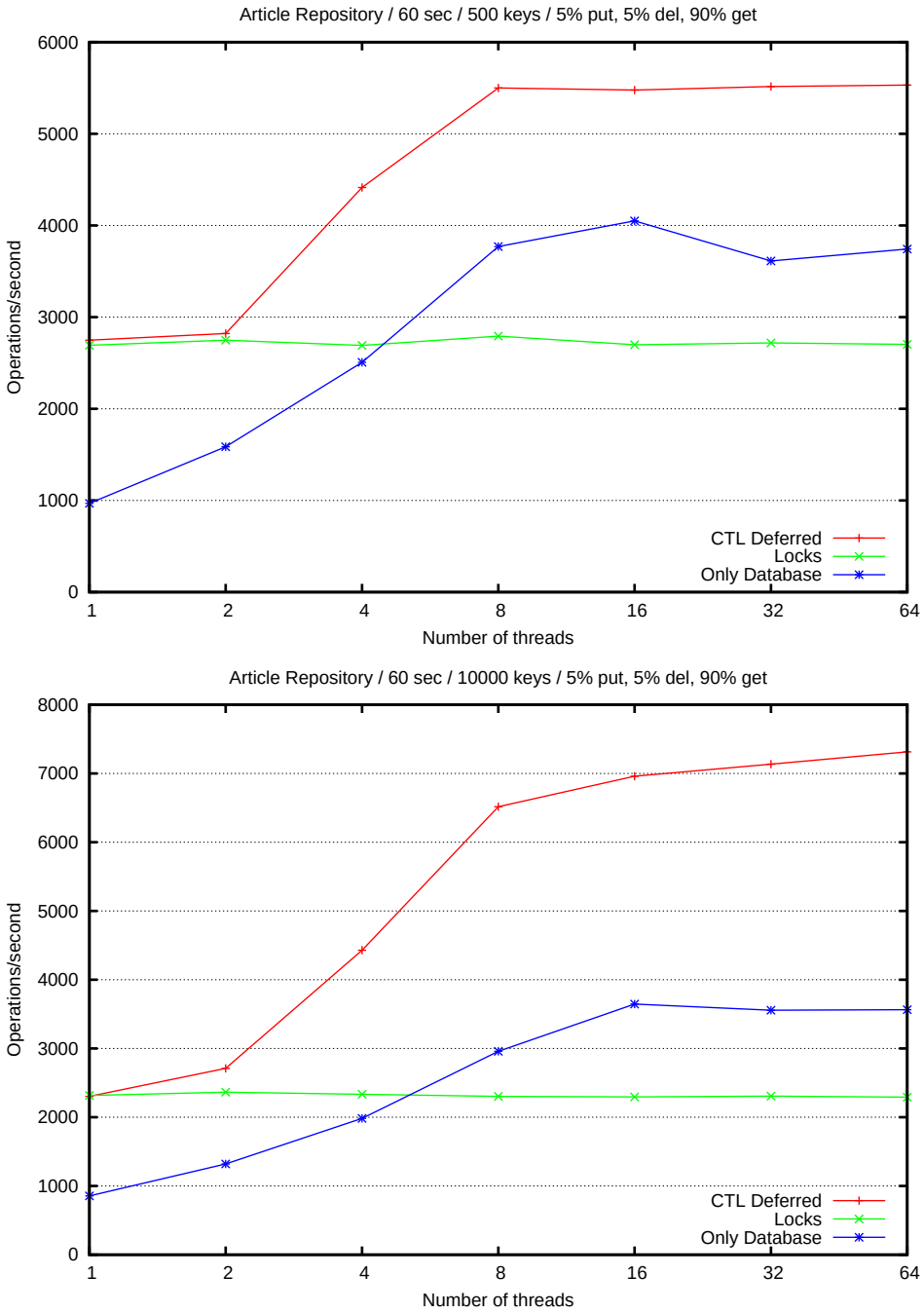

Figure 5.2: Repository 5% inserts, 5% deletes, 90% lookups with high contention (top) and low contention (bottom)

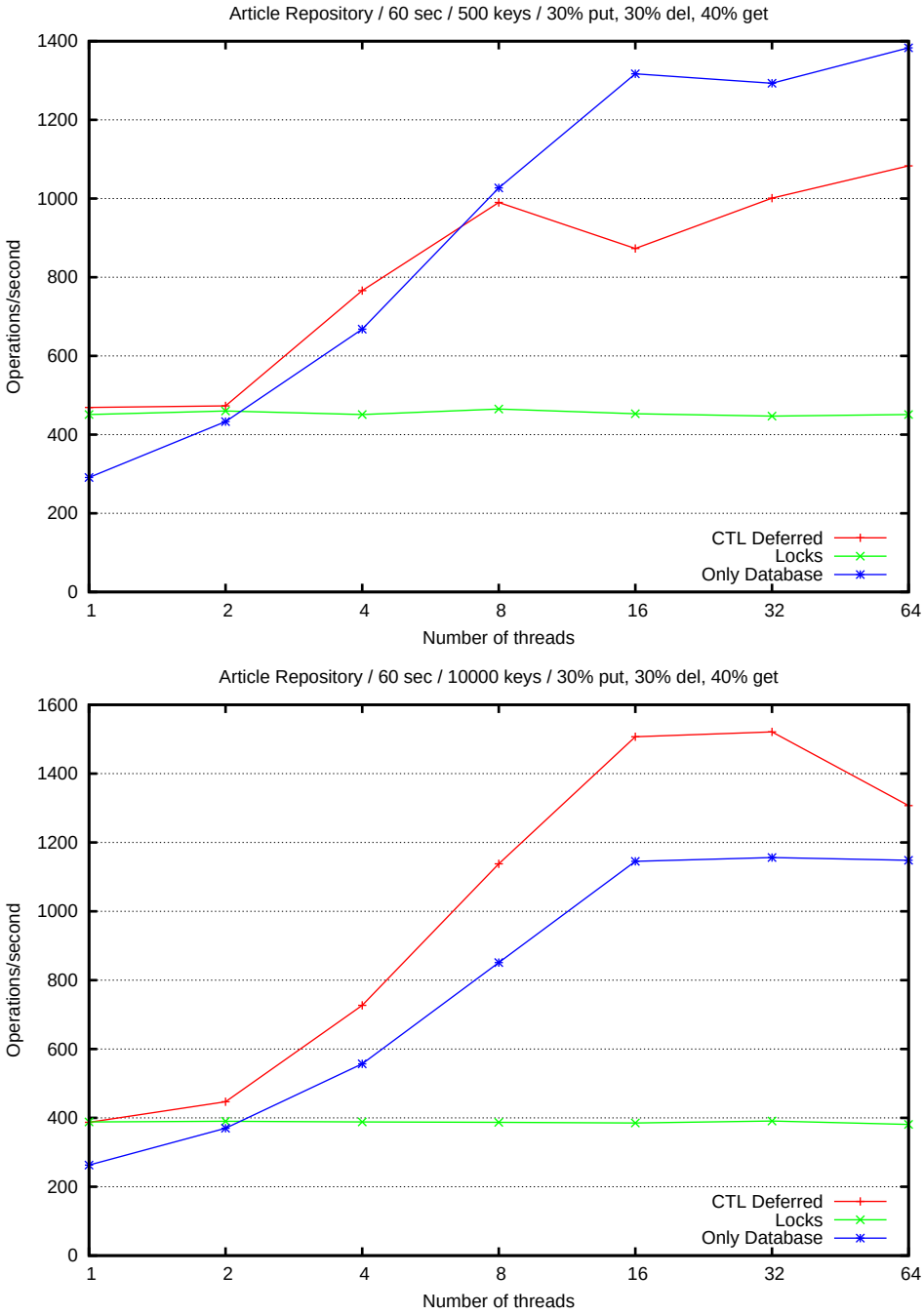

Figure 5.3: Repository 30% inserts, 30% deletes, 40% lookups with high contention (top) and low contention (bottom)

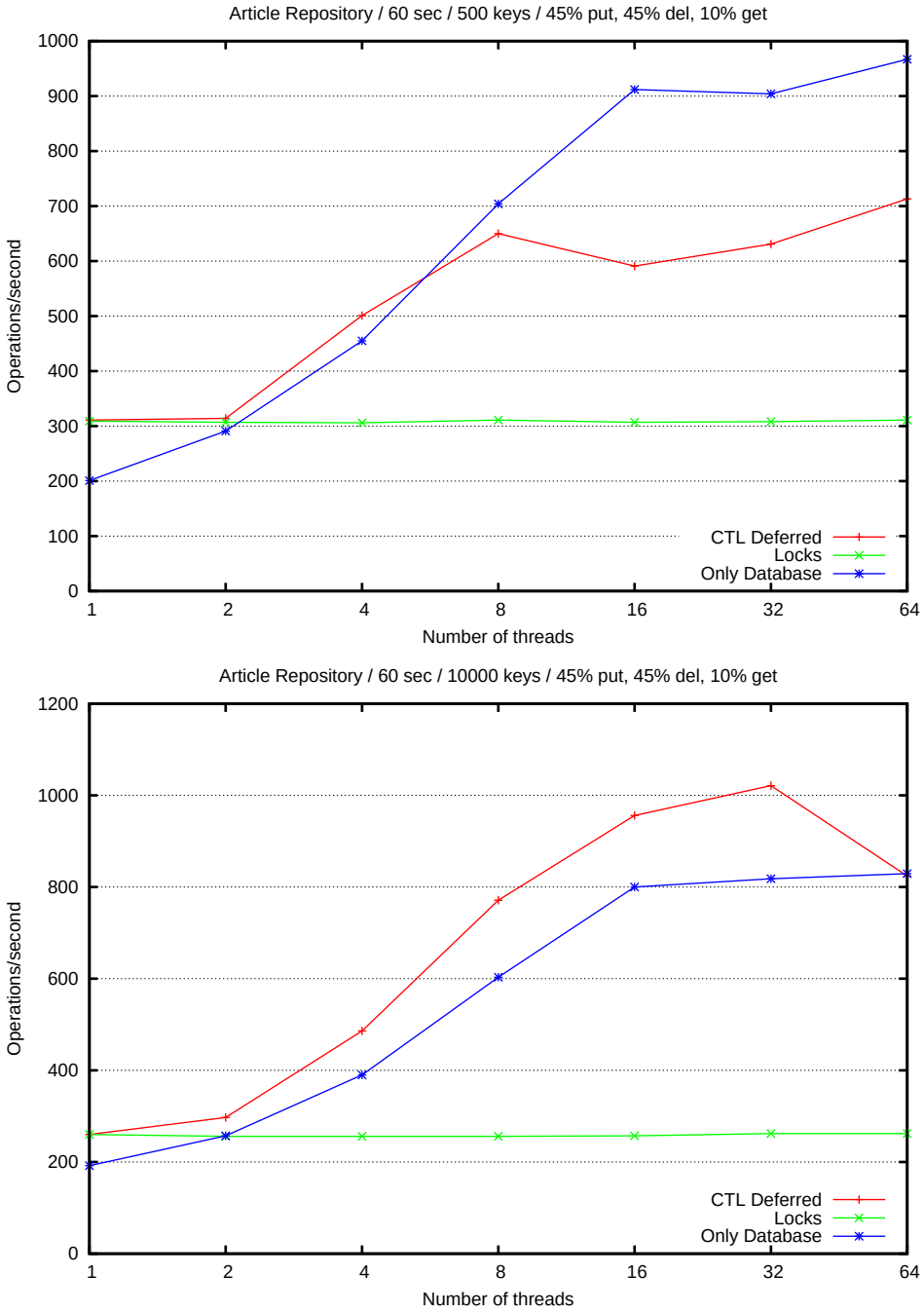

Figure 5.4: Repository 45% inserts, 45% deletes, 10% lookups with high contention (top) and low contention (bottom)

6

Conclusions

This work has presented a solution to the problem of integrating database transactions with memory transactions. The path to achieve such a solution led to the creation of an handler system which allows a user to create compensating actions to revert the effects of apparently irreversible operations within the context of a memory transaction.

The handler-based technique presented in this thesis is a generic and elegant approach to solve the problem of executing irrevocable (but compensable) operations in the context of a software memory transaction, at a negligible cost. It can also be used to easily revert irrevocable (but, again, compensable) operations inside a library that will be executed within a memory transaction. This technique is not tied to any specific problem neither to any specific solution. It is also independent of the specific model or implementation of the STM framework. The proposed technique only depends on the programmer to correctly use the handlers and create the operationally effective solution.

This handler system was the basis of the integration of database and memory transactions. Using the *two-phase-commit* protocol to commit both, memory and database transactions, CTL becomes the first STM engine to implement this integration. With the front-end for the CTL transaction start function, users are able to access the database data very easily from within a memory transaction. Besides ease of use of this technique, also the performance gains, relatively to a lock based approach, are an advantage.

Other contribution of this work was the identification and its resolution of the problem raised by the differences between database and memory transaction isolation lev-

els. Most STM engines provide serializable schedules and, most DBMS provide more relaxed approaches. To integrate database and memory transactions, both engines must generate equal schedules, otherwise consistency errors may occur.

6.1 Future Work

The implemented unified model was the simplest that could be done, but it is a starting point for the study of more advanced features. Support for more than one database transaction simultaneously is a feature that was limited only by the technology used to communicate with the database. Using another API that supports the distributed transaction XA protocol, it should be simple to implement such a support.

Other interesting feature is the support for nesting transactions (closed nesting) in our unified model. As database accesses are very time expensive, using nested transaction would avoid repeating database accesses.

Adapting our unified model to support speculative database accesses, is also an interesting feature with some usefulness. Being able to access speculatively the database and, at the time of receiving the real value, the transactional engine would decide automatically if the current transaction continues or abort its execution.

The handler system could also be improved by letting a user to use transactional API inside the handlers that execute inside the transaction. This way a user could access shared memory positions, increasing in this way the functionality and usability of the handler system.

Raw Test Data

A.1 Handler System Overhead Tests

A.1.1 Load Pattern: 5% put, 5% del, 90% get

Threads	Total Operations	Operations/second	Total aborts
1	29088933	969385	0
2	53181533	1772233	2426
4	98722308	3289888	13512
8	172572663	5751041	52215
16	239139152	7969286	121222
32	191076724	6368072	105782
64	95643569	3187700	74302

Table A.1: Red-Black Tree, CTL Deferred Update, 30 sec, 1000 keys, 5% put, 5% del, 90% get. Without Handlers

Threads	Total Operations	Operations/second	Total aborts
1	27751290	924772	0
2	51707668	1723106	45345
4	95922885	3196513	257845
8	169897360	5661735	1019467
16	241453922	8046527	3419825
32	196551385	6550618	3870733
64	102384266	3412436	4408400

Table A.2: Red-Black Tree, CTL Deferred Update, 30 sec, 1000 keys, 5% put, 5% del, 90% get. With Handlers No Free

Threads	Total Operations	Operations/second	Total aborts
1	28043822	934538	0
2	51531958	1717240	52320
4	94596643	3152318	286527
8	164760035	5490610	1220105
16	229804999	7658377	3629943
32	198939790	6630223	3927133
64	112093040	3735751	4353330

Table A.3: Red-Black Tree, CTL Deferred Update, 30 sec, 1000 keys, 5% put, 5% del, 90% get. With Handlers Do Free

A.1.2 Load Pattern: 45% put, 45% del, 10% get

Threads	Total Operations	Operations/second	Total aborts
1	18543212	617942	0
2	26084615	869249	52050
4	43722849	1457097	255696
8	71441946	2380684	909669
16	75353200	2511154	1364061
32	53866017	1795187	1020079
64	17088337	569481	375548

Table A.4: Red-Black Tree, CTL Deferred Update, 30 sec, 1000 keys, 45% put, 45% del, 10% get. Without Handlers

Threads	Total Operations	Operations/second	Total aborts
1	17075964	569036	0
2	25159362	838428	172020
4	39921454	1330343	740708
8	58295765	1942687	2054860
16	78430610	2613725	4914808
32	41393083	1379493	6903115
64	13101266	436642	5897352

Table A.5: Red-Black Tree, CTL Deferred Update, 30 sec, 1000 keys, 45% put, 45% del, 10% get. Without Handlers No Free

Threads	Total Operations	Operations/second	Total aborts
1	16369415	545498	0
2	23867026	795366	184577
4	39858796	1328246	909087
8	64887403	2162409	3410701
16	75844541	2527561	7970603
32	58946580	1964559	7547202
64	18468035	615556	5589514

Table A.6: Red-Black Tree, CTL Deferred Update, 30 sec, 1000 keys, 45% put, 45% del, 10% get. Without Handlers Do Free

A.2 Article Repository Performance Tests

A.2.1 Load Pattern: 5% put, 5% del, 90% get

Contention: 500 keys

Threads	Total Operations	Operations/second	Total aborts
1	166136	2748	0
2	170469	2822	165538
4	266720	4415	417455
8	332079	5500	734307
16	330438	5477	603804
32	332885	5516	527604
64	333818	5532	570060

Table A.7: Article Repository, CTL Deferred Update, 60 sec, 500 keys, 5% put, 5% del, 90% get.

Threads	Total Operations	Operations/second	Total aborts
1	162735	2693	0
2	166062	2748	0
4	162582	2692	0
8	168818	2793	0
16	163079	2698	0
32	164289	2718	0
64	163346	2703	0

Table A.8: Article Repository, Lock, 60 sec, 500 keys, 5% put, 5% del, 90% get.

Threads	Total Operations	Operations/second	Total aborts
1	58507	968	0
2	95854	1586	0
4	151386	2507	0
8	227568	3770	0
16	244586	4050	0
32	218117	3613	0
64	226025	3743	0

Table A.9: Article Repository, Only Database, 60 sec, 500 keys, 5% put, 5% del, 90% get.

Contention: 10000 keys

Threads	Total Operations	Operations/second	Total aborts
1	139150	2303	0
2	163784	2711	11067
4	267491	4429	32463
8	393465	6516	83208
16	419859	6960	95791
32	430640	7135	139903
64	441408	7315	331421

Table A.10: Article Repository, CTL Deferred Update, 60 sec, 10000 keys, 5% put, 5% del, 90% get.

Threads	Total Operations	Operations/second	Total aborts
1	139881	2316	0
2	142854	2364	0
4	140942	2334	0
8	139072	2302	0
16	138748	2296	0
32	139467	2307	0
64	138454	2292	0

Table A.11: Article Repository, Lock, 60 sec, 10000 keys, 5% put, 5% del, 90% get.

Threads	Total Operations	Operations/second	Total aborts
1	51712	856	0
2	79773	1320	0
4	119721	1982	0
8	178514	2958	0
16	220177	3645	0
32	214663	3556	0
64	215118	3563	0

Table A.12: Article Repository, Only Database, 60 sec, 10000 keys, 5% put, 5% del, 90% get.

A.2.2 Load Pattern: 30% put, 30% del, 40% get

Contention: 500 keys

Threads	Total Operations	Operations/second	Total aborts
1	28384	469	0
2	28567	473	25029
4	46318	766	70342
8	59812	990	188161
16	52713	873	296890
32	60405	1001	436798
64	65374	1083	440279

Table A.13: Article Repository, CTL Deferred Update, 60 sec, 500 keys, 30% put, 30% del, 40% get.

Threads	Total Operations	Operations/second	Total aborts
1	27268	451	0
2	27873	460	0
4	27295	451	0
8	28114	465	0
16	27414	453	0
32	27059	447	0
64	27309	451	0

Table A.14: Article Repository, Lock, 60 sec, 500 keys, 30% put, 30% del, 40% get.

Threads	Total Operations	Operations/second	Total aborts
1	17588	291	0
2	26203	433	0
4	40365	668	0
8	62042	1027	0
16	79545	1317	0
32	78067	1293	0
64	83454	1383	0

Table A.15: Article Repository, Only Database, 60 sec, 500 keys, 30% put, 30% del, 40% get.

Contention: 10000 keys

Threads	Total Operations	Operations/second	Total aborts
1	23429	387	0
2	27047	447	1141
4	43988	727	4499
8	68716	1138	14687
16	90947	1507	35703
32	91830	1521	60385
64	78896	1307	124938

Table A.16: Article Repository, CTL Deferred Update, 60 sec, 10000 keys, 30% put, 30% del, 40% get.

Threads	Total Operations	Operations/second	Total aborts
1	23427	388	0
2	23601	390	0
4	23494	388	0
8	23444	387	0
16	23282	385	0
32	23652	391	0
64	23073	381	0

Table A.17: Article Repository, Lock, 60 sec, 10000 keys, 30% put, 30% del, 40% get.

Threads	Total Operations	Operations/second	Total aborts
1	15912	263	0
2	22394	370	0
4	33654	557	0
8	51354	851	0
16	69128	1145	0
32	69788	1156	0
64	69340	1148	0

Table A.18: Article Repository, Only Database, 60 sec, 10000 keys, 30% put, 30% del, 40% get.

A.2.3 Load Pattern: 45% put, 45% del, 10% get

Contention: 500 keys

Threads	Total Operations	Operations/second	Total aborts
1	18805	311	0
2	19020	314	13318
4	30292	501	40364
8	39270	650	111900
16	35679	591	190234
32	38087	631	266670
64	43042	713	258884

Table A.19: Article Repository, CTL Deferred Update, 60 sec, 500 keys, 45% put, 45% del, 10% get.

Threads	Total Operations	Operations/second	Total aborts
1	18695	309	0
2	18569	307	0
4	18535	306	0
8	18836	311	0
16	18552	307	0
32	18655	308	0
64	18843	311	0

Table A.20: Article Repository, Lock, 60 sec, 500 keys, 45% put, 45% del, 10% get.

Threads	Total Operations	Operations/second	Total aborts
1	12175	201	0
2	17586	291	0
4	27520	455	0
8	42492	704	0
16	55064	912	0
32	54597	904	0
64	58368	967	0

Table A.21: Article Repository, Only Database, 60 sec, 500 keys, 45% put, 45% del, 10% get.

Contention: 10000 keys

Threads	Total Operations	Operations/second	Total aborts
1	15734	260	0
2	17969	297	840
4	29377	486	3028
8	46570	771	8233
16	57720	956	22495
32	61667	1021	41943
64	49731	824	63248

Table A.22: Article Repository, CTL Deferred Update, 60 sec, 10000 keys, 45% put, 45% del, 10% get.

Threads	Total Operations	Operations/second	Total aborts
1	15717	260	0
2	15475	256	0
4	15518	256	0
8	15508	256	0
16	15574	257	0
32	15840	262	0
64	15877	262	0

Table A.23: Article Repository, Lock, 60 sec, 10000 keys, 45% put, 45% del, 10% get.

Threads	Total Operations	Operations/second	Total aborts
1	11646	192	0
2	15579	257	0
4	23572	390	0
8	36427	603	0
16	48335	800	0
32	49362	818	0
64	50076	829	0

Table A.24: Article Repository, Only Database, 60 sec, 10000 keys, 45% put, 45% del, 10% get.

B

CTL Handler System API

B.1 Handler Function Types

```
1   typedef void (*_ctl_handler)(Thread *Self, void *args)
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.

Description:
- Type used for *commit*, *pos-commit*, *pre-abort*, *pos-abort* handlers.

```
1   typedef int (*_ctl_prepare_handler)(Thread *Self, void *args)
```

Return:
- **int** returns 1 if succeeded or 0 otherwise.

Parameters:
- `Self` Transaction Manager descriptor.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.

Description:
- Type used for *prepare-commit* handlers.

B.2 Handler System Functions

```
1  void _ctl_register_prepare_handler_priority(
2              Thread *Self,
3              _ctl_prepare_handler handler,
4              void *args,
5              int priority
6  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.
- `priority` Define the handler priority.

Description:
- Registers a *prepare-commit* handler indicating explicitly its priority.

```
1  void _ctl_register_prepare_handler(
2              Thread *Self,
3              _ctl_prepare_handler handler,
4              void *args
5  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.

Description:
- Registers a *prepare-commit* handler with default priority number 10.

```
1  void _ctl_register_commit_handler_priority(
2              Thread *Self,
3              _ctl_handler handler,
4              void *args,
5              int priority
6  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.
- `priority` Define the handler priority.

Description:
- Registers a *commit* handler indicating explicitly its priority.

```
void _ctl_register_commit_handler(
            Thread *Self,
            _ctl_handler handler,
            void *args
)
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.

Description:
- Registers a *commit* handler with default priority number 10.

```
void _ctl_register_pos_commit_handler_priority(
            Thread *Self,
            _ctl_handler handler,
            void *args,
            int priority
)
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.
- `priority` Define the handler priority.

Description:
- Registers a *pos-commit* handler indicating explicitly its priority.

```
1  void _ctl_register_pos_commit_handler(
2              Thread *Self,
3              _ctl_handler handler,
4              void *args
5  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.

Description:
- Registers a *pos-commit* handler with default priority number 10.

```
1  void _ctl_register_pre_abort_handler_priority(
2              Thread *Self,
3              _ctl_handler handler,
4              void *args,
5              int priority
6  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.
- `priority` Define the handler priority.

Description:
- Registers a *pre-abort* handler indicating explicitly its priority.

```
1  void _ctl_register_pre_abort_handler(
2              Thread *Self,
3              _ctl_handler handler,
4              void *args
5  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.

Description:
- Registers a *pre-abort* handler with default priority number 10.

```
1  void _ctl_register_pos_abort_handler_priority(
2              Thread *Self,
3              _ctl_handler handler,
4              void *args,
5              int priority
6  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.
- `priority` Define the handler priority.

Description:
- Registers a *pos-abort* handler indicating explicitly its priority.

```
1  void _ctl_register_pos_abort_handler(
2              Thread *Self,
3              _ctl_handler handler,
4              void *args
5  )
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `handler` Handler function pointer.
- `args` Pointer to a void type variable. Can be used to pass data into the handler.

Description:
- Registers a *pos-abort* handler with default priority number 10.

CTL Database Integration API

```
1  void TxDBStart(Thread *Self, SQLHDBC dbc, int roflag)
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `dbc` ODBC database connection handler.
- `roflag` Transaction is read-write if `roflag` is 0, it is read-only otherwise.

Description:
- Starts a memory transaction with support for accesses to the database associated with the `dbc` handler.

```
1  void TxStart(Thread *Self, int roflag)
```

Return:
- **void**

Parameters:
- `Self` Transaction Manager descriptor.
- `roflag` Transaction is read-write if `roflag` is 0, it is read-only otherwise.

Description:
- Starts a memory transaction.

```
1  int TxCommit(Thread *Self)
```

Return:
- Always returns the value 1.

Parameters:
- `Self` Transaction Manager descriptor.

Description:
- Commits a transaction, if transaction started with `TxDBStart` function it will also commit the database transaction.

```
1  intptr_t TxLoad(Thread *Self, intptr_t *addr)
```

Return:
- `Value of the transactional variable with address` *addr*.

Parameters:
- `Self` Transaction Manager descriptor.
- `addr` Address of a transactional variable.

Description:
- Loads the value of a given transactional variable address. The transaction aborts and retries if the variable has changed since the beginning of this transaction.

```
1  void TxStore(Thread *Self, intptr_t *addr, intptr_t value)
```

Return:
- **`void`**

Parameters:
- `Self` Transaction Manager descriptor.
- `addr` Address of a transactional variable.
- `value` Value to be stored.

Description:
- Stores the value value on the transactional variable with address `addr`. The transaction aborts and retries if the variable has changed since the beginning of this transaction.

Bibliography

[ALS06] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness*, pages 70–81, New York, NY, USA, 2006. ACM.

[BBG+95] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 1–10, New York, NY, USA, 1995. ACM.

[BK91] Naser S. Barghouti and Gail E. Kaiser. Concurrency control in advanced database applications. *ACM Comput. Surv.*, 23(3):269–317, 1991.

[BLM05] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Deconstructing transactions: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Publisher unkownn, Jun 2005.

[Cac07] João Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Universidade Técnica de Lisboa, Jully 2007.

[CMC+06] Brian D. Carlstrom, Austen McDonald, Hassan Chafi, JaeWoong Chung, Chi Cao Minh, Christos Kozyrakis, and Kunle Olukotun. The atomos transactional programming language. *SIGPLAN Not.*, 41(6):1–13, 2006.

[CRS06] João Cachopo and António Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[Cun07] Gonçalo Cunha. Consistent state software transactional memory. Master's thesis, Universidade Nova de Lisboa, November 2007.

[DB2] Db2 database management system. `http://www.ibm.com/DB2`.

[DLC08] Ricardo Dias, J. M. S. Lourenço, and G. Cunha. Developing libraries using software transactional memory. In *Proceedings of CoRTA (Compilers, Related Technologies and Applications)*. Instituto Politécnico de Bragança - ESTG, 07 2008.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Distributed Computing*, volume 4167, pages 194–208. Springer Berlin / Heidelberg, October 2006.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[EN00] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, 2000.

[FH07] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2):5, 2007.

[FLO+05] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2):492–528, 2005.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[Gra78] Jim Gray. Notes on data base operating systems. *Operating Systems*, pages 393–481, 1978.

[Gra81] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.

[Har03] Tim Harris. Design choices for language-based transactions. Technical report, UCAM-CL-TR, August 2003.

[Har05] Tim Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[HLM06] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 253–262, New York, NY, USA, 2006. ACM.

[HLMWNS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*, pages 289–300, New York, NY, USA, 1993. ACM.

[HMPJH05] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[HR87] Theo Haerder and Kurt Rothermel. Concepts for transaction recovery in nested transactions. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 239–248, New York, NY, USA, 1987. ACM.

[KR81] H. T. Kung and John T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[LC07] João Lourenço and Gonçalo Cunha. Testing patterns for software transactional memory engines. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 36–42, New York, NY, USA, 2007. ACM.

[MBL06] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2), 2006.

[MH06] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[Mos81] J. Eliot B. Moss. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Massachusetts Institute of Technology, April 1981.

[Mos06] J. E. B. Moss. Open nested transactions: Semantics and support. In *WMPI*, Austin, TX, February 2006.

[MyS] Mysql database management system. `http://www.mysql.com`.

[Ora] Oracle database management system. `http://www.oracle.com`.

[Pos] Postgresql database management system. `http://www.postgresql.com`.

[RSPML78] Daniel J. Rosenkrantz, Richard E. Stearns, and II Philip M. Lewis. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, 1978.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[Sco06] Michael L. Scott. Sequential specification of transactional memory semantics. In *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[SKS06] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, fifth edition, 2006.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.