Universidade Nova de Lisboa

Faculdade de Ciências e Tecnologia

Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Transactional Filesystems

Artur Miguel Adriano Martins

(26306)

Orientador

Prof. Doutor João Lourenço

Lisboa, 2008

Nº do aluno: 26306

Nome: Artur Miguel Adriano Martins

Título da dissertação: *Transactional Filesystems*

Palavras-Chave:

- Sistema de Ficheiros Transaccional
- Sistema de Ficheiros (SF)
- Memória Transaccional
- Memória Transaccional por Software
- Linux
- Transactional Locking II (TL2)
- Consistent Transaction Layer (CTL)

Keywords:

- Transactional File System (TFS)
- File system (FS)
- Transactional Memory (TM)
- Software Transactional Memory (STM)
- Linux
- Transactional Locking II (TL2)
- Consistent Transaction Layer (CTL)

To my beloved Carolina, whose shining enlightens my path.

Acknowledgements

After all the sweat and dedication, I could not finish my work without paying my humble and honest gratitude to the following friends.

To my supervisor João Lourenço for all his support and sincerity, not only as a mentor, but also as a good friend.

To Nuno Preguiça for sharing his brilliant opinions and suggestions.

To my sister Ana Luisa for all the support during these years.

To my beloved Father, Artur Martins, and my beloved Mother, Luisa Adriano.

To all my good friends who always stood by me. You know who you are.

Abstract

The task of implementing correct software is not trivial; mainly when facing the need for supporting concurrency. To overcome this difficulty, several researchers proposed the technique of providing the well known database transactional models as an abstraction for existing programming languages, allowing a software programmer to define groups of computations as transactions and benefit from the expectable semantics of the underlying transactional model. Prototypes for this programming model are nowadays made available by many research teams but are still far from perfection due to a considerable number of operational restrictions. Mostly, these restrictions derive from the limitations on the use of input-output functions inside a transaction. These functions are frequently irreversible which disables their compatibility with a transactional engine due to its impossibility to undo their effects in the event of aborting a transaction.

However, there is a group of input-output operations that are potentially reversible and that can produce a valuable tool when provided within the transactional programming model explained above: the file system operations. A programming model that would involve in a transaction not only a set of memory operations but also a set of file operations, would allow the software programmer to define algorithms in a much flexible and simple way, reaching greater stability and consistency in each application.

In this document we purpose to specify and allow the use of this type of operations inside a transactional programming model, as well as studying the advantages and disadvantages of this approach.

Resumo

A implementação de software correcto não é uma tarefa trivial, muito menos quando estamos perante um panorama que requer uma implementação com suporte à concorrência. Para superar este facto, vários investigadores apoiam a inovadora técnica de aplicar a abstracção dos conhecidos modelos transaccionais dos sistemas de bases de dados a modelos de programação, permitindo ao criador de uma aplicação definir conjuntos de operações perante o sistema como sendo uma só transacção e, assim, obter uma semântica previsível proporcionada pelo modelo transaccional em utilização. Protótipos deste novo modelo de programação são hoje em dia disponibilizados por várias equipas de investigação mas encontram-se ainda longe de estar acabados, dado a um número considerável de limitações na sua utilização. Em grande parte, estas limitações são derivadas da impossibilidade de utilização de funções de entrada e saída no interior de uma transacção. Estas funções são na sua maioria irreversíveis, o que inviabiliza a sua compatibilidade com o sistema transaccional, visto que o mesmo poderá eventualmente necessitar de abortar uma transacção e desfazer os efeitos de uma dada operação.

Existe no entanto um conjunto de operações de entrada e saída que não são necessariamente irreversíveis, e que podem proporcionar uma ferramenta de elevado valor quando disponibilizadas no modelo de programação transaccional sugerido previamente: as operações sobre sistemas de ficheiros. Um modelo de programação que englobasse as operações sobre memória e também sobre ficheiros no contexto de transacções, permitiria ao criador de uma aplicação definir algoritmos de uma forma muito mais flexível e simples, alcançando maior estabilidade e consistência na sua aplicação.

Neste documento propomo-nos a especificar e possibilitar a utilização deste tipo de operações num modelo de programação transaccional, bem como estudar as vantagens e desvantagens desta abordagem.

Index

List of Figures

List of Tables

1. Introduction

Writing concurrent programs is notoriously difficult, and is of increasing practical importance. Current concurrency abstractions are difficult to use and make it hard to design computer systems that are reliable and scalable. These abstractions are often based on low-level techniques like *locks* or *semaphores*. Furthermore, concurrent program built upon these techniques are always difficult to compose without knowing about their internals.

To address some of these difficulties, several researchers have proposed a new abstraction for programming models that derived from the transactional models of databases: the transactional memory models. Having the possibility to declare blocks of code as being transactions brings a new environment to concurrent applications programming, since the application can rely on the atomicity and isolation of a group of several operations; that is, either all happen or none, and all have a well defined semantics in presence of concurrency. Using transactional memory instead of locks brings well-known advantages derived from the properties of transactional models, like freedom from deadlock and priority inversion, automatic roll-back on exceptions or timeouts, and freedom from the tension between lock granularity and concurrency.

The key idea is that a block of code, including nested calls, can be defined as a transactional block, with the guarantee that it runs atomically and isolated with respect to every other transactional blocks. Transactional memory can be implemented using optimistic synchronization meaning that instead of taking locks, the thread runs without locking, accumulating a thread-local transaction log that records every memory reads and writes it makes. When the block completes, it first validates its log to check that it has seen a consistent view of memory, and then commits its changes to memory. If validation fails, because memory read by the method was altered by another thread during the block execution, the block is re-executed from scratch, or, its execution and all the changes made are simply discarded. This takes us to the problematic situation of declaring input/output (IO) operations inside a transactional block because of the impossibility to undo their effects.

Transactional memory eliminates, by construction, many of the low-level difficulties that plague lock-based programming. However, allowing support for IO operations in the transactional model is an obstacle difficult to overcome. The purpose of an IO operation isn't changing memory locations but instead changing other input-output locations like file system files, socket buffers, console buffers and such, leaving us with the problem of how to allow reversibility to these operations (needed when a transactions rolls back) in order to maintain the atomicity property. Some operations (like shutting down a remote computer, or writing to a console) are completely irreversible and should not be allowed within a transactional context.

However, IO operations that interact with storage devices (such as writing to a file system) can fit in a transactional model because the system can restore the contents back to their previous versions, allowing roll backs and guaranteeing atomicity. Common file system implementations today perform their primitive operations transactionally (like creating or deleting a file) in order to guarantee that the internal data structure (meta-data) that allows the correct storage of data remains consistent. The file system data structures are often complex and require that several locations are updated together in order to maintain their logic dependency. These composable operations must therefore be performed atomically; that is, either all the locations are updated correctly, or none are and the update will be totally ignored. Several different implementations of file systems achieve this by implementing an internal transactional model in their own way, using complex techniques like logging or reordering the disk writes and performing them asynchronously.

Actually, there is the possibility to operate with these two different models (the inner transactional model of a file system and the transactional memory model) in order to offer a model that allows not only to operate over memory locations but also over file system entities like files and directories.

1.1 Motivation

The motivation for this dissertation relies on the belief that a transactional engine can control accesses to both types of locations: memory and file system. This engine would allow performing memory and file operations as transactions, which would greatly improve the simplicity of writing consistent and stable programs by using the transactional programming model.

For example, imagine that a user is working on an application that has a large project opened, and in the event of saving the project, the operating system crashes, due to an unexpected error like a bug or a simple power failure. There is a strong possibility that the user has lost his project since the file saving process has been aborted in an unpredictable intermediate step. The project can become a combination of the new and the old versions of the project, becoming either a blended file or a group of files where some are new and some are old. When the user reopens the application, it will probably detect inconsistencies and abort opening the project.

This problem is parallel to the problem of the file system primitive operations that must perform together. File system designers implemented techniques to ensure that composable operations perform atomically, and we believe that our transactional engine can integrate the in-memory model and this file system model, offering the possibility for an application to declare several operations as a transaction, like the often long task of saving a project including all the in-memory computations required. With this technique, the application consistency would be guaranteed since a transaction is forced by the transactional engine to either commit successfully or roll back all the intermediate effects. In the previous application example, the user might have lost recent changes to his project, but at least the previous version remains accessible and consistent.

1.2 Solution

In order to accomplish this task, the resulting transactional model that cooperates with the two models referred, must be designed and implemented with precision and strictness. This model should be made available through an API implemented in the software transactional memory engine, offering special transactional primitives similar to the existing functions for memory and file system operations. The fundamental functions in the API are those that allow the possibility to start and end transactions, to read and write memory locations and to handle files, supporting their opening, reading, writing and closing. Applications that expect to take advantage of the transactional engine must therefore be predesigned and implemented to use this API. Also, the transactional model made available in the API must be correctly documented and structured, achieving foreseeable effects with its use. The API should implement as many primitives as possible to operate transactionally over memory and file system locations, but still

avoiding an exaggerated complexity. One of the main purposes of this solution is to simplify the complexity of writing consistent concurrent code, so implementing a demanding API would not fulfill our objectives.

1.3 Main contributions

The main *contribution* of this dissertation is the specification and development of a software transactional memory (STM) engine with file system support, based upon the further development of an existing STM engine: the *Consistent Transaction Layer* prototype (also known as *CTL*). We aim at performing the integration of this engine with an existing Unix-based file system, resulting in an STM engine with file IO operations support.

Additionally, the expected benefits of this approach shall be evaluated by using the appropriate tools to benchmark and compare the effects of our transactional engine.

The prototype implements a transactional model that best fits for the theoretical objectives, therefore being strictly and carefully designed. Our approach is therefore well documented and each design decision of the project was studied and tested towards certifying its righteousness.

We hope to contribute to the scientific community by studying the advantages and requirements for achieving an extended transactional support in nowadays programming models.

2. Transactional Models

In today's computer science, a *transaction* can be defined as a limited group of operations that respect specific model rules, allowing properties like stability and concurrency to exist in scenarios of sensitive applications like bank management systems.

A common example of using transactions is the event of transferring cash between two banking accounts, where the successful completion of a task depends on the successful completion of all the other tasks. In a cash transfer, a banking system needs to take a certain amount of funds from one account, then update this account withdrawing these funds, and then update the second account with the extra received money. So for the event of transferring cash between two accounts, it is unacceptable to perform only some of these steps correctly and others not. If the system allowed such a mistake, after the fund transfer the total amount of funds in the two accounts might even be different; the system could have taken funds from one account but not deposit them in the other, or the inverse, the system could have deposited in the second account but never subtracted from the first one.

All these problematic situations can be avoided if the banking system implemented cash transfers as transactions, since a *transaction* is group of operations that behave as a single operation: either all happen, or none. So after applying transactions in the example above, the system would either correctly transfer the money or simply do nothing, which is a convenient behavior.

Defining a simple transactional model [1], we can say that transactions are always in one of the following states:

- ***Active*** – Initial and working state. For the transfer example, the transaction remains in this state while operating successfully with the two banking accounts.
- ***Partially committed*** – The system puts a transaction in the temporary "*Partially committed"* state whenever the transaction tries to terminate with success. The system then tries to successfully commit the effects of the transaction.

- ***Failed*** – This is the state when a transaction has to be aborted, triggered either by the transaction itself, or by the transactional engine. The system will then perform the *undo* of the effects of any operation performed by the transaction until this point.

- ***Aborted*** – This is a final state when a failed transaction has been rolled-back. None of the effects of the transaction are now visible.

- ***Committed*** – This is also a final and successful state. When a transaction is in the "*Partially committed*" state and the system successfully performs all the committing operations, the transaction terminates with success and reaches this "*committed*" state.

The transitions between states are also strict and can be modeled in the next diagram:

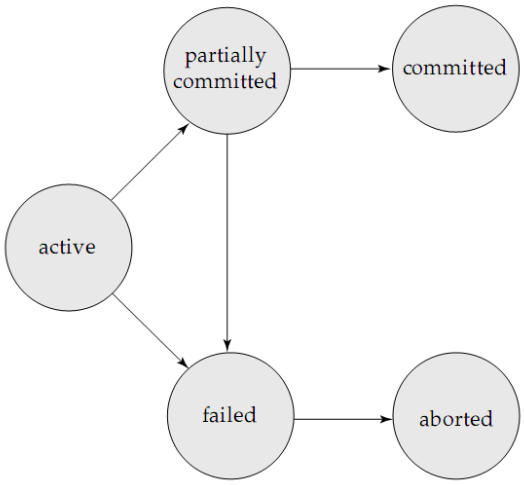

Diagram 1 - Transactional transitions model

The significance of each transactional state as well as the details for the transitions in the diagram will be explained as the basic properties of a transactional model are analyzed: The ACID properties of the transactions.

2.1 The ACID Acronym – Standard Transactional Properties

Although a transaction processing system can define specific rules for advanced processing patterns, the basic but important properties that are common to almost every transactional

context are known as the ACID properties of transactions. The word "**ACID**" is an acronym for ***Atomicity***, ***Consistency***, ***Isolation*** and ***Durability***.

Starting with the definition of ***Atomicity***, this property defines an "all-or-nothing" semantic for the set of operations inside a transaction. Since transactions allow the possibility to group a finite number of operations, the "*Outcome"* of a transaction defines whether all operations succeed, or all didn't even happened. The outcome of a transaction is a Boolean choice, defining success and effectiveness (*commit*) or failure and ineffectiveness (*abort*) to all the effects of the operation set. This choice is normally an action taken by the creator of the transactional object at the end of the operation set, however, both the creator and the transaction processing system can abort a transaction at any time if they want to. Aborting a transaction is generally a more complicated action then committing, since all of the effects of the inner operations have to be undone. For example, a transaction between a remote client and a local database system can perform a large number of data insertions over a table and, at the end, simply lose its network connection before having the opportunity to finish its job and commit the transaction. This situation forces the transaction system to abort and rollback the user transaction, undoing all the changes made to the table like if that set of operations had never happened.

This takes us exactly to the second property of the ACID acronym: ***Consistency***.

Consistency means that a transaction is a correct transformation of the system state, otherwise it aborts. The fact that the aborted transaction mentioned above triggered a rollback of all of its actions at the target system, allows not only the transaction to try and execute all of its operations again, but also to guarantee that the database system only changes from a consistent state to another consistent state, whichever was the outcome of the last transaction. This also implies that a transaction starts and ends with a consistent database, but it can be inconsistent meanwhile. A piece of data is consistent if it fits in all consistency constraints defined by the underlying transactional model, but the overall consistency of a database is guaranteed by committing a consistent transaction into a consistent system. It would be impossible to check the whole system against all the consistency constraints every time a transaction tries to commit, so the definition of system consistency becomes somewhat circular and dependant. Analyzing the Atomicity and Consistency properties of transactions together, we inevitably find that their importance is critical. Let's think about all the simple fund transfers between bank accounts that happen daily in the entire world. In a fund-transfer, the system has to make sure that the sender

account loses a certain amount of credits and that these exact credits are credited at the second bank account. If this transaction somehow fails in an intermediate step, the situation is rather delicate and must imperatively be handled. Both of the operations (taking money from one account, and adding it to another account) cannot occur individually but, if they do (if transaction failed), the system must return to its original state before the transaction, like if not a single operation of this transaction has happened (Atomicity property). Otherwise, the sum of the total funds in the two accounts could even be different, which is an unacceptable inconsistency for this scenario.

However, there are other possible situations that respect atomicity and consistency but result in unacceptable scenarios. This takes us to the third property of the ACID acronym: ***Isolation***.

The ***Isolation*** property is defined by the ability of executing multiple transactions concurrently, but without allowing side-effects between them. If we think about the previous example of a fund-transfer transaction and we consider that another transaction concurrently executes, this second transaction can read inconsistent values from the bank accounts (for example reading the balance of the first account right after it has been subtracted but still not added anywhere else), use these values, and then commit by its own, ignoring whatever the outcome of the previous transaction was. In this example, the system is consistent (the first transaction successfully rolled-back, and the second transaction successfully committed), otherwise the second transaction might have used values that never did exist consistently. The main idea behind the isolation property is that correctly executing concurrent transactions leave the system in the same state as if they had been executed one at a time in a specific order. This method is called "serial execution" [1], which is an effective solution but with a drastic performance break: no concurrency supported. Concurrency is essential for a performing system, mainly because of two good reasons:

- ✓ ***Improved throughput and resource utilization***

Transactions are a set of operations. These operations consume time and resources; some are IO requests, others are simply CPU calculations, but the fact is that computer systems allow parallelism between IO and CPU operations which is a potentiality that should be explored with concurrent transactions. One transaction can be running in the CPU while the second is waiting

for an IO request. This improves throughput: the number of handled transactions in a given amount of time.

- ✓ ***Reduced waiting time***

Without concurrency, transactions execute serially; one after another. One of the problems that outcome from this approach is that fast and small transactions might have to wait for other big transactions to finish processing, which leads to an unpredictable waiting time, no matter what the transaction estimated size was. If the transactions execute concurrently we reduce the average response time by sharing the system resources between them.

Other intermediate solutions between full concurrency and serial execution have been developed, normally considered as "isolation levels" (rule sets of locking degrees which occur when selecting data), and we will briefly introduce them below after we study the problematic phenomenon that happen without serializability.

Let's consider two transactions (T_1 and T_2) running concurrently in the same system, and accessing two objects named X and Y. The three typical phenomenons that happen in a transactional scenario are:

- ***Dirty Reads*** – Consider the following example sequence:

1)	**T_1** writes **X**	
2)		**T_2** reads **X**
3)	**T_1** aborts	
4)		**T_2** writes **Y**
5)		**T_2** commits

Step 2) is considered to be a *dirty read* since the value that was in the object X at the time of step 2) was uncommitted data. Even worse, that data could have been used by T_2 to write Y and then successfully committing these changes, based on rolled-back values.

- ***Non-Repeatable Reads*** – Consider the following sequence:

1)	**T_1** writes **X**	
2)		**T_2** reads **X**
3)	**T_1** writes **X**	
4)	**T_1** commits	
5)		**T_2** reads **X**
6)		**T_2** commits

This phenomenon of a non-repeatable read is found again in step 2). Like the name suggests, the instance of X read by T_2 in steps 2) and 5) can be different, no matter whether T_2 is reading committed or uncommitted data.

- ***Phantom Reads*** – This phenomenon is very similar to a *non-repeatable read*, however it refers to situations where operations have conditions to respect, such as a *where* clause in an SQL query. Some systems implement techniques to prevent *non-repeatable reads* that work correctly, but fail when the reads involve selecting different sets of data from different sources and with a specific restriction, for example when using a *where* clause. This is an anomaly deriving from the optimized granularity of the locks applied in a conditioned access to data.

Having these phenomenon stated, we can now study the four typical levels of isolation in transactional management systems. The objective of these four levels is to regulate a balanced operational state between concurrency and serializability, through defining the size (and shape) of the locks applied over data, by one transactional operation. Each level then fixes a different set of the stated phenomenon. The Table 1 illustrates the properties of the isolation levels based in the permission of the phenomenon just described:

Isolation Level	*Prevents*		
	Dirty Reads	Non-Repeatable Reads	Phantom Reads
Read uncommitted	✗	✗	✗
Read committed	✓	✗	✗
Repeatable read	✓	✓	✗
Serializable	✓	✓	✓

Table 1 - Isolation levels and phenomenon awareness

As stated above, the "*Serializable*" level of isolation prevents all kinds of concurrency errors, by strongly restricting concurrency. Generally in this level, a transaction that gains access to data also gains a full-system exclusive lock that will be only released when the transaction achieves its outcome and commits. On the opposite side, the "*Read Uncommitted*" is the lowest allowed level of consistency. It does not attribute locks at all, allowing a full concurrency support, but also allowing a range of problems that are hard to prevent.

Back to the ACID acronym, the last property left to study is ***Durability***.

The basic idea behind ***Durability*** is that when a transaction successfully commits, its effects are permanent and will not be rolled-back or undone by the system; intentionally or even by failure. Durability is normally a set of methods used to avoid abnormal functioning after system crashes, at all the possible intermediate states that the system may be. The more situations that are crash-recoverable, the more *Durable* the system is. Implemented techniques are often efficient and based on log files. For the previously stated example of the fund-transfer between two accounts, imagine that all worked as expected, but the system crashes right after it confirms committing of the transaction to the participant. An example of an approach to *Durability* in this situation is to log all the actions waiting to be done in the transactional log, *before* allowing the transaction to receive its committing confirmation. In this particular case of a system-crash, if the creator of the transaction gets the confirmation of a successful commit then that means that the system has securely saved all the information on what is left to do, and even if it crashes right at this moment, the system is able to fully execute the transaction like it meant to, by checking the log file and concluding that the transaction has not been applied yet. This way the transactional system guarantees its *Durability*: the transaction creator is sure that if he was able to commit his transaction, then the data he changed is permanent.

2.2 Common Extensions to the Standard Model

The traditional transactional model, defined by the ACID properties and concepts of serializability for schedules, becomes limited and insufficient for complex scenarios. Focusing on the Isolation property of the traditional model, it imperatively inhibits cooperation between two transactions, not allowing them to even know their relative order of execution since any serialization schedules of concurrent transactions is acceptable [2]. Today, the number of scenarios for concurrent applications that require features like multipart cooperation and long running computations is increasing, leading to the creation of several extended transactional models. These new models break some limitations of the traditional model, and enable a much larger set of possibilities.

The following section will focus on the main approaches to new transactional designs.

2.2.1 Nested Transactions

The first important step in the evolution of traditional (flat) transactions was the development of *Nested Transactions* [3]. Unlike the flattened transactions, nested transactions can be represented as a tree of transactions; one single transaction is named the *root* (or *top-level* or *outer*) transaction, and the others are *sub* (or *leaf* or *inner*) transactions.

A simple example of five nested transaction is displayed above in Diagram 2[1].

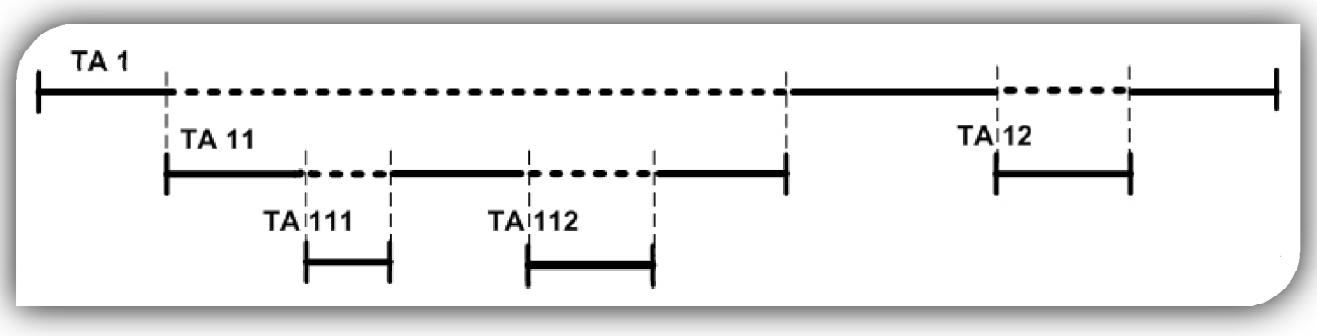

Diagram 2 - Nested Transactions Example

In the diagram, TA1 is the root transaction and TA11 is the sub transaction. Notice that TA111, TA112 and TA12 are not only sub transactions but are also *flattened* transactions, since they have no sub transactions and therefore their execution is continuous.

There are three main types of nested transactions [4] which can be defined by the *lock keeping* and the *outcome* behavior of the model:

- ***Flat Nesting*** – All locks acquired by a sub transaction are kept in the root transaction until its outcome, which implies that changes from sub transactions are visible to subsequent transactions in this tree, but not to other transactions outside this transactional tree, until the root transaction commits. When a sub transaction aborts, the entire transactional tree is rolled back.

- ***Closed Nesting*** – All locks are also kept until the outcome of the root transaction, however this model defines that the abort of a sub transaction only rolls back the effects of that sub transaction, and leaves the root transaction intact.

[1] Image taken from the "*Deliverable 4*" project web site in *www.edos-project.org.*

- ***Open Nesting*** – Locks are freed when the lock-acquiring transaction finishes, which means that the root transaction can see the results of its inner transactions. The outcome behavior of this model also does state that when a sub transaction aborts, the root transaction is allowed to continue.

These types of transactional models with nested transactions do not respect all the ACID properties in all transactions. For example, the *Closed* model guarantees all the four properties for the root transaction but not for its sub transactions; *Durability* cannot be guaranteed because the root transaction can abort and roll back the entire transactional tree effects. Even so, the *Closed* model is extremely useful if modularization is an issue: a complex operation is decomposed into sub operations, which are then executed using sub transactions.

2.2.2 Compensating Transactions

The outcome of a transaction can never be predicted by the transactional system or by the transactional programmer. Transactions can abort unexpectedly due to system issues or connection failures, which leads to the importance of being able to undo the operations performed by a running transaction. A *Compensating* transaction is conceptually a transaction made to undo the effects of a specific transaction. However this original transaction must be a set of reversible actions so that all of them can be *compensated* (undone) if necessary, leaving the system in its consistent state like before executing the original transaction. Obviously not all the operations possible inside a transaction are *undoable*; dumping data to an output stream or shutting down a remote computer are irreversible actions that the transactional system should not support inside transactional contexts. These *Compensating* transactions are useful especially for scenarios of long transactions, which will be explained immediately below.

2.2.3 Long Running Transactions

A transaction is considered *long* when its execution last considerably more than the average execution time of other transactions in the same system. Long transactions are abnormal between the others since they will probably collide with them due to long lasting locks, which inevitably causes drawbacks in the system performance. It is therefore necessary to avoid locks for long running transactions.

If the transaction is read-only, a very efficient technique could be used[5]: Consider a long running read-only transaction named α, and all the other concurrent normal length transactions named β. If the system is able to log the effects of operations performed by β, then it can always show the original system state to α like when it started, by undoing the effects of β operations (recorded in the change log). With this approach α can acquire fake shared-locks for an object. The other transactions can normally execute (eventually locking and changing data), while α can also see its data unchanged during all its execution, without forcing colliding transitions to wait or abort.

Another approach has been presented by *Hector Garcia-Molina* and *Kenneth Salem* in 1987 [6]. The *Saga Transactional Model* defines a long-running transaction as a set of sub transactions, all of them with a related *Compensating* transaction that can be triggered to semantically undo the committing of its associate. So either all sub transactions can commit, or all should be undone and rolled back in a chained reaction, all the way to the first sub transaction. This solution is largely used in the database transactional models from a long time ago, since it correctly embraces the problematic of the long lasting locks held by long running transactions. With *Saga* these locks can be freed between each compensatable sub transactions.

2.2.4 Distributed Transactions

The problematic behind *distributed* transactions can easily be understood if we imagine a single transaction that performs several different operations over two or more hostnames on a network. In order to maintain properties like *Atomicity*, the transactional systems running in each transactional participating hostname must coordinate and define a substantially consistent protocol.

A common algorithm for correctly distributing transactions is the *two-phase commit protocol*. The main idea behind applying *two-phase commit* to a distributed transactional environment is to use each transaction manager as two-phase commit agents, all similar to each other, which carry out the protocol's execution for each transaction. The standard protocol defines two phases: a commit-request and a definitive-commit phase. The former is used by the coordinator of the group (normally the agent which started the distributed transaction) to ask to all other agents for a commit possibility. All the participant try to execute the transaction up to the point where they are asked to definitively commit, registering the necessary entries in their

redo and undo logs, and finally the participants answer back to the coordinator with their outcome. When the coordinator receives all the acknowledgments from the participants, a definitive-commit request is sent. This request tells all the participants to try to commit and release locks for the specified transaction, again answering back their results. If at any time in the two phases the coordinator receives an *abort* from any participants, he notifies the rest of the group to perform their rollbacks and abort the overall transaction.

Again, the problem behind *two-phase commit* is that it holds locks while waiting for the protocol outcome, which can become a significant drawback specifically for scenarios of long transactions.

2.3 Transactional Memory

In 1977, *Lomet* observed that an abstraction similar to a database transaction might make a good programming language mechanism to ensure the consistency of data shared among several processes[7]. In the last few years, a lot of research and interest has grown around these topics, leading to the study of new software and hardware techniques for implementing transactional memory. In nowadays programming models, it is often required that concurrent threads cannot temporarily observe changes made by other threads (similarly to the *Isolation* property), which is normally implemented using locks that reserve access of an object to a single entity. Also, the coordination of concurrent threads is today a responsibility of the application programmer, using complex low-level techniques like semaphores or mutexes. If a programmer had a middleware available that implements a transactional model supporting Atomicity and Isolation for sets of computations, it would simplify the complexity of writing concurrent code while improving stability and modularity. He would then take advantage of the *Atomicity* property simply by wrapping normal computations around transactions, without having to worry about making sure that the effects of this computations would either all be visible, or none. Also, all the issues raised by an eventual concurrent access to shared memory would disappear. The Isolation property applied to the transactional memory scenario would guarantee that no transaction can see the changes that another transaction is doing, so, all executions would find a non-conflicting schedule, improving concurrency. Consistency is not a requirement for the Transactional Memory model since there is no standard organization of data; database transactions work over tables, cells, rows or columns, but memory transactions can work on very complex and flexible

data structures. However, the Atomicity property would also help in implementing *Consistency*, regarding that if a transaction runs on a previously-consistent system, no matter what the outcome of that transaction is, it will not produce an unpredictable set of results.

The forth property of the ACID acronym is also expendable to the Transactional Memory model. *Durability* states that a successful transaction always produces the effects it was meant to, and that they will be kept visible and durable on the system. Obviously, memory values (like variables or structures for example) are in volatile RAM and would require that the transactional memory model specified techniques to always save this data into persistent storage, keeping these values permanent on the system. This requirement would inevitably imply performance degradation, without adding useful features to the model, becoming pointless and disadvantageous to typical scenarios of transactional memory.

In contrast to not requiring Consistency and Durability, a good TM model should be aware of another important property: *Performance*. In database transactions, the data resides on disk rather than memory. Statistically, a single hard-drive access takes 5 to 10 milliseconds. In that same time, a single core processor could do millions of instructions, which means that computation is not a severe problem for database transactions. However looking into the accesses in transactional memory models, these are targeted to main memory, which incurs a cost of at most several hundred instructions. Therefore, a transaction engine should not perform too many computations at each operation. When we have in mind that memory accesses are very fast, it becomes easy to understand the problem of adding more computations to each operation. Comparing the overhead added to the original version of operations, the transactional calculations acquire a significant percentage of the total computations required per operation.

In sum, the main requirements for transactional memory are: *Atomicity*, *Isolation* and *Performance* [4].

2.4 Software Transactional Memory

Since the discovery of the possibilities of a Transactional Memory model, researches about possible implementations have gain consistency, essentially in the last few years. In 1991, *Herlihy* and *Moss* introduced the concept of Hardware Transactional Memory (HTM), showing that limited transactions could be easily implemented using simple additions to cache mechanism and exploring existing cache coherence protocols [8].

However, a hardware-only approach reduces flexibility since it creates physical limitations and strict protocol definitions, and so about a decade later, the first approaches to Software Transactional Memory (STM) implementations begun. Several different syntaxes appeared in STM prototypes since 2003, when *Herlihy*, *Luchangco*, *Moir* and *Scherer* [9] used a library with methods to control the transaction progress. Lately in 2005, *Tim Harris* proposed the *atomic* construct to represent a transaction[10]. A set of operations surrounded by the *atomic* keyword, represent an *atomic and isolated code block*. Finally around 2006, the authors of the first approach (*Herlihy* and *Moss*) also defined *transactional functions* and *methods*, where a programmer can define a method (or function) as being a transaction, and then later call it via a *transaction manager* [11].

STM appears to offer two main advantages over HTM:

- ✓ ***Flexibility***. Software is certainly easy to change. Additional implementations, features and improvements can be developed and easily deployed. Hardware has fixed limitations, due to fixed-size hardware structures like caches, and
- ✓ ***Portability***. STM is easier to integrate with existing systems and programming languages.

However, proposals for software transactional memory have some drawbacks related mainly to performance, due to the *overhead* of making all the necessary computations at the software level. In 2005 *Mark Moir*[12] described an approach to transactional memory models, which isn't only software or hardware. *Moir* defined *Hybrid Transactional Memory* (HyTM) as the challenge of supporting only some kinds of transactions in hardware, and others in software, not exposing to programmers all the limitations of HTM [12].

This dissertation fits on implementing a transactional filesystem using a software transactional memory engine, so the detailed study of both HTM and HyTM is outside the scope of this work.

2.4.1 STM: Common Features and Design Approaches

In this section we will go deeper with STM, studying its features and design approaches.

2.4.1.1 The Atomic Block

As referenced before, the *atomic* keyword[11] was presented to delimit blocks of code that should execute as a transaction.

```
void print_sum(int x, int y) {
    atomic {
        int K = x + y;
        print("Result: " + K);
    }
}
```

Code section 1 - Atomic block example

In Code section 1 above, the operations inside the *atomic* keyword are granted with the Isolation and Atomicity properties. All the functions inside the atomic construct also execute transactionally, so the *print* primitive inside the block will also inherit the same properties. One of the most important benefits of this syntax is that it enables a simple way to provide an abstraction to the transactional implementation.

2.4.1.2 The Retry Statement

Since there is no definition about the order in which concurrent transactions execute, there are a considerable number of scenarios where transaction ordering is crucial. For example, imagine that a transaction needs to compute a value based on the results of another transaction; the programmer does not know which one will commit first, so a technique to detect when is the right time for a transaction to execute would be useful.

*T*he *retry* [10] statement was introduced to support coordination, so that a transaction that reaches a step and executes a *retry*, aborts and then starts again. With this new statement, a transaction that finds that a certain element is not yet available, can simply ask the system to be restarted, hoping that its scope is different in the next execution and that the wanted element is

now present and ready to be used. In Code section 2 shown below we can see a useful combination of *atomic* and *retry* keywords:

```
atomic {
    if (buffer.isEmpty()) retry;
    Object x = buffer.getElement();
    …
}
```

Code section 2 - Retry example [22]

In this example we can see the *isolation* property: if the *buffer* hasn't got any element it will not acquire one in time for the *getElement* call. Therefore the transaction explicitly *retries* hoping that the buffer state will be different at the next execution, maybe thanks to the execution of a concurrent transaction. *Harris* also made the suggestion that the STM system should only restart the execution of the transaction when there are some visible modifications in that specific atomic scope, which would offer a great improvement to the normal retry approach, avoiding a busy waiting solution that would result in a large number of loops and therefore computations.

2.4.1.3 The OrElse Statement

Another useful mechanism to work with concurrency was also introduced by the authors of the *retry* statement: the *orElse* statement [10]. From the specified left-to-right evaluation, if a programmer uses an *orElse* statement, the left side will be primarily executed and in case of an abnormal termination or non-explicit abort, the right side is executed. This requires that *orElse* is used inside an atomic block, since it consists on the composition of two transactions and can also *commit*, *abort* or *retry*.

Obeying the left-to-right evaluation and considering that T1 and T2 are transactions used in order by an *orElse* statement (*T1 orElse T2*), the STM system would start by executing T1, and then follow the operation logic presented below:

1. If T1 terminates (commits or explicitly aborts), the statement is complete.
2. If T1 executes a *retry*, or non-explicitly aborts, T2 starts executing.
3. If T2 succeeds (commits or explicitly aborts), the statement is complete.
4. If T2 executes a *retry*, the *orElse* statement waits until a location read by either transactions changes, and then restarts (executing *T1 orElse T2*).

2.4.1.4 Defining Transaction Granularity

TM systems must implement methods to detect conflicts between two transactions when both reference the same *data unit*. Therefore the STM system must have a reference to all of the currently used *data units* and keep some metadata associated to it, obviously introducing overhead and memory usage when the number of references grows. For the same execution of a transaction in exactly the same situation and the same system-state, the amount of the overhead and memory consumption by the STM metadata, should be proportional to the size of the *data unit*, that is, the *transactional granularity*.

Different values of granularity are possible; with a coarse granularity design, the system would keep much less metadata and perform much less computation to control transactional conflicts, but it would frequently raise conflicts even in unnecessary situations, which degrades concurrency. With a fine granularity design, the system has to keep lots of references and do lots of computation; however, conflicts would be much rarer and would be very precise. Taking for example an array in a fine granularity design, the system would have to keep metadata for each position of the array, and do transactional computations for each access to it. On the other hand, if the array has size *n* then it would be possible to concurrently execute *n* transactions at the same time, each one changing its own position of the array, without triggering fake and inconvenient conflicts. The same array in a coarse granularity design where the whole array is considered a single transactional unit would have completely different results; if at least two transactions try to access it at same time, and one of them is updating the array, the STM system would raise a conflict. However, it would only keep a single metadata reference, and do a lot less computations.

As a result of this problematic, there is no correct definition for the most appropriate granularity, since it may vary depending on the system and the target scenario in which STM is to be used. Two main strategies for STM granularity are Word-based and Object-based.

In Word-based STM systems [13], concurrency is greatly improved and the fake-conflict situations are almost extinguished, but the transactional overhead and memory consumption are quite larger. On the other hand, the Object-based approach to STM systems [9] intend to group the fields of an object to the same metadata reference, allowing only a metadata entry in the STM

for each object in memory. Obviously, concurrent accesses to different fields of the same object will raise unnecessary conflicts.

2.4.1.5 The Direct and Deferred Update Methods

Typically, transactions operate accessing memory locations and possibly changing them. Because transactions can explicitly abort or be aborted by the transactional engine, the STM needs to be able to rollback to previous states, which can be achieved by either keeping track of changes made, or making changes in a cloned, temporary memory area.

These two types of approaches are called *Direct-update* and *Deferred-update*.

In the *Direct-update* design (which most recent systems use), the system allows a transaction to write values directly into the memory location in reference, having the change permanently written in the correct place so, no changes need to be made if a transaction commits. But to support rolling back, the system must also keep log of the previous version of the value of that memory reference, and then restore them one by one in reverse order.

On the other hand, the *Deferred-update* design states that when a transaction tries to change the value of a memory location, the STM system leaves the destination address untouched, and effectively makes the change in a private, cloned area. Transparently to that single transaction, reading the value of the same destination memory address would now return the new value not present in that address, but instead in the transaction's temporary copy. When a transaction commits, the system has to copy-over all the different data between the private transactional memory and the real memory. In contrast, when a transaction aborts, the system only has to clear that temporary private memory zone, leaving the real memory zone untouched.

These two approaches are quite different, and questions arise to unveil which one is the most performing. The answers are not perfectly clear, but it is sure that in a commit-optimistic system, the direct-update leaves data in place, so it is much more efficient; contrasting to a commit-pessimist system where the deferred-update leaves the data untouched, again being more efficient.

2.5 Consistent TL – An implementation of STM for x86 based on TL2

In the past year of 2007, *Gonçalo Cunha* worked on a Master Thesis at *Faculdade de Ciências e Tecnologia*, *Universidade Nova de Lisboa*, studying a possible implementation of a

configurable STM system for x86-based machines, running the Linux operating system with the *GCC* compiler [4]. The thesis was named "*Consistent Transactional Layer"* (or *CTL*) [4]. For a prototype implementation, Cunha decided to begin with the implementation of STM presented by *Sun Microsystems* named *Transactional Locking II* (also known as "*TL2*"). TL2 was designed for SPARC machines and implemented using the SUNPRO C compiler, so the implementation of CTL started by the hard task of porting the TL2 system to x86.

2.5.1 Approaches and Features

TL2 [14] was chosen among other implementation as a start, since it presented certain desirable features that other STM systems did not. TL2 is a locking STM implementation that performs locks at commit-time only, holding locks for very short duration times, guaranteeing seamless compatibility with *malloc/free* memory management systems. Also, TL2 implements a new technique for a consistent STM engine, with the introduction of *global version clocks* which help to validate if a transaction is on a consistent state.

However the CTL prototype achieved more stability and feature-richness than the original TL2. We will study some of these improvements in the following sections.

2.5.1.1 Explicit User Aborts

Allowing a transaction to explicitly abort was not implemented in the original TL2 system. CTL implements the user-specified abort which can be convenient for many scenarios. There are a considerable number of algorithms that perform calculations in the starting steps, but reach a conditional phase where its execution might no longer be necessary or possible, and the algorithm aborts. The effects of the calculations made before the abort need to be undone in the system, however, the transaction did not abort unexpectedly. In this scenario, the user-explicit aborts are logical and appropriate. For example, a mailing system that processes multipart messages tries to put back together all the pieces from each message. However if a time-out is reached or a part was lost in the transfer, the system explicitly wishes to roll-back all the effect of composing an incomplete message.

2.5.1.2 Automatic Transaction Retry

In order to avoid forcing the programmer to check the value of a *commit* call in a transaction and probably putting the transaction code block inside a loop, CTL engine implemented an *automatic retry* feature, which defines an *exactly one successful commit* semantic.

```
do {
    TxStart();
    a = TxLoad(x); //possible collision
    result = TxCommit();
 } while ( !result );
```

Code section 3 - without Automatic Retry

```
TxStart();
a = TxLoad(x); //possible collision
TxCommit();
```

Code section 4 – with Automatic Retry

Code section 3 shows a situation where the programmer had to check the value of the *commit* call in order to make sure that everything was successfully completed. The right side shows the simplicity achieved by the automatic-retry semantics.

2.5.1.3 Nested Transaction Support

Consistent TL supports transaction nesting in a way similar to the *closed nesting* technique studied above. The system behaves differently for user-explicit aborts and system triggered aborts (like in colliding transactions, for example). If a sub-transaction chooses to abort, only the effects of this sub-transaction are rolled back and the parent transaction remains operating normally. If the sub-transaction collides with another concurrent transaction, the system triggers an abort, and the whole transactional tree is rolled back and retried. In this model, the use of large nested transactions in a high collision rate scenario is not useful, suffering the same problems as non nested or flat nested transactional models.

2.5.1.4 Update Strategy and Validation Modes

The update strategy in the original TL2 system was based on a deferred update technique using a *Redo log*. In this method, the changes to memory values are updated in a temporary copy of the real values, privately associated with the current transaction. This associated log records every change and applies each one to the real locations at commit time only. If the transaction aborts, the log is simply discarded.

Along with the logging technique, a consistent state validation algorithm is performed before and after each read operation. First, when a transaction starts, it reads the *global version clock* into an internal and private transactional *timestamp*. In the event of a memory read access, the algorithm ensures that the location has no locks and, if so, the *version clock* associated with that location is checked before and after reading the value. This allows detecting any possible changes by another process. If at both checks the version clock of a location is lower or equal to the current transaction timestamp, the variable has not been changed. The write accesses do not implement any intermediate algorithm. Each write simply logs the write attempt to a write set, which is later processed at commit time. When committing a transaction, the system locks all variables waiting to be written, checks for consistency in all events logged in the read set, increments the global version clock, copies each operation from the redo log to their real locations and finally, releases all the locks with the updated value of the version clock.

In the original TL2 implementation, the transactional granularity was fixed at a fine grain design, treating every possible location as words [14]. Word-mode also allows access to objects, treating each internal field individually as words. As studied above, this granularity design can improve concurrency and collision-detection but degrades performance due to taking more memory for transactional meta-data, and more CPU processing for transactional operations.

The CTL prototype not only implements the Deferred Update/Word Mode technique mentioned above, but also introduces a *direct update* mode (using an Undo Log) and a coarser granularity design possibility by using Object Mode as an alternative to Word Mode. In the direct update technique, the main differences for the deferred update is that variables (in this case words or complete objects) are stored in their real locations instead of being stored in a temporary location, and the value that was on the real location is backed-up to the undo-log in order to allow aborting the transaction and performing rollback. The consistency validation mode is very similar to the algorithm explained for Deferred Update/Word Mode.

The special Object Mode is a valuable modification to the STM engine. This mode allows the engine to store only a single entry in the Undo Log for several read accesses to fields within the same object, and allows each field to be accessed directly (after the corresponding object open calls) with a traditional, non-transactional API. Even being advantageous for some situations, this type of approach has two problems: first, the transactional granularity is coarser which brings all the disadvantages explained in 2.1.4.1.4 (*Defining Transaction Granularity*);

second, making object fields accessible from outside the transactional API opens the possibility for the transaction to run in an inconsistent state, since another transaction can concurrently change the same location. The CTL API implemented dedicated functions to operate in this mode, including a check to detect if the transaction has entered an inconsistent state, by explicitly comparing the version clocks of an object and the current transaction.

2.5.1.5 The new Handler System

A late new feature that makes CTL even more flexible is the *Handler System,* developed by *Ricardo Dias* for his Master Thesis [15]. This system allows the CTL programmer to define a set of specific functions to be called automatically in well defined transactional states. *Dias* defined five intuitive new states for a transaction:

- ***Pos-Start*** – The transaction has just made the required calls to begin, but has not performed any operation.
- ***Pré-Commit*** – The transaction triggered a *Commit* call and its effects are about to be made permanent in the system. This state lies between the last transactional operation and the start of the commit call execution.
- ***Pos-Commit*** – This identifies the state where the transaction has correctly committed to the system and is about to end.
- ***Pré-Abort*** – The transaction has, explicitly or not, been triggered to abort. This state lies between the normal execution state and the *abort* phase.
- ***Pos-Abort*** – This is the state where the transaction has already aborted and will be ended, not retried. This state is only triggered if the user aborts explicitly without the *retry* mode specified; otherwise the transaction will be automatically restarted and *pré-abort* will precede *pos-start*.

Having these practical states defined, *Dias* developed a way to assign function calls to when a transaction is in one of these states, which means that the CTL programmer can define *Handlers* that happen for a specific transaction in one of those five states.

This *Handler System* is a *must* for transactional memory models that intent to allow operations that are not easily undone as a memory change.

A simple example to justify the *Handler* use is a memory allocation call made inside a transaction (*malloc*). If for example the variable was named *myVar*, and the transaction is retried a couple of times, the application will keep allocating free memory to the *myVar* variable without ever freeing it. By adding a *pré-abort handler* that frees the variable, at each transactional retry, the action "*myVar = malloc(size)"* will be undone by "*free(myVar)*", and the allocated memory will not be definitely lost and crushed by a new allocation request, but instead it will be freed and then reallocated correctly.

2.5.2 Prototype API Specifications

The CTL engine prototype offers an API that allows the use of transactional memory in a normal application simply by using the CTL transactional versions of IO functions to read and write memory locations, and by declaring them within transactional code blocks. Also, the prototype offers special functions to operate in either Word or Object modes and to enhance the transactional control.

More specifically, the API declares a data structure called "*Thread*" that intuitively represents an executing thread. A normal application thread can declare this structure and initialize it by calling "*TxNewThread*". After that, the application can use this obtained structure to start CTL engine transactions with it, calling the "*TxStart*" procedure from the API with the new "*Thread*" data structure acquired. At this point, the application thread represents a transactional thread, which means that it can operate with the CTL API transactional functions to load and store memory words or objects transactionally. When operating in object mode, it is possible to check if a transaction is running in a consistent state by calling "*TxValid*". Finally, the application thread can call the functions "*TxCommit*" and "*TxAbort*" to respectively commit or abort all the transactional operations declared since the "*TxStart*" call. Therefore, the transactional operations used between *TxStart* and *TxCommit/TxAbort* behave all together atomically and isolated, or in other words, transactionally.

3. Evolution of the Unix-based Filesystems

In 1969, *Bell Labs* received their first hard disk which was freely assigned to the *Multics* operating system team [15]. One of the team members, the engineer *Kenneth Lane Thompson*, started experiencing a few data representation techniques, creating a simple filesystem that allowed primitive operations over files and the static creation of directories. Although this filesystem was not totally useful, it did help *Ken Thompson* in defining and testing the principles of a file system.

In March 11th, 1971, *Ken Thompson* and *Dennis Ritchie*, both ex-members of the then abandoned *Multics* project, released a new operating system called *UNIX*, packed together with the *Unix Programmer's Manual* for *Thompson*'s *B* programming language. This operating system was later re-written in 1973 using the new *C* programming language developed by *Thompson*'s partner, *Dennis Ritchie*.

About a year later in 1974, the first UNIX-based and C-based filesystem was released: it was named as *File System*, and lately named as *System V File System* (commonly referred to as *S5FS* or just *FS*). The *FS* system was simple and slow, but powerful. It had a single *superblock* and multiple *inodes* in the disk first blocks, followed by raw data. The superblock described information about the storage characteristics, and the inodes were used as indexes to locate files on disk and provide meta-data. Even though file systems have suffered enormous improvements, the simple definition of superblock, inodes and even the system hierarchy have not changed much since FS. Thompson and Ritchie really did spend quite a lot of thought on file systems, and did worry about designing structures that fit in a futuristic point of view.

In 1984, Bell Labs researcher named *Marshall Kirk McKusick* released the *Berkeley Fast File System* (*FFS*) [16] which solved many problems of the original *FS* and greatly increased the hard-disk bandwidth from 5% (*FS*) to near 47%, achieving a greater performance. Several techniques like *Block fragmenting*, *cylinder grouping* and better layout policies allowed the FFS to take greater advantage of the underlying hardware, also increasing flexibility and usability.

3.1 The Unix File System Architecture

As mentioned before, the structural design purposed by *Thompson* and *Ritchie* for S5FS has a clear vision on how a general file system implementation should be designed. Many characteristics of FS are still used in today's file systems, although improved and optimized to overcome certain problems. We will now briefly study some design approaches of FS and their evolution, as well as some new techniques developed in later file systems, mainly in FFS.

The original implementation started by specifying that the disk is split into partitions, and each is formatted with its own file system. The concept of a *File*, a stream of data aggregated under a common *filename,* provided a very important abstraction in the programming field as well as a good method to organize data. Files are normally organized in a tree-like hierarchy where intermediate nodes are named *directories*.

A structure named *superblock* [17] is responsible for keeping several information about the partition, such as its type, total number of data blocks or a pointer to a linked-list of free data blocks (in original S5FS). Every existing file also has a simple structure called *inode*, responsible for keeping important information about the corresponding file such as its type, its access mode, two timestamps for access and creation times, the number of bytes in the file, number of links to this file, an integer representation of the file owner and its group and also a link to the first data-block used by the file. Also, inodes were originally structured to define several levels of indirection, allowing the file system to define files that cannot be represented by a single inode. This was achieved by referencing the next inode at the end of the current inode. Data blocks could only be used for a single inode, wasting a lot of disk space whenever a block was used to write just a couple of bytes. Lately with the release of Berkeley's FFS, data blocks allowed internal fragmentation in order to solve this problem, allowing blocks to be split into multiple sub-blocks.

Another problem of the original FS was the fragmentation of the file system after deleting and inserting several files. The distribution of data across the system became sparse (fragmented), since deleting a file adds its corresponding data blocks to the free data-blocks list, and adding a new file sequentially uses these blocks again to spread the file's data, degrading the seek-times required to read all blocks from a file. Approaches to solving this problem vary from system to system and even in the most recent file systems, fragmentation is never totally avoided. The most common solution to avoid increasing fragmentation is the periodic task of

defragmenting; physically reordering the blocks assigned to each file in order to keep them sequential or *close* to each other, for each file.

Other important improvements introduced by FFS focused on optimizing the seek-times. A cylinder is the group of the physical disk-blocks on a plate that are equidistant from the plate center. Since plate rotational seek-times are lower than the times required to realigning the magnetic sensor to a new cylinder, the system should always try to assign data-blocks for a new file within the same cylinder. Also, FFS specified that each cylinder should not only keep a copy of the system meta-data (the superblock), but also a *bitmap* of the cylinder's free inodes and data-blocks, eliminating the primary necessity for a free-block linked-list.

FFS also introduced the file locking system. A process could then acquire a shared lock for reading a file, allowing the file-system to temporarily put the file in read-only mode thus allowing more shared locks for another processes; however, a single process could also request an exclusive-lock to make changes to the file and blocking file access to any other process rather than the exclusive-lock owner.

Finally, another important feature of FFS over FS was the introduction of *soft-links*. FS implemented *hard-links* as a reference to the first inode of a file, structurally inhibiting the creation of links to files stored in remote file systems or even in another partition. FFS introduces soft-links as a reference to a complete UNIX path to the target file or directory, enabling the creation of links to a wide range of useful scenarios.

3.2 Designing Consistency

In a predictable and stable scenario of events in an operating system, the data structures explained above allow us to guarantee that the file system is always in a consistent state. The system has a set of metadata structures that identify and define all the other data blocks existing on the disk, updating both these types of blocks in a specific and correct sequence whenever it is necessary. Systems are not static or totally predictable and they have to be designed and prepared to simple but problematic events, like a power failure in the middle of an operation.

In the specific example of appending a couple of megabytes to an existing file, the system has to use new blocks, fetching their indexes and updating them in the free-block bitmap, finding a free inode and initializing it to record the new blocks of the file and its directory path. Finally, the system can actually write the new data blocks physically on the disk. Any external events

that consequently inhibit these actions to perform correctly and together leave the logic of the file system structure invalid; the file system is inconsistent. In the given example, if the system crashes after the meta-data update but before the system writes the data blocks, after booting the system, the file system would list a new file size and internally a new set of inodes and data blocks, but in reality these data blocks may be filled with other old file contents, which corrupts the original file and opens an unacceptable hole in the security of the system.

An extra premise is then added to the task of designing a good file system: crash recovery and guaranteed consistency.

Several file systems have purposed and implemented extraordinary techniques to partially or totally fix this problem, also facing some new design problems. Implementing consistency is complicated since it easily adds an operational overhead slowing down performance as well as adding long waiting times for checkups when necessary, normally at system boot after an incorrect shutdown.

This brings us to the study of two excellent approaches to file system consistency: the *Soft-Updates* and the *Journaling* techniques.

3.2.1 The Soft-Updates Technique

The consistency of the file system structure in FFS was guaranteed by the synchronized writes to disk. The file system had to perform writes in a proper sequence and in synchronized (waiting) mode, which added an overall low performance but guaranteed meta-data consistency. The ordered writes mean that the disk has to seek for specific blocks one by one and perform their correct changes. Seeking takes a considerable time. Also, the disk writes were synchronous which meant that even after the disk spent time changing to the correct position, it also had to spend more time to actually write the data physically.

In 1999, sixteen years later since the release of FFS, its author (*McKusick*) teamed up with *Gregory R. Ganger* and purposed a brilliant technique to overcome the consistency and performance limitations. The *Soft-Update* technique [18] relied on delaying disk writes to a memory buffer (*write-back caching*), performing them preferably when the disk and the system are less busy, and in a strategic order to minimize seek-times. Soft updates have been proven to reduce the disk write by 40 to 70 percent, especially on file-intensive systems such as web sites or mail servers for example.

Consistency was guaranteed by a complex dependency-tracking technique. The Soft Updates system performs write-back caching which delays the real disk writes, but enables the possibility of processing metadata-aware algorithms to calculate dependencies of each delayed write, monitoring and tracking dependencies for twelve types of different possible changes to the file system structure.

With an extremely detailed level of dependency information for each block change, even the circular dependencies problem for related disk-writes was solved. For example, when a buffer of inodes needs to be written on disk, the list of safe and consistent inodes is processed, and the other not yet consistent inodes are temporarily rolled back to their last safe values, waiting for the disk data writes to complete. At that point, the rolled back inodes can be rolled-forward to the current correct values. Since the soft updates write buffer is locked throughout the time in which the contents are rolled-back, any processes that try to use the buffer are blocked until the roll-forwards are performed [18].

This advanced technique is the key to guaranteeing that the file system on the disk is permanently in a consistent state. Reducing the number and types of corruption, soft updates were designed to ensure that the only eventual inconsistency is lost resources. The write order dynamically chosen by soft updates algorithms ensure consistency to the metadata level, but not to new data blocks. Only after new data has completely been written to the disk, the metadata updates are performed. In the event of a system crash or instant reboot, the physical file system has a consistent state and an integral metadata structure including all its related data-blocks. The new blocks however may not belong to any inode, and will be marked free again in the next run of the file system check utility.

The File System Check Utility (*fsck*) is the entity responsible for validating the consistency of the file system. Historically, fsck task was to check all orphaned-inodes (inodes that were marked as in use but were unreferenced by the system) and recover them to the existing file system in the *lost+found* directory. However the result of running this procedure over a system with soft updates would fill the lost+found directory with lots of partially deleted files. The fsck utility must check the system to verify if it is running soft updates and if so, fsck must delete orphaned-inodes instead of recovering them. Derived from the structural design of the soft updates technique, fsck also has another important simplification: it can always rely on the information presented in the bitmaps and thus checking only the subset of inodes that the

bitmaps mark as in use. Although some of the inodes marked in use may be free, none of those marked free will ever be in use.

Another interesting feature of soft updates is the *snapshot* capability [18]. A *snapshot* is a file representing a frozen image of a filesystem at a given instant in time. The advantages of having a snapshot of the file system vary from taking a full backup to running fsck in a live-running system, all without having to stop the file system operations for a considerable time. A simple change to fsck in order to make it work over snapshots and not over the partition provided instant crash-recovery boot up. It is completely safe to begin using the file system after a crash without running fsck; however, there is a large possibility for some loss of file system space in each crash. Therefore, if the system detects an unclean shutdown at boot, it should take a snapshot, return to normal operation, and run fsck in the background, checking the snapshot for the existence of the mentioned orphaned-inodes. At the end, fsck returns a list of the new bitmaps, and the system computes the task necessary to effectively delete the dirty inodes.

3.2.2 The Journaling Technique

This second approach is the most used in commercial implementations of file systems and is commonly named *Journaling* or *Log Enhancement*. The name derives from the use of a *journal*, like an operation circular *log* file, where all the changes are saved before actually being written to disk.

Since the write buffer enforces a *write-ahead logging* protocol, all the operations that are waiting to become effective are previously registered in the persistent storage log, allowing the system to be able to maintain its consistency even after an unexpected failure. At a system reboot, it is possible to redo the operations present in the log that have not been completed successfully, bringing the system back to a consistent state before allowing its use.

There are two different types of journaling, mainly differing in the choice of the amount of information to save in each log entrance; meta-data and data journaling, or meta-data-only journaling. It is safer to log all block changes to the journal file, allowing full operation restore whenever necessary. However, two issues arise from this approach: each disk write is doubled (one write in the log and other in the real destination) which obviously degrades throughput to less than a half, and the second issue is the log size which is considerably large. This type of journaling is one of the few consistency approaches for file systems that totally guarantee that no

data losses occur unexpectedly. The second type of journaling (and most used in common file systems like *ReiserFS*, *Ext3*, *XFS*) is based on logging only the meta-data updates to the log, which results in a log file of approximately one percent of the entire partition space and increases the throughput by eliminating the double writes requirement [19]. It is therefore evident that meta-data journaling lowers the guaranteed consistency since the system will not be able to recover the data blocks of a specific operation when necessary. Meta-data-only journaling is a compromise between reliability and performance, ensuring that the system can be recovered quickly when next mounted, but leaving an opportunity for data corruption in the event of a synchronization failure between unjournaled file data and journaled file meta-data.

Ext3 is an example of a meta-data-only journaling file system, improving the precedent Ext2 recovery methods. The complete Ext3 file system was designed by one of the authors of the original Ext2 (*Stephen C. Tweedie*) from 1998 to its release date within the Linux Kernel 2.4.15 in November 2001. In his first paper describing the evolution of Ext2 to Journaled Ext3 published in 1998, Tweedy already had a clear idea on how to implement the meta-data-only journaling technique, correctly predicting that the meta-data-only journal file only needed three different types of elements:

- ***Metadata blocks*** – A metadata block entry in the journal file contains a complete block of metadata that needs to be physically deployed. This means that if an operation needs to change just a small part of a metadata block, the entire block (including the recent change) is dumped to the journal file. When the system effectively writes this block to disk, it has to write the full block completely. Tweedy specified two reasons for this approach:
    I. Journal writes are sequential and therefore fast enough. Also, these writes can be batched into large clusters, which are efficiently handled by the disk controller.
    II. Working with the entire block instead of working with a specific sub part (the changed part) spares the system from a lot of CPU work.

- ***Descriptor blocks*** – A descriptor block is always written to the log sequentially before metadata blocks, since they describe how many metadata blocks are written ahead, and where are their physical destinations on disk.

- ***Header blocks*** – These types of blocks are like journal indexes. They are written at fixed locations, and they contain a unique sequence number, and a reference to the head and tail blocks currently present in the log. At a system recovery, the header block with the highest sequence number is searched, and then the values for the head and tail blocks contained in that header block are used to specify the recovery run length.

The problem with any of the mentioned journaling techniques is that they inevitably add performance degradation due to the journal processing computations, and to the additional required IO operations to save information to the journal file. Several improvements like sequential operation batching and optimizing the write order minimize this overhead to acceptable levels. The fact of having an asynchronous write of the file system operations to the journal file implements a semantic similar to Soft Updates, which opens the possibility for properly reordering writes for disk throughput improving, but does not guarantee durability for every operation.

More than half of the Unix-based file systems choose journaling for maintaining consistency, but each one implements journaling in its own way. Choices vary in details like log size and location, log elements and their contents, recovery and checkup algorithms performed by fsck, but generally journaling is easy to implement and to integrate with existing file systems, providing a stable consistency model.

3.3 Transactional Filesystems

The definition of a transactional file system is normally ambiguous. The file systems that we have previously studied implementing consistency and crash recovery techniques, can be considered as *internally transactional* since they can atomically group a batch of operations which either all complete with success (commit) or all are discarded (abort). In the example of creating and writing to a new file, every file system has to perform several meta-data and data writes and all of them need to be performed in order to leave the system consistent. This group of operations that must perform together can be considered as a *file system transaction*.

Taking Soft Updates as an example, the algorithm performs the data writes first, and then schedules the meta-data writes in order to make the file accessible. However, if the meta-data writes fail, the data-blocks will be freed in the next execution of fsck which guarantees atomicity

for this “create file” transaction. Journaling file systems can also define a transaction for this file creation example, since whenever the journal records a meta-data update, the system guarantees that it will be correctly performed as soon as the log is processed; at the next schedule or at a system recovery event.

These implementations of transactions for file systems exist internally to guarantee that the primitive file system operations, like creating or deleting a directory, always perform correctly and never break the consistency of the file system data structures. However, it would be powerful and useful to export this transactional possibility to the application level, allowing a process to abstractly group a set of IO operations, granting transactional properties for this group as if it was an internal file system transaction.

Similarly to the abstractions studied before in transactional memory, implementing a transactional model for sets of disk IO operations would become quite useful for common applications. In the example of an application that saves personal user settings in profiles (like the GNOME Window Manager, or the Mozilla Firefox web browser), these profiles consist in complex groups of files which interconnect and all together define a user profile. A simple unexpected application crash or system reboot could happen in the middle of a profile-update process, possibly preventing this application to work correctly in the next execution or even worse, requiring that the user deletes the personal profile losing all the configurations. In this scenario, if the application programmer was able to work over an IO API that could define the whole operation of saving a profile as a *transaction*, the application would never meet an unexpected profile configuration, since the update would either be correctly written, or totally ignored.

Back to the definition ambiguity, we believe that a transactional file system not only should be *internally transactional* (meaning that the standard operations over the disk structures perform correctly) but it also should be *externally transactional,* providing an API to export a consistent transactional model to the programming level.

A few approaches for a fully-transactional file system have been proposed in the scientific world, like the *Amino FS* [20] or the *PerDiS FS* [21], but all of them use a relational *database management system* (or *DBMS*) as a background transactional engine. This approach simulates a transactional file system by transparently transforming all the file system data structures and their methods to relational data in a SQL database. The transactional model exported by these

implementations is therefore inherited by the underlying transactional model exported by the database. Although these approaches proved to work successfully for certain scenarios [20], their portability and system overhead resulting from having a full DBMS running permanently in the system background were a major limitation.

Another approach that would overpass this limitation is possible, by implementing the external transactional engine *on top* of an existing file system. The only implementation known today is the new *Microsoft Transactional NTFS* (also known as *TxF*) which uses an operating system component named *KTM* (*Kernel Transactional Manager*) to act as a transactional engine that offers an API to export a transactional model for file system operations. Although efficient, this implementation is proprietary and not open-source, thus obliterating all the goal of having a portable and configurable implementation of a transactional engine to run over an existing file system.

3.3.1 The Transactional Models for File Systems

As we mentioned before, the underlying transactions in internally-transactional file systems have the objective of maintaining data structure consistency while implementing fast recovery algorithms. Returning to the ACID acronym, we can say that the following properties hold the internal transactional models of the studied file systems:

- ***Atomicity*** is one of the most important properties existing in this model, allowing consistency for file system primitives (like the creation of a file) which need that a set of write operations *behave* atomically.

- ***Consistency*** can also be found in this model, since the data structures on the disk (containing meta-data and data) are maintained healthy. In database systems the Consistency property has a different perspective, which is translated to different assertions; for example, a foreign key in a table refers to an existing primary key of another table. In file systems, Consistency is the health of the underlying data structures that support the storage. Atomicity is normally a requirement to implement Consistency in any data structures.

- ***Isolation*** is essential for file system transactions. Some implementations block accesses to files that are waiting to be updated, guaranteeing that transactions execute serially and do not conflict. Other transactional approaches allow a process to read values changed by another running operation, but implement cascading rollbacks whenever necessary.

- ***Durability*** is also problematic for file system transactions. A transaction that reaches its overcome is durable since it is written totally in persistent storage. However, implementations like Soft Updates and asynchronous metadata-only journaling provide a fast consistency model, but open the possibility for data loss, not guaranteeing Durability. These implementations can achieve durability through system calls (*fsync*) but this property is not a requirement of the underlying model.

These properties of the model fit for the objectives of keeping the system consistent but may not fit in the objectives of a fully-transactional file system. For a file system that exports a transactional API to the programmer, it is important to support all the four properties of the ACID acronym. In order to achieve this, a lot of transactional code is necessarily added to the transactional engine implementation, since the model is rather different from the underlying one.

The ACID acronym for a file system with such semantics has a different perspective:

- ***Atomicity*** is strict but defined by the programmer, since the length of a group of operations is defined in the application code. The IO operations group, wrapped around by an atomic block for example, must implement an all-or-nothing semantics to the whole operation set, just like each operation behaves individually with the file system. A system with journaling or even with a Soft Updates implementation is not enough; an atomic block can perform a lot of successful IO operations but the system cannot forget the changed data, because the atomic block can be interrupted and therefore aborted, requiring that all the IO operations previously written to disk are rolled back.

- ***Consistency*** property is also required, but an internally transactional file system already satisfies this property. The platform that needs to remain consistent is the file system architecture.

- ***Isolation*** is also important, allowing concurrent execution of transactions to operate correctly. However, isolation is once more a particular property that can be implemented in different shapes. For example, a transactional model could allow the read of uncommitted data to other transaction, but it would have to make the second transaction dependent from the outcome of the first one. This takes us back to the study of the isolation levels presented before, known for defining an operational balance between concurrency and coherence.

- ***Durability*** fits in this transactional model, but it was not implemented by the underlying consistency model. However, it is of great value to the API abstraction. Thankfully, durability can be achieved with relative ease by performing fsync calls at the end of transactional code blocks, forcing the system to actually flush its IO buffers and perform the corresponding algorithms, and only then return and commit the transaction.

Several extensions to the standard ACID model defined above are also possible and useful. Nested transactions, for example, are of great interest to most transactional programming abstractions. These extensions should be implemented in an external transactional engine, in order to keep the underlying file system engine simpler and efficient.

3.3.2 Specifications for a Transactional Filesystem API

The task of defining an API interface for a transactional file system can become quite complex. In this section we intent to define a starting point to the design of such a particular API.

First, existing applications should need no changes to benefit from the file system consistency in individual IO operations, but the capability of defining multiple IO operations for a single transaction must be explicitly programmed at application design-time. The new transactional API must give shape to the transactional model defining each property as strictly as possible, in order to simplify the transactional code and therefore the implementation. Since the transactional abstraction for IO operations is powerful and minimizes the difficulty of writing stable and consistent programs, providing a complex transactional API would be of no interest, making implementation a lot more difficult for both the API programmer and the application programmer.

One important consideration to take before designing the API is to avoid the change of any existing system calls, maintaining backwards compatibility. Adding a transactional argument to every IO system call is obviously unacceptable. The API should provide a *start_transaction* call which associates the current process or thread with a new transactional scope, ended by a *commit* or *abort* calls also provided by the API. If the process or thread terminates without effectively calling commit, the system should rollback the entire transaction. The API should then define a considerable number of transactional operations establishing a parallelism to the corresponding operations in standard IO interfaces. For example the *fopen* call must be available in the API in a transactional version with an intuitive but different name ("*trans_fopen*" or simply "*tx_fopen*"). In order to use these new calls and guarantee that they will become effective, they need to be used between a *start_transaction* and a *commit_transaction*, and the later must succeed.

Extensions to the transactional model like nested or long-running transactions must also be predicted by the API designer, whether allowing them or not. By providing more extensions to the standard model, the API becomes more flexible and useful. For example, focusing on the creation and use of transactional scopes, it would be useful to improve the notion of associating a transaction with a process. The *Amino* API assigns a unique identifier to a transaction, allowing that another process takes control of a running transaction by requesting association with that identifier [20]. This extension is useful for distributed transactions and for a lot of multi-threaded scenarios where concurrency is essential. Model extensions should be carefully chosen and implemented, maintaining the ease of use of the API while improving its flexibility.

In fact, even if the API designer chooses not to implement special extensions to extend the model, the simple design of an API that enables the creation of transactional scopes and the use of transactional versions of the IO primitives inside that scope, allows the application programmer to take advantage of all of the ACID properties in a simple way, having all the benefits of predictable semantics.

4. Possible TFS Approaches

The choice for the best approach to achieve a fully transactional file system is not simple. Focusing on our prototype implementation, several specifications must be defined and facts be evaluated. In this chapter we will focus on some of the possible approaches to our prototype, and study their advantages and disadvantages.

As stated previously, the file system should not only be internally transactional but should also provide a transactional model to user applications by offering a new file IO API. These API specifications have been introduced and studied before. The study of the possible integration techniques between the API and the underlying system is the main objective of this chapter.

4.1 A completely new File System

Probably the most vital choice for our prototype implementation is whether to use or not an existing file system. If we consider the option of implementing a completely new file system we would probably get optimized results while still obeying our objectives. This would be the bottom-most approach, since the prototype would have to start from the block-device level code and grow up to the Kernel's file system IO API needs. The file system consistency technique implemented would not only guarantee atomicity for primitive IO operations (like *journaling* guarantees for *write* call) but would also fulfill the needs of our external transactional model. This approach should therefore be a file system designed from scratch and totally independent from other file systems, probably overcoming any other possible implementation approaches when comparing benchmarks between them. Since it would not be an altered project but a completely new design to meet our objectives, the file system itself could achieve worse results when compared to other file systems using simple benchmarks, but it would support application-level transactions in a native mode and probably be very efficient at it. Existing file system projects are optimized to their primary objectives (mainly consistency, throughput and fewer architectural limitations) and that is why changing crucial project sections, like the consistency technique for example, would probably degrade throughput.

As we think about the possibilities for developing an entirely new file system for *Linux*, we started by studying the *Linux Virtual File System* (VFS). Back in 1994 when *Linus Torvalds* released version 0.95 of the Linux operating system, a new *kernel layer* was added to the project: the *Linux Virtual File System* (Diagram 3[2]).

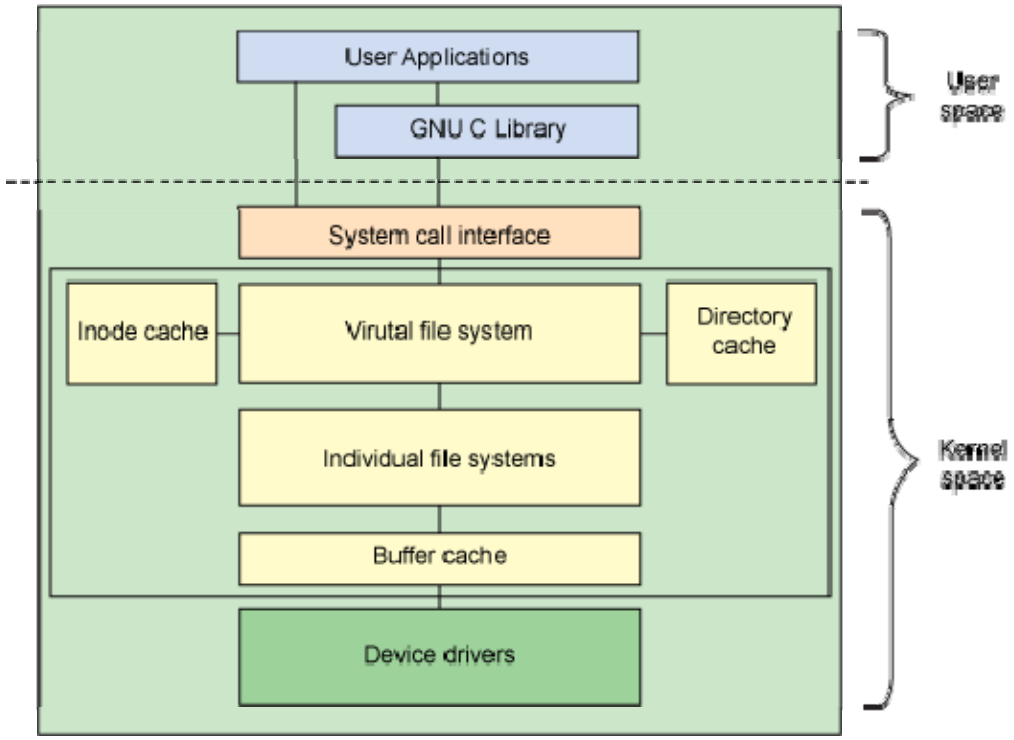

Diagram 3 - The Linux Virtual File System (VFS)

This layer offered a programming interface that allowed developers to write modules for the Linux Kernel which represent support for new file systems. Other projects began to develop around VFS and today it is even possible to add a file system to the Linux operating system that interacts with the kernel in user mode only, giving a great flexibility to new file systems, and increasing the security in the core design of Linux. The most famous and well-succeeded projects that allow this are *Linux Userland File System* (LUFS), and the superiorly supported *Filesystem in Userspace* (FUSE). As a result of the cleanness design of VFS inherited from *Torvalds*, this Kernel API allowed a whole new perspective for file system developers; different kinds of file systems arose easily around the World.

[2] Image taken from the IBM website at *www.ibm.com/developerworks/linux/library/l-linux-filesystem/*.

With VFS, the kernel does most of the hard-work while the file system specific tasks are delegated to the individual file systems through a special technique that will be succinctly explained. When a *block device* based file system is mounted, the VFS layer must read its *superblock*. Each file system has its own superblock read routine which must calculate the file system topology and map that information into a VFS superblock data structure which is then kept by VFS for each mounted file system. These VFS superblocks contain not only information about the file system, but also pointers to each file system specific functions. When the kernel needs to process an IO call to a VFS file system, instead of directly calling each function of the file system, the kernel uses the "*Operation Tables"* (sets of function pointers) to find the correct function for each operation, enabling different file systems to have their own handlers, that is, their own structural mechanisms. After defining the VFS required data structures and entry points defined in the C header file "*include/linux/fs.h*" (included in the Linux kernel source-code), the new file system should then be ready to become mounted. This would allow the standard, non-transactional IO accesses to our file system to be executed using the standard IO functions, since the system now fully-recognizes our file system as a standard one.

Even though our new transactional file system approach would be architecturally different, several standard operations related to the superblock, inodes and files have to be implemented in order to support this generic kernel IO access. This layer has the excellent support for defining operation tables which enables flexibility in designing standard file systems; however, it is still impractical to add new operations to the standard file system basis.

Regarding the transactional support we want for our file system, as we stated in section "API specifications", each IO operation belonging to a transactional context needs to have a similar version that includes the transaction identifier, and primitives to *start*, *commit* and *abort* transactions, so that the transactional manager can evaluate and perform the correct operations. Our new file system could then operate gracefully by interacting with the system using only VFS, but in order to export a transactional model, an additional API library and a running transactional manager must probably be implemented. This manager should then be designed to blend and operate with both the special IO API and the standard system IO functions. It also has several other important tasks, from controlling the running transactions and handling their concurrency, to guaranteeing that the model properties are applied in any situation. Since we would design the file system from scratch, the transactional capabilities should be made available

only by using the API, but the mechanisms that make this possible should start from the file system data structures design, enabling optimized levels that are impossible in any other approach.

Having this stated, this approach would start by studying and specifying possible data structures for a file system that fits standard and transactional file IO. After this design and its proper evaluation, we would start to study the support of generic IO operations required by VFS in order to allow the file system to be mounted and used by the standard IO system calls. Finally, a design of the transactional engine would take place, enabling special IO functions to be used inside transactional blocks by interacting with the file system using the standard IO API.

In a preliminary remark, this approach would have a major number of advantages, but like stated at the section introduction, it would require a huge amount of time and workload not only for investigation and modeling, but also for the hard implementation task.

4.2 Using an existing File System

Implementing a new file system would not only demand time but it would also require us to go through all the hard work of investigation and evolution already done by other existing file system projects. In an *open source* environment, it is generally better to logically regroup the subparts of an entire project in such a way that the project becomes a group of operating modules, where each has its own well defined tasks. This way (and since we are talking of a *free software* scenario) the project can use modules from other existing projects and spare the overall development (and debugging) time demands. This time and effort can then be spent on the specific and unique modules of the project.

Using an existing stable and mature file system would guarantee us a trustable working base. We have chosen to study two of the most popular file systems; the *Extended File System* version 2 and 3 [22], and the *Reiser File System* version 4 [23]. Our choice for these two projects derives from their technical specifications, which we weight and compare further ahead.

A true candidate for this approach is *Stephen Tweedie*'s famous *Extended File System* (*ext* version 3, or even the beta version 4). Ext3 is known for its stability and overall throughput, which would offer us a great starting point. Also, it is a widely-spread file system and would probably make our approach more well-known and more tested. The project evolution from version 1 to version 2 introduced the major parts of the file systems functionalities. Later, ext3

was created over ext2 only to add *journaling* to the file system and enhance some performance related issues. It is fair to say that ext has reached such success that it is being ported to many different operating systems, like the *GNU Hurd*, the *LITES* server and the *Masix* OS.

After exploring the architectural design of the Ext2FS [22], we will briefly explain some technical details that we find very interesting in the later versions of ExtFS:

- ✓ ***Extendable and evolutionary design –*** The file system was designed with its evolution in mind, containing hooks that can be used to add new features. Many people is working on extensions to ext3, enabling features like on-the-fly compression, undelete, and *Posix* ACL semantics.

- ✓ ***BSD or System V r4 semantics can be selected at mount time*** – Allows the file system owner to select the desired semantics for file creation and *groupid* inheritance.

- ✓ ***BSD-like synchronous metadata updates*** – A mount option allows the system owner to request that metadata is written synchronously on the disk when modified, maintaining a strict consistency policy.

- ✓ ***Configurable logical block size*** – Block size choice is a balance between performance and space loss. Larger block sizes reduce block seek overhead and improve the performance of sequential reads, but also increases the minimum size occupied be every single file on the disk.

- ✓ ***Fast symbolic links*** – Symbolic links do not use any data block on the system. The target names are located on the inode itself, saving space and speeding the link usage since less block reads are necessary. Of course, the inode space is limited, so not every symbolic link may become a *fast* symbolic link. The maximum target name width for a fast symbolic link is 60 characters in Ext v3.

- ✓ ***File system state tracking*** – By keeping a flag indicating whether the file system is "CLEAN", "DIRTY" or "ERRONEOUS", the kernel always knows when a *flush* (*bdflush*) or *fsck* are needed.

- ✓ ***FSCKs at regular intervals*** – A few superblock variables count the number of days passed and the number of mount times since the last fsck, to force the system to fsck the partition in order to avoid danger situations. Regular and unaware users always tend to skip the file system checks, leading to disk inconsistency in their own systems.

- ✓ ***Secure deletion of files*** – A file structure special field indicates that when the file needs to be deleted, the data blocks belonging to the file must be refilled with random data. This protects the *undeletion* of the old file contents.

- ✓ ***Immutable and append-only files*** – Low-level read-only file protection is useful for protecting sensitive OS configuration files. Also, low-level append-only protection is useful for many situations, like for example, security log files which are only supposed to grow.

- ✓ ***Physical "block groups"*** – The architectural design of the Extended File System begins by defining that the partition is made by one leading super-block and a set of *block groups*, analogous to the BSD FFS's cylinder groups. However, since modern disk drives are optimized for sequential accesses and tend to hide their physical geometry, block groups are not tied together physically. This design also implements a *sparse superblock* feature, having all the advantages previously stated for FFS by keeping local copies of the partition's superblock.

- ✓ ***Kernel optimizations taken successfully*** – Improving IO throughput, Ext takes advantage of the buffers inside the optimized versions of VFS. Ext performs *read-aheads* for directories and files, taking advantage of the sequential read speeds and the principle of locality. Also, the kernel optimizations always try to allocate related inodes and data together on the same block group. When increasing a file's size, Ext searches the free inode blocks bitmap in the file's block group for an available byte (8 adjacent bits), succeeding in preventing fragmentation with a hit-rate around 75% even on very full file systems.

Another elected file system to use as a starting base is "*reiser4"*, the evolution of *Hans Reiser*'s "*reiserFS"*. The Reiser file system uses a variant of the classical balanced trees

algorithms, resulting in more efficiency for small files (less than 4 kilobytes) and large files (larger than the total data blocks of an ext block group) when compared to the Extended File System. Block allocation algorithms of ReiserFS reach their best advantages when file sizes are near 100 bytes. When benchmarking the file system (obviously depending on the operating system, its buffering policy and the file system load) Reiser4 reaches throughputs of 4 times larger than the Ext3 ones. Also, the project has an excellent development team (which we would like to thank for the sympathy and availability) who has done a great job on designing modularity and enabling the file system to behave according to administrators desired policy; that is, the active *plug-ins*.

The project also has the special capability of offering two kinds of libraries for plug-in support. The first library is the *per-Superblock Plugins Library* (SPL) which aggregates plug-ins that work with low-level disk layouts (Superblocks, formatted nodes, bitmap nodes, journal, etc). For example, the partition format in reiser4 is a disk format plug-in, assigned to the partition's superblock. The second library available is the *per-File Plugins Library* (FPL), which aggregates the so called file managers which work with disk layouts (like item plug-ins), representing some formatting policy (like the format plug-in). The *topmost* plugins of file types are to service the VFS entry points in the Operation Tables. These *file managers* are pointed by the inode's plug-in table attributes.

This plug-in support of reiser4 would allow our prototype to install itself in a specific *algorithm* of the file system, allowing it to perform the operations required by the external transactional model. This is a valuable advantage since we think that it would simplify the hard task of changing the vital parts of an existing project without damaging it.

Unfortunately, reiser4 also has some problems of its own.

Even though stability has been mainly improved in the latest version (v4), and both the authors and the users claim that the system is now robust, compatibility keeps giving some problems in some unexpected situations, like in the later Linux kernel versions, NFS versions, and even user interface applications like KDE4 and Sun's Netbeans 6. Obviously reiser4 is a great project and compatibility is a time demanding task that the team will surely take care of. Even so, probably the major problem of reiser4 is now the team situation, as the *Namesys Corporation* is currently closed, and its founder *Hans Reiser,* is currently imprisoned.

The project is now depending on the great effort of only a few developers. Under a new leadership conducted by *Edward Shishkin,* other great programmers that previously belonged to *Namesys* keep their good faith and have never abandoned the project. This is obviously an unfortunate situation that we must ponder in our choice, since it affects not only our support possibilities, but also has its own weight in the lifetime probabilities of our project.

In an attempt to measure the stated, we can affirm that the approach of using one of these two file systems to support transactional operations is reasonable and would probably even achieve remarkable benchmarks. However, the difficulty of correctly changing vital parts of an existing complex project, coupled with the need for re-adapting our code each time the project evolves to new versions, makes this approach hard to thrive, since in the event of a crucial change in the original project, our code would certainly no longer fit.

Even so, this is still an inspiring approach that fits our objectives and therefore deserves our total consideration.

4.3 File System Independency

Last but not least, other types of approaches are possible to achieve our objectives; a solution that is independent from the underlying file system would be more compatible and *elegant*.

The first method our team suggested in order to achieve this solution is by caching the application's transactional IO requests in the central RAM memory, before actually writing to the file system. From this perspective, applications that use our prototype, perform transactional writes and then abort, would not even change the file system's structure. If otherwise the application tried to commit a transaction, the prototype would have to perform all the real IO operations before allowing the transaction to reach its outcome. For committing a transaction, a special technique should be studied to guarantee the *Atomicity* property of the external model: if a crash happens in the middle of the commit operation (where the prototype uses standard IO to commit changes), some operations might have reached persistent storage while others did not, not even guaranteeing their ordering because of kernel's *write caching*. After the system reboot, the underlying file system could find some visible effects of the uncommitted transaction (standard atomic IO operations that have reached the physical layer) and the new transactional file system should then rollback the visible changes in order to guarantee the Atomicity property

of our API. The prototype should therefore have its own *journaling technique* in order to guarantee the transactional model properties, or it should control the VFS view of the *dirty data blocks*, making the system undo the changes at the mount time of the underlying file system. Other problems also arise from the fact of having IO changes held in central memory: imagine a transaction that tries to copy a file to another, whose length is larger than the free RAM memory.

This approach would have the great advantage of not requiring a disk partition in a specific format, fitting whichever standard file system the user has chosen. A considerable disadvantage however is that several further techniques and methods must be studied carefully before this approach is a possible and reasonable choice. The complexity of designing a middle-layer between the transactional API and the Kernel's VFS API is not simple, but it is an advantageous possibility.

4.4 Approach Evaluation

We have analyzed the main differences between the various possible prototype implementations, but they become sparse arguments if we do not analyze and weight them in our current scenario. We must evaluate the several approaches not only by their expected results but also by the time we have left to implement our prototype and the number of the elements of the programming team.

In a measurable way of comparing our stated arguments, we decided to make a table that classifies, from zero to five, the pros and cons of each approach. The resulting table (Table 2) is shown below.

Approach\Specs	Simplicity of Design	Expected Throughput	Compatibility and Independency	Adequate to Schedule	*Total*
New FS	0	5	2	1	**8**
Ext FS	2	4	3	2	**11**
Reiser v4	3	4	3	3	**13**
Mid-Layer	4	3	5	4	**16**

Table 2 - Evaluation of Stated Approaches (More is Better)

The above results are an estimated scalar value that represents the weight of the arguments we have previously stated.

After analyzing the results presented in Table 2, we decided to look a little further into the architectural and functional specifications for an intermediate layer that allowed us independency from the underlying file system.

Having our path now defined, the next chapter will introduce our architectural approach for achieving the Prototype.

5. TFS Architecture and Development

This section will describe and specify each technical detail in the implementation of the prototype, starting from the abstract architecture to the deepest details of internal operations.

In the task of creating an intermediate transactional file system layer, we specified an abstract architecture for the implementation, represented in diagram 4.

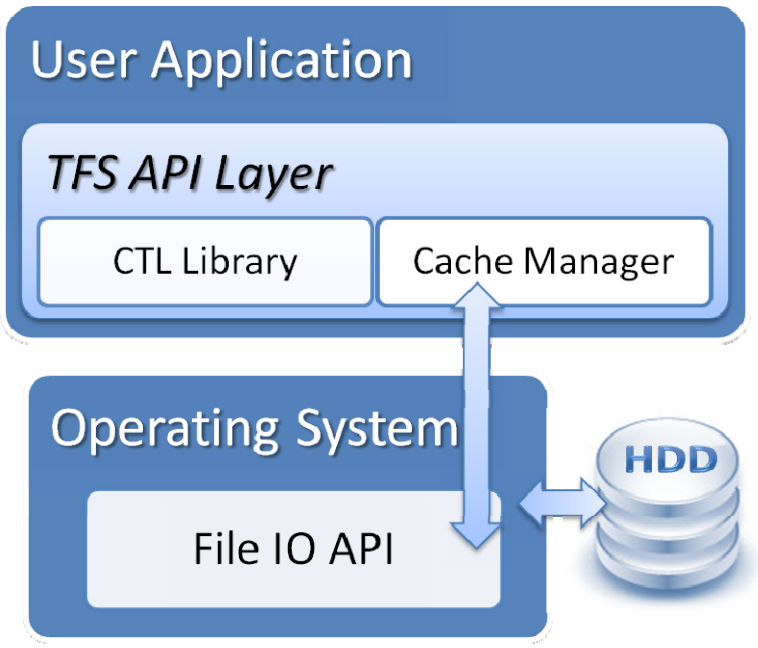

Diagram 4 - Overall Architecture

The prototype will consist in a well-documented API that has a built-in transactional engine, and is able to do the required work needed to emulate transactional accesses to regular files for a single application.

The application that uses our library must be programmed and compiled with our API, resulting in a binary application that accesses the file system transactionally. Also, we are aware that this approach also has some downsides, like the individuality of the transactional engine; that is, two applications that were compiled with our API and are simultaneously running do not obey the transactional semantics between them, if they operate over the same file. Since each application is compiled with our library built-in, the first one that requests a file will probably

win the race for achieving an exclusive lock for that file and the second application will fail on opening the file. However the prototype will still never run into an inconsistent state and this limitation will not occur if the two applications don't request access to the same files.

Our architecture will then use the standard file system IO primitives available in the operating system, allowing a transactional access for files of different underlying file system types. Our API will represent a substitute for the standard "*stdio.h*" calls to operate with files, and each of these operations should be as similar as possible to the original ones.

We will explain below how the prototype was achieved by specifying each internal component, following by an explanation of the resulting features, and even a study of the resulting semantics related to the transactional ACID properties.

By the end of this chapter, the reader should clearly understand not only the components that made this approach possible, but also its resulting potential, being capable to recognize possible use-scenarios of the prototype.

5.1 Detailed Architecture

This section will introduce the internal architectural components and explain briefly their tasks and responsibilities. After explaining which parts our prototype is made of, we will have some specific sections for the main components, where we will detail their semantics and implementation.

Starting with an insight of the main components of the prototype, there are basically two modules that provide the required mechanics, as represented in Diagram 5.

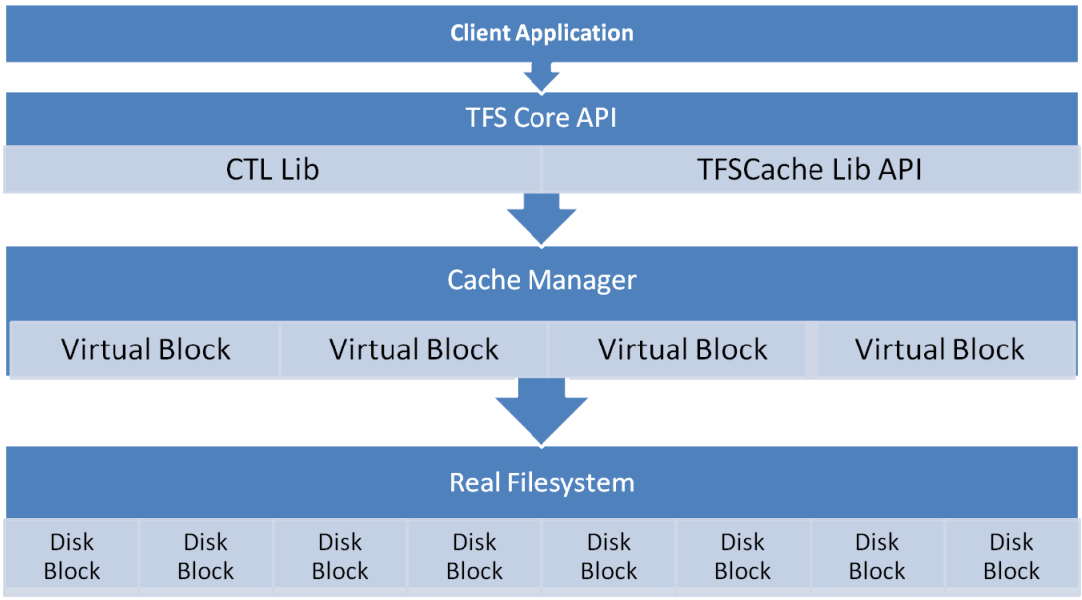

Diagram 5 - Internal Prototype Components

The first module contains our external API and is responsible for the transactional management and control of the requests that the user-application does. This module is named the "*TFS Core API*" (or just "*TFS API*"), and it is available by including the C header file named "*TFS_API.h*". This is the only include that the user should do to benefit from the whole prototype functionalities, and the API documentation is available in the Annex A – API. In order for this module to achieve its tasks, it will include the CTL library [4] for the transactional functionality: collision detection and consistency verifications. Also, this main module requires the mapping of the disk files as special, virtual blocks. This mapping is done by our secondary module, the TFS Cache Manager (also specified in Annex A – API).

The Cache Manager provides primitive operations to access groups of data from inside files; that is, virtual blocks that represent sets of one or plus real file system blocks. These virtual blocks are cached by the cache manager, which also provides primitives to rollback these blocks to their in-disk versions, or to synchronize their in-memory versions with the corresponding disk offsets. This allows the prototype to provide concurrency support for transactional file access.

The Diagram 6 presented below shows three distinct synchronization levels of the architecture. Starting from the bottom, the prototype will request exclusivity for the files opened, keeping a single file handler with write permissions for each file. The layer represented in the middle is the Cache Manager, which defines a file as a set of virtual blocks and provides basic

functional primitives for manipulating them. The Cache Manager uses *mutex* locks to fully-synchronize all block-operations. However, allowing concurrent accesses to parts of opened files was a specification that we wanted to achieve, and so the top-layer allows multi-threaded operations. The automatic synchronization of the client application threads allows schedules for file transactions that are not necessarily serialized. These capabilities rely on CTL for consistency verification.

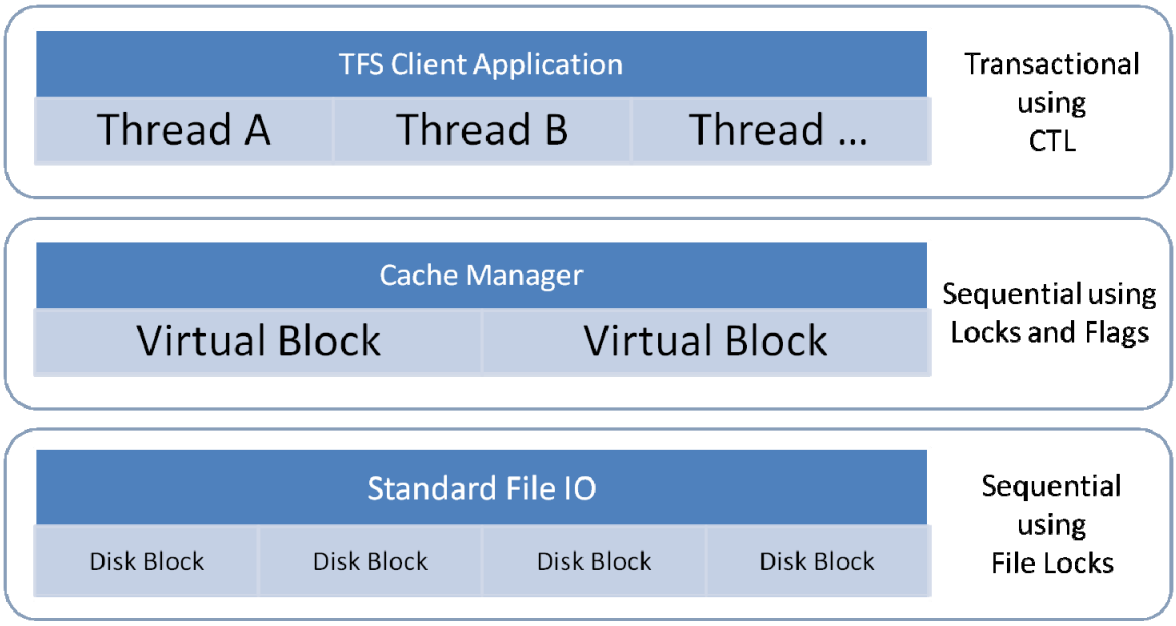

Diagram 6 - Concurrency Architectural Support

Having stated the main architectural specifications of the TFS prototype, the two main modules (the TFS API module and the Cache Manager) will have their own sections below, where we will explain their implementation details and functionalities.

5.2 The Cache Manager

The Cache Manager module has already been referred to as the virtual-block manager. We needed to modularize the entire prototype, and so we decided to isolate all the file IO operations and file caching inside a single, separately available module. This "Cache Manager" module has an API that can be used for other applications, as long as they require reading or writing files, and has the major benefit of caching file contents whenever the user reads or writes them. The central data structures inside Cache Manager that provide this functionality are mainly three: a

Hashtable, an *Opened File* structure definition, and a S*ingly-Linked-List*. The Diagram 7 below shows the structural design.

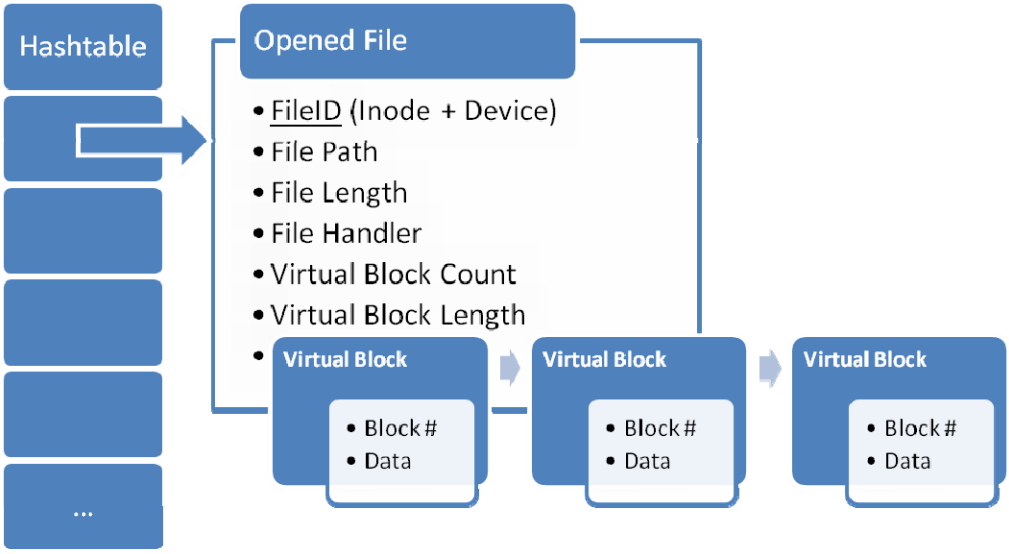

Diagram 7 - The Cache Manager Data Structures

The TFS Cache Manager contains a *Hashtable* of opened files, where the unique identifier for the file is not the path (because of the possible links to the same file), but instead the structure we call *fileUniqueID;* a pair of an integer representing a *deviceId* and a long representing the *inode* where the file starts. Both of these values are obtained by the path specified at the open call. For each file present in the Hashtable, the prototype keeps a singly-linked-list of the virtual blocks opened for the file. Other fields enhance the capabilities of an opened file in cache, since they provide useful information: the number of virtual blocks, the block's length, and even the operating system file handler, which Cache uses to access the file exclusively.

Apart from the modular approach and virtual-block caching feature, the virtual-blocks also define the *Granularity* of our file transactions.

The Diagram 8 below shows a file opened in Cache Manager and the granularity of a transactional object. Each virtual-block represented is accessible transactionally, but only by a call to the Cache Manager. This call returns the memory containing the block, and so the TFS layer is able to *log* the accesses to that block number in the current transaction read/write sets, using CTL.

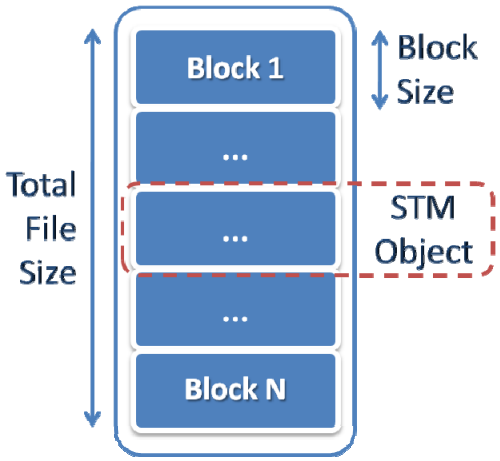

Diagram 8 - Virtual Blocks and CTL

This means that, for the example given in Diagram 8 which has *n* virtual-blocks for a file, our prototype allows *n* concurrent transactions consistently changing the file, as long as each transaction operates on a different virtual block.

The *length* or *weight* of the transactional grain is therefore inversely proportional to the number of virtual blocks of a file. If the file had only one block, then the resulting schedules for transactions that operate over the file would be serialized; that is, no concurrency would be supported at all. The number of virtual blocks has a defined top-limit, being an integer number defined in the Cache Manager Header file, which can be changed without compromising the prototype. However, specifying an extremely large number of virtual blocks per file has the downside of producing large transactional read/write sets when fully reading or writing a file, which could raise a problem inside the transactional manager itself. In our specific implementation, the CTL STM Engine does not yet support unlimited read/write sets, so our prototype would not be stable and robust if we removed the maximum virtual blocks per file, since the application could easily crash if, for example, accessed twenty files within the same transactional scope. Also, an important reason for this coarser granularity is the previously explained problems of fine granularity transactional engines in the section 2.4.1.4 (*Defining Transaction Granularity*). A fine granularity requires more computation and more waste of space by the transactional engine.

The virtual-blocks specifications presented above have special rules and behavior when a transaction is growing a file in memory. The event of a transactional file growth is special and

problematic, so we decided to make a single sub-chapter explaining the semantics for growing files, later in the chapter "*Major Features"*.

Another very useful feature of the Cache Manager is that it implements on-disk *journaling* of the virtual-block write calls. Again, this technique proved very important for our prototype, and will be clearly specified later in the features chapter.

Concluding this section about the Cache Manager module, we now present a table briefly describing the main Cache API calls.

Functions	Description
Cache_Fopen	Opens an existing file or creates it, returning a fileUniqueID
Cache_Fgrow	Grows the specified file to a new length
Cache_RefreshBlock	Rollback the contents of a virtual block, fetching it from the OS
Cache_SyncBlock	Commits the changes to a virtual block, writing it to OS file
Cache_Fclose	Closes a previously opened file
Cache_GetBlockAddress	Gets a pointer to a specific virtual block structure in memory
Cache_GetBlockData	Gets a pointer to the *data* of a virtual block structure
Cache_StableState	Ask Cache Manager to *synch* its writes to Journal file

Table 3 - Main Cache Manager API Calls

5.3 The TFS Transactional Layer

For any application that wants to make use of TFS, the only module required to be included is the *TFS API*. By including the file "*TFS_API.h*", the application can then use the transactional API for doing transactional file accesses, knowing almost nothing about the TFS internals (the *black box* analogy for libraries). As referred before, the API aims to emulate the standard file system IO calls available in the Linux operating system, offering a vast amount of primitives suffixed by their original name, with the hope to help the programmer of an existing application to easily benefit from transactional semantics by simply wrapping the file IO calls around a transactional *start* and *commit* calls.

The TFS module operates with CTL and the Cache Manager (as shown before in Diagram 5) in order to achieve the desired semantic for file operations. The CTL STM engine will be responsible for creating transactions and constantly verifying their consistency. Also, we took good use of the *Handler System* to trigger important actions in specific cases.

The connection between files and the STM engine was not trivial. The Cache Manager defined the virtual blocks as a very simple structure made of:

```
typedef struct cacheBlockElem {
        int             BLOCK_NO;
        void *          DATA;
}cacheBlockElem, *cacheBlockElemPtr;
```

Code section 5 – Cache Virtual block Structure

Knowing this, TFS does the *synchronization* between a running CTL transaction and a virtual block by using the *BLOCK_NO* variable in a transactional STM context. This makes the consistency check verifications a responsibility of the CTL STM engine, since when a transaction needs to read one block *DATA*, it needs to be able to read the *BLOCK_NO* variable transactionally through CTL. More details of the mechanism that TFS uses for reading and writing files transactionally will be available in the TFS Call Mechanics section (6.3.2), in the *TfsFread* and *TfsFwrite* calls.

Before explaining the algorithms of the TFS API core module, it is imperative to look at what data structures TFS has, presented in the diagram 9.

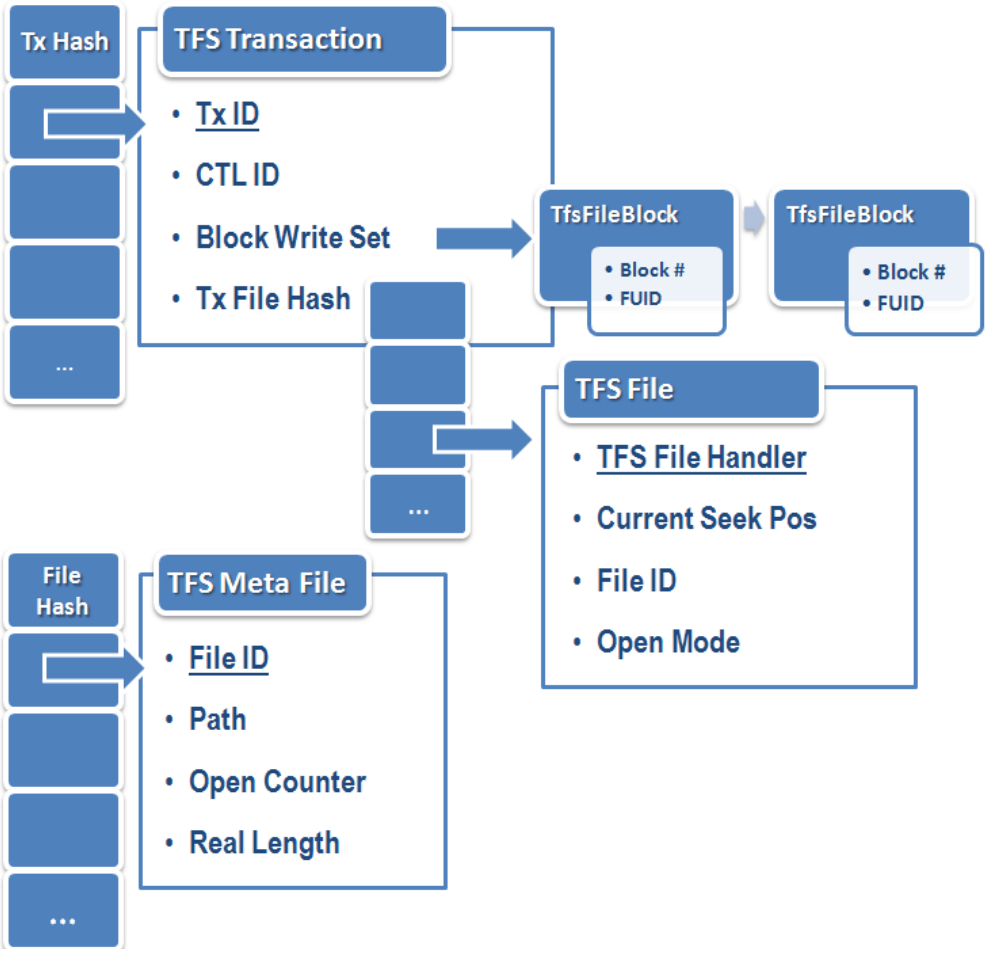

Diagram 9 - The TFS Data Structures

The above diagram contains all the data structures used in the top-most layer of the prototype: the TFS module. To elucidate the diagram, we can say that this module has two main structures; one is the *Transactional Hashtable* that takes care of running transactions and their operations over files, and the other is the *File Hashtable* which contains the *Metadata* of each used file. Because of the intent to modularize and organize our prototype, notice that the TFS API File Handlers are kept top-most at the transactional level, and the more concrete identifiers (like the File ID, real length and open counters) are logically grouped at the bottom in the metadata hash, directly related with the Cache Manager and therefore the real files.

Starting from the Transactional Hashtable, it stores important information related to a running transaction:

- ***Tx ID*** – A unique integer identifier that the transaction obtains when it starts.

- ***CTL ID*** – The private CTL identifier for the current transaction.
- ***Block Write Set*** – A singly-linked-list that keeps reference of which file virtual blocks the current transaction has changed.
- ***Tx File Hash*** – A Hashtable that keeps structures representing the files opened by the current transaction. These structures store the TFS *File Handler* (a unique integer identifying the file handle), the current *fseek* position of the file handler, the *fileUniqueID*, and the open mode of the file (explained below in the *TfsFopen* section).

Apart from the transactional Hashtable, the other Hashtable is the *Metadata Hash*. This structure aims to keep a map with information of the operating system files. For example, when a file grows, the metadata hash will only reflect the new length *after* the transaction reaches commit and the OS file is changed. Otherwise, isolation would not succeed.

As a result of this structural approach, the TFS module is able to integrate the two transactional components (CTL + the transactional hash) and the two real components (metadata hash + the Cache Manager). The algorithms that work with these structures are implemented in each function of the API, which we explain in the *TFS Call Mechanics* section (6.3.2).

5.3.1 CTL Configured Mode

We have stated that CTL is highly configurable. For the needs of our prototype, CTL was compiled with a special configuration, which we show below:

Setting	Chosen Value
***Encounter* or *Commit* mode**	**Encounter** mode
32 or 64 bits	**32** bits
***Object* or *Word* mode**	**Word** mode
Handler System	**Yes**

Table 4 - CTL Settings for TFS

The only setting that we could have chosen differently was the architecture. The 32bit architecture decision was made only to improve compatibility, since a 32bits code should run on a 32 or 64bits machine.

We had explained above that TFS does the synchronization point between the CTL transactions and the virtual blocks using the *BLOCK_NO* variable inside the virtual block structure. The variable is an integer value, which is the same byte-size as a Word. Therefore, in order to perform a CTL *TxLoad* or *TxStore* over the integer value, either we used *Object* mode with an *object length* of 4bytes and had to deal with the consistency checks, or we would simply use *Word* mode and have the pleasant guarantee that the CTL transactions are always consistent.

Regarding the log-replaying modes of CTL, if we used the *CMT* (or *commit*) mode, the read and write sets of a transaction would only be verified at the commit time, and the contents of the virtual blocks (or even their length) could have been changed and would have to be rolled-back. By using CTL in the *Encounter* mode, the read/write sets are checked at the moment of accessing the block, and in case of a collision the transaction is instantly aborted and the block access is denied.

The Handler System was also important to TFS, since it was the simplest way to implement compensating actions for certain operations. We had given the example of a *pre-abort handler* that frees a variable that was allocated inside a transaction, but in the specific case of our TFS algorithms, we used the handlers for many situations, like calling *Cache_SynchBlock* for the *BlockWriteSet* of a transaction at its *pos-commit* state, or calling *Cache_RefreshBlock* for the same *BlockWriteSet* if the transaction enters the *pré-abort* phase.

5.3.2 TFS Call Mechanics

This chapter is dedicated to clarify the use of our data structures for each specific operation. We will explain the algorithmic effects of starting and ending transactions, and of course, performing our transactional operations over files, derived from the external use of the API. This chapter also has the objective of completing the API documentation with a more technical explanation on how TFS works.

5.3.2.1 TfsStart

```
TFSThread    TfsStart                                                    ();
```

The *TfsStart* call is imperative to any transaction, since it updates the TFS structures inserting a new transaction, it also starts a new CTL Thread and a CTL Transaction. It not only marks the location to where CTL will push the transaction after an abort-and-retry event, but also returns to the user application a *unique transaction identifier*, used as a transaction handler to the other TFS API calls.

Since the *TxStart* CTL call only keeps a pointer to the stack frame where the call was made, the function scope where the TfsStart call was made cannot terminate; otherwise CTL will generate a *segmentation fault* if it retries the transaction. Because of this aspect, our TfsStart API function has to be a *C Macro Definition*; the code of the macro will then be inserted directly in the user application by the compiler pre-processor, and the function scope of the TxStart call will be the scope where the user does the "TfsStart()" call. Apart from getting a new CTL Thread and Transaction, and a new TFS ID, the TfsStart macro also instantiates and inserts a new transaction structure into the transactional Hashtable, filling the fields described in section 5.3 (*The TFS Transactional Layer*).

At this point, TFS uses the Handler System to attach a *pré-abort handler* to refresh blocks listed in the transaction *blockwriteset*, and another *pré-abort handler* to remove the transaction from the transactional Hashtable. Notice that we did not implement a pre or pos-commit handler because these states happen inside our *TfsCommit* function.

5.3.2.2 TfsFopen

```
TFSFILE      TfsFopen                (TFSThread Self, const char *path, const char *mode);
```

The *TfsFopen* call is the equivalent to the *fopen* standard system call. However it is necessary to start a transaction with *TfsStart* first, in order to pass the *transaction ID* handler to TfsFopen as an argument. By opening a file, TFS checks if the file is not already opened, and if so, TFS adds it to the metadata hash requesting its *fileUniqueID* from the *Cache_Fopen* command. Otherwise, it simply increments the *open counter* field of the file structure in the metadata hash. Having the meta-information consistent, TFS also creates a new and unique *file*

handler ID to be returned by the TfsFopen function, and appends the opened file to the list of opened files of the current transaction, and setting the current SEEK position to the start of the file. The TfsFopen function also receives a *path* to the desired file, and an *open mode*. Table 5 lists the valid *open modes* for TfsFopen.

Open Mode Flag	Description
R_FLAG	Opens for reading only, with shared access.
CRW_FLAG	Opens for reading and writing, with shared access.
RW_FLAG	Opens for reading and writing. Append requests exclusive access.

Table 5 - TfsFopen Open Modes

The *R_FLAG* marks the file for *read* only. The other two flags allow writing but their behavior differs when the *write call* attempts to grow the file, appending some data to it (that is, the current offset plus the write length are larger than the current file length).

The *CRW_FLAG* allows concurrency over the file at the same time that the *write call* extends the file length, while in the *RW_FLAG* mode the file is locked exclusively while the transaction does not terminate, but as an advantage of the RW_FLAG, the file can grow without any limits.

The reason for these two different semantics and for the limitation is inherited from two elements: the Cache Manager architecture, and the CTL read/write sets length limitation. The Cache Manager recognizes the files as sets of virtual blocks, which get mapped into the CTL transactions read/write sets when a transaction reads or writes a certain block. Since CTL has a limited number of entries for each set, the number of virtual blocks cannot augment freely.

Therefore, we specified two behaviors: either the file virtual blocks grow together (*RW* mode), maintaining the number of blocks (overcoming the CTL limitation); or the cache simply appends new blocks (*CRW* mode), and the user has the responsibility of not performing long transactions (by *long* we mean a big number of operations within the same *Start* and *Commit* calls).

Note that as a result of the RW mode block growth, each virtual block can inherit data from the block at its right side, meaning that if the file had been accessed concurrently, the other transactions might have inconsistent read/write sets. For example, a block that previously referred to 100bytes at offset 1000 when it was accessed now can refer to 150 bytes at offset

1500. As a solution for this inconsistency, when a transaction opens a file in RW mode and performs a write call that requires growing the file, the TFS system checks the file's open counter (in the *metadata* hash) and retries the transaction if there are other transactions operating over the file; otherwise, the file gets locked exclusively for the current transaction and is allowed to grow each block indefinitely.

TfsFopen function also adds a *pré-abort* handler to the current transaction, giving it the information needed to close the file. If the application does a correct TfsFclose of this file and later aborts, the *close file* handler checks the transaction list of open files and if it no longer exists, the handler simply returns.

Concluding this section about TfsFopen, the function either returns a valid unique file handler to be used with other TFS operations, or in case of failure returns one of the following flags, described below in Table 6.

TfsFopen Return Flags	Description
TFS_FILE_IS_LOCKED	The file is currently exclusively locked by another TFS transaction in *RW_MODE*
TFS_MODE_DENIED	You do not have the desired permissions in the real file system for the specified file and the specified open mode. Notice that if the file does not exist, TfsFopen automatically tries to create it. Therefore if TFS cannot create the file, this flag is also returned.

Table 6 - TfsFopen Return Flags

5.3.2.3 TfsFclose

```
int          TfsFclose                    (TFSThread Self, TFSFILE fp);
```

The *TfsFclose* call receives a TFS transaction ID and a TFS file handler ID. The function objective is simply closing the transaction handler for the file, and closing the metadata information if the file open counter is now zero.

The only other specification that the TfsFclose function has is for files being closed in RW mode. For these files, since even the blocks that have not been accessed by this transaction might have changed their length and contents, all the virtual blocks opened are added to the

transactions block write set. This guarantees consistency of the write and grow operations, since it will make the commit call synchronize all blocks together.

5.3.2.4 TfsFread

```
long        TfsFread    (TFSThread Self, void *ptr, int size, int nmemb, TFSFILE stream);
```

This call is the basic function to retrieve data from a file in a transactional context. All the TFS API functions that need to access data (like *TfsFscanf* for example) make internal use of TfsFread. The function is analogous to the *fread* standard call; it has similar arguments, but receiving an extra *transaction ID* (returned by *TfsStart*) and *transactional file handler* (returned by *TfsFopen*).

For reading a file, the function calculates the virtual blocks required to intersect the current seek position plus the desired read length. After having the list of required blocks, the function iterates over each one, requesting the block from Cache Manager, and performing a CTL *TxLoad* call to the *BLOCK_NO* field of the virtual block. This forces the CTL engine to check the read permission of the current block for this transaction, and if a collision is detected, the transaction is aborted and automatically retried. Otherwise, the function will keep iterating over the required virtual blocks until it has all the required data.

The function is also aware that if the file has been opened and grown in the *Concurrent RW* mode (CRW flag) it has an in-memory unlimited extension of data which we call *metablock extension*. This metablock temporarily keeps the file contents that go beyond the file length, and if the transaction successfully commits, the commit call adds the new virtual blocks to the file.

At the end, the function returns the total read length or returns EOF if at any time it runs outside the file contents, always considering the possible metablock extension.

5.3.2.5 TfsFwrite

```
long    TfsFwrite    (TFSThread Self, const void *ptr, int size, int nmemb, TFSFILE stream);
```

The TfsFwrite function is the only function to provide transactional writes to files. Again, all the TFS API functions that require writing (like *TfsFprintf* for example) make internal use of TfsFwrite. The function represents the TFS version of the standard *fwrite* call even maintaining

the same arguments, except for receiving the transaction identifier as an extra parameter and the file handler as a *TFS File Handler*.

Like TfsFread, this function also calculates and iterates over the required virtual blocks. However, the function is internally split in two different functions, in order to achieve the required semantics for the two different write-enabled open modes. As we stated before, if the file is in RW_MODE and the TfsFwrite call needs to grow the file, it makes sure that no other transaction is using it. Otherwise the transaction aborts and is retried. Contrarily, when growing a file in CRW_MODE, the function attempts to acquire a lock on the last virtual block of the file, and creates a private, in-memory, block of data (called *metablock*). This block will keep the file contents that overflow the file length, and it will be discarded at the *abort* event, or synchronized to new blocks in a *commit* event.

The return value of the function in case of success is the number of bytes written; otherwise, the function returns the TFS_MODE_DENIED flag, meaning that the current transaction is in read-only mode, or that the file is currently locked for growing in RW_MODE.

5.3.2.6 TfsFprintf

```
int          TfsFprintf          (TFSThread Self, TFSFILE stream, char *format, ...);
```

The *TfsFprintf* call is analogous to the *fprintf* standard call, which produces a file output according to a very flexible format. The function has variable argument length, since the programmer can *print* several variables inside the *format* string, or simply print text.

To achieve the desired similarity, TfsFprintf was defined as a macro, taking advantage of the "…" keyword for a macro argument. By defining for example "*function(arg1, A...)*" the macro associates to the letter "A" all the arguments except the first "arg1". Having *captured* the variable arguments, more specifically the format and any extra variables, we used the system call named *snprintf* which produces the same output of *fprintf* but directed to a memory location; not to a file handle. The memory location is filled by snprintf obeying the format specified, and its contents are finally dumped to the file using the normal TfsFwrite call with the appropriate parameters.

Even though this solution seems rock-solid, it offers a single limitation: the snprintf call takes a pointer to a pré-allocated memory area, which is TFS responsibility to make sure it has

enough length. Since TFS has not parsed the complex *format* and *variable arguments*, it cannot know what the final length result is. For that reason, we defined a constant at the top of the library header file, named *FPRINTF_MAX_LEN* which is the argument of the malloc function that fills the buffer. This buffer is dumped to the file, and is finally freed.

5.3.2.7 TfsFscanf

```
int          TfsFscanf          (TFSThread Self, TFSFILE stream, char *format, ...);
```

The *TfsFscanf* call simulates an *fscanf* call for a TFS file. Fscanf, like fprintf, has variable argument count and a flexible reading format, which made us define it as a *macro* just like we have done with TfsFprintf.

The function behaves much like the TfsFprintf macro. It uses the equivalent operation for a memory buffer (*sscanf*) but first fills it with contents from the file. Since one of the arguments specified is the max length to read, the macro calculates from the current seek position, which one is smaller; the max length to read or the bytes left in the file. The result of this calculation is the length to be read from the file to the buffer (using TfsFread), which is then processed by the *sscanf* function and returned to the user.

5.3.2.8 TfsFputc

```
int          TfsFputc           (TFSThread Self, int c, TFSFILE stream);
```

Similar to the *fputc* call, *TfsFputc* write a character (cast to an unsigned char) to the desired file. The function casts the *char* to an *unsigned char*, and then uses the *TfsFwrite* primitive with the required parameters.

The function returns either the integer value of the character that has been written, or returns EOF if the end of file is reached.

5.3.2.9 TfsFgetc

```
int          TfsFgetc                    (TFSThread Self, TFSFILE stream);
```

This function has the same design of the above, except that it reads an *unsigned char* from the specified file, at the current seek location. The return values are the same as in TfsFputc.

5.3.2.10 TfsFseek

```
int          TfsFseek          (TFSThread Self, TFSFILE stream, long offset, int whence);
```

The *TfsFseek* function sets the current cursor position of the specified transaction, for the specified file, to a new location. The *whence* argument (like in the *fseek* call) determines what to do with the new offset value.

There are three flags that can be specified, the table below clarifies them.

Whence Flag	Description	Example
SEEK_SET	The given offset is the new cursor position, starting from the beginning of the file	If the offset given is 10, the cursor moves to the 10th byte of the file
SEEK_CUR	The given offset is added to current offset	If the current offset is 30, and the given offset is 10, the cursor moves to byte 40.
SEEK_END	The given offset is the distance to the end of the file	If the argument is 1, the cursor moves to the last byte of the file.

Table 7 - The Whence Flags

5.3.2.11 TfsFrewind

```
int          TfsFrewind                  (TFSThread Self, TFSFILE stream);
```

TfsFrewind moves the file cursor position to the beginning of the file.

This function is the equivalent of performing a TfsFseek call with the offset argument as zero, and the whence flag set to SEEK_SET.

5.3.2.12 TfsFtell

```
long        TfsFtell                    (TFSThread Self, TFSFILE stream);
```

This function simply returns the current file cursors position. For example, if the file has just been opened, this call returns 0 since the file *seek* cursor is at the beginning. This call is equivalent to calling *TfsFseek(0, SEEK_CUR)*.

5.3.2.13 TfsCommit

```
int         TfsCommit                   (TFSThread Self);
```

A transaction that does not outcome on a *commit* call has the same effect as if it has never started. The task of the *TfsCommit* call is exactly to make effective all the actions performed by a specific transaction since it started.

The function starts by processing the transaction block write set and asking the Cache Manager to synchronize the blocks contents with the file in disk, overwriting the later. After asking for synchronization of the block list, the TfsCommit call also asks the Cache Manager to emerge to a stable state (using the *Cache_StableState* function); that is, a state where we can guarantee that all the pending cache operations are flushed to the disk.

After *committing* the current transaction effects, TFS needs to tell CTL that the current thread (associated to the current transaction) has ended, and therefore the virtual block addresses currently owned by the transaction are now free to be used.

Finally, TfsCommit checks if it is the last transaction currently running. In that case, it frees the two layers of TFS: the transactional Hashtable and the metadata Hashtable, also freeing all their possibly still inserted elements.

5.3.2.14 TfsAbort

```
void        TfsAbort                    (TFSThread Self, int retry);
```

The *TfsAbort* call has the objective of rolling back the current transaction effects. Since any altered virtual blocks are already present in the transaction blockwriteset, and *refreshblock* pré-

abort handler is always associated to any transaction, simply calling CTL's abort primitive will trigger this function tasks.

The function also has a *retry* argument; if *retry* is zero, the transaction is aborted and not automatically retried. Otherwise if the *retry* is one, the transaction is rolled back and automatically restarted.

5.4 TFS Major Features

Among the many possible features that a transactional file system has to offer, we decided to implement a few that prove the goals of this research. Note that some features described here are purely the result of applying transactional semantics to file systems, while others are optimizations performed in the architecture.

The following sections will briefly clarify some of the best features, explaining their usefulness and how they were achieved.

5.4.1 Concurrent File Access

The architectural approach for the prototype specified two main layers, the Cache Manager and the TFS Transactional Layer. By having a Cache that maps sets of disk-blocks (virtual blocks) in memory locations, and having all the accesses controlled by a transactional engine, it is possible to have a simultaneous access of read and write from distinct transactions to different parts of the same file. This extended concurrency for files cannot be achieved by standard file system IO functions. With TFS library, the programmer can request a number of write or read handlers concurrently for the same file, and even perform changes to it, without experiencing problems such as *dirty* or *unrecoverable reads.*

Concurrent file access feature is important and could be very useful for applications that have heterogeneous data in the same file. As an example, let's think about .MDF database files which keep data and metadata inside, and require consistent updates. Nowadays these files are created, updated and read sequentially, but if the DBMS developer had TFS available at design time, it would be possible to specify separate threads for metadata updates, data updates and even simple read accesses. This would improve concurrency by allowing multi-threaded operations over the (possibly large) database file.

5.4.2 File Caching

The design mentioned above in section 5.4.1 (*Concurrent File Access*) implements a concurrent access model for files, but it also uncovers another valuable advantage, the in-memory caching of virtual blocks. Since the Cache Manager module is responsible for *low-level* mapping of disk blocks into virtual blocks and storing them into complexity-efficient data structures, it is easy to implement a *cache-like* behavior for the recently opened files.

Standard caching algorithms can be implemented in several different ways and can also be later optimized for specific operations, improving the overall cache hit rates. The current prototype does not implement cache optimizations, although they are possible for future improvements.

The Cache Manager keeps the recently opened files in a hashtable, and manages the virtual blocks of each file inside its hashtable element, using a singly-linked-list. For simplification of the integration with the STM engine, the cache is only freed when TFS detects no running transactions, since otherwise the same virtual block could be freed and reallocated in a new memory location, making the CTL synchronization point invalid.

The profit of having a low-level file caching is proportional to the number of accesses to the same set of virtual blocks. For example, a program that periodically needs to read or write a certain file, may obtain speedups when using TFS; the file is loaded once to the central RAM memory and from that moment until all transactions terminate, all file accesses are mapped to memory accesses. We have previously stated that, statistically, a single hard-drive access takes 5 to 10 milliseconds. In that same time, a single core processor could do millions of instructions which include accesses to the central RAM memory. This means that depending on the behavior of the example application, our prototype could imply an enormous speedup because of this caching feature.

5.4.3 Semantics for Growing Files

There are two different modes for opening a file in *write-enabled* mode, explained before in detail in the section 6.3.2 (*TFS Call Mechanics*); sub-section 6.3.2.2 (*TfsFopen*).

The two modes are named *Exclusive Mode* and *Concurrent Mode*. These modes allow new ways of appending data to a file in a concurrent environment. In normal file system accesses, the *file handler* obtained by the *fopen* call to open the file with append permissions would implicitly

lock the file for any other thread or application. Our mode named *Exclusive Mode* is analogous to this standard behavior, having an unlimited (except for the underlying file system) grow length. The *Concurrent* mode introduces a new concurrent opportunity, since when a transaction needs to append data to the file, it simply asks the transactional engine for exclusive write-access permissions to the last virtual block of the file, leaving all the previous blocks unlocked.

We think that these two *append* modes provide a more flexible file API than the standard one, which can easily become handy for multi-threaded applications that operate concurrently over the files.

5.4.4 Journaling for Atomicity

TFS transactional model would become insufficient if we did not guarantee *Atomicity* for each transaction. With atomic file updates we can guarantee that a transaction will never produce its effects partially, since when it performs the first action, the other actions are already logged in a journal file, and can be later automatically reprocessed before the TFS layer is operational again.

The *Journaling* technique studied above for single file system updates, guarantees consistency of the file system data structures that keep data and metadata. After a lengthy study of a possible *journal* for TFS, we implemented a journal file integration that guarantees transactional Atomicity for TFS, and therefore resulting in *enhanced consistency* for TFS-enabled applications.

Like described before, standard applications that depend on personal user data (like the GNOME Window Manager) spread through several different files, never have the guarantee that on a system boot, the user profile is consistent. The system guarantees through journaling that each file is consistent in the file system (the data and metadata match each other), but nothing guarantees the application that half of the user profile files are new and updated but the other half still contains the old profile data.

By implementing journaling for TFS transactions, the application knows that if the transaction commits, all the file changes made inside that transaction will take effect between commit time and the next TFS operation.

5.4.5 Exponential Back-off Algorithm

In STM engines implemented with the *retry* feature, when a transaction aborts and retries, the thread is immediately switched to the location where it started the transactional scope. This is accurate for in-memory transactions, since their operations are performed considerably fast and the transaction can find a consistent schedule opportunity a few milliseconds after it has been aborted.

TFS implements file system IO operations which are thousands of times slower than the memory IO operations, so the instant retry of a transaction can find a consistent execution, but statistically it will abort a considerable amount of times. Imagine a transaction that is copying a 100 megabyte file to a new one, while other transaction is trying to read the original file. The result is that not only the disk throughput would be high while the copy is in progress, but also the CPU cores would be suffocated with the permanent abort and unsuccessful retry of the second transaction. As a result, we implemented a fast *exponential back-off algorithm* that pauses a transaction before it retries, using an exponentially growing time slice. This time slice begins with 1 millisecond, and grows exponentially until reaching a defined maximum (defined in *TFS_API.h* file as *RETRY_MAX_LOOP_TIME*) currently set to 5 seconds.

This feature was an optimization to the prototype which resulted in a smoother execution of TFS applications in specific scenarios like the mentioned concurrent transactional file copies.

5.5 Evaluating the resulting ACID Properties

Having studied before the characteristics and supporting techniques of standard transactional models, we decided to analyze briefly the resulting model of the TFS prototype.

Evaluating the ACID acronym, our model supports the four properties. We will clarify why and how below:

- ✓ ***Atomicity*** – TFS transactions behave atomically; either all effects are visible, or none are. This behavior is assured by the TFS ability to *refresh virtual blocks* from files whenever a transaction aborts. Also, *atomicity* for *commit* operations is assured by the *journaling* technique, guaranteeing that if a *commit* operation starts, all its data is already in the log file on persistent storage, and can therefore be replayed.

- ✓ ***Consistency*** – If we consider the file system consistency, we can affirm that it is the responsibility of the underlying file system on which the TFS-application runs. However, a new type of consistency is introduced by our prototype: the *application data consistency*. Since applications can now guarantee atomicity for operations over several different files, they can acquire a new level of consistency by using TFS; the application will always see and leave the file system in a desired, consistent state.

- ✓ ***Isolation*** – TFS transactions cannot see the effects of concurrent transactions. This feature was possible by delicately using the CTL STM engine together with its new Handler System, among a few other techniques for operating with the Cache Manager.

- ✓ ***Durability*** – The definition of durability itself makes it inapplicable to memory transactions since memory is on volatile storage. TFS performs transactions on hard-disk drives (like database transactions do) which implies working with persistent storage. Durability is therefore present in TFS, but only due to the fact that the IO performed on commit always reaches the persistent storage of the underlying file system. Note that this property would not be applicable if a TFS application runs over a ram-drive or other non-persistent storage devices.

6. TFS Evaluation

In this chapter we will demonstrate the tests that TFS has overcome. The objective of each test is specified inside each section, since they vary from feature testing, performance measuring to stress-testing.

The following tests are in the "tfs_tests" folder, and are explained briefly in the following table.

Test file name	Description
Tfstest_A.c	Non-transactional Cache operations
Tfstest_B.c	One thread testing operations
Tfstest_C.c	Multi-threaded *Datarace*
Tfstest_D.c	Multi-threaded, Multi-transactions *Datarace*
Tfstest_E.c	Multi-threaded, Multi-transactions, Multiple-files *Datarace*
Tfstest_F.c	Extra API functions tests
Tfstest_G.c	RW_MODE tests
Tfstest_H.c	CRW_MODE tests

Table 8 - TFS test file units

All test applications were developed using *Ubuntu* Linux 8.04, fully up-to-date at 10th, July 2008. The binaries were compiled for 32bits and tested on the same distribution of Linux, but in two different hardwares:

- ✓ ***Laptop*** – Intel Core2Duo with 2x 1.8GHz CPU, 2,5Gb of DDR2 RAM with latencies 4-5-5-15, and a single 4200rpm SATA HDD (average throughput 30mb/second). OS installed for x86 architecture.

- ✓ ***Desktop*** – AMD X2 with 2x 2.7Ghz CPU, 2GB of DDR1 at 500mhz with latencies 2-2-2-5, and 4x 7200rpm SATA "Single Plate" HDD in a Raid-0 array (average throughput 250mb/second). OS installed for AMD64bits, with *multi-lib* support.

6.1 Cache Testing

The first test application we developed was a non-transactional use of the Cache Manager, since it was the first module implemented. The objective of this test was to certify that the Cache Manager implementation was correct, including all the memory allocation, the journal utilization, virtual block mapping, synching and refreshing.

The test file performs the following actions:

1. Load an original existing file completely
2. Changes every vowel to the "_" symbol
3. Refresh every virtual block except block 4
4. Synchronize all the blocks
5. Open a new empty file
6. Grow to the file length of the original file
7. Read every block from the original file and Write to the new file
8. Synchronize every block of the new file

The result of this execution is that the original file gets all the vowels in its virtual block number four replaced by the "_" character, and the new file (named "OUT.txt") is a perfect copy of the original file, verified by the bash command *diff*.

We ran this test by executing the *Tfstest_A.c* file in the same directory as the file LICENSE.txt and performing *diff* over OUT.txt and LICENSE.txt.

6.2 Single-Thread Operations and Multi-Threaded *Datarace*

As seen before in Table 8 - TFS test file units, test file from B to D were the first ones to test file IO in a transactional context, from non-concurrent situations of operating over a file, to stress-testing situations of data racing for the same file. We consider that *Datarace* is the situation where, in a loose ordering, several processes or threads try to access the same location for reading and for writing, leading to concurrency and consistency tests. These tests forced the consistency and concurrency algorithms to become stable and correctly working and finally pushing our prototype to the expected behavior.

Although test B was a non-concurrent test of starting, executing and committing a transaction, test C used a considerable number of threads doing the same, and test D used those threads to provoke collisions and the mentioned datarace for the same virtual block of a file. The three tests can be run independently, needing only to compile and execute the desired one.

As a conclusion fact, the main purpose of these three tests was to test the resulting schedules of multi-threaded applications that read and write the same locations of a file, without having to code any ordering or synchronization algorithms. TFS proved worthy and valuable.

6.3 Flexible "Undesired Interleavings" Tests

Even though the above tests from A to D tested all the operational mechanics of our prototype, test E was probably the most important, since it operated simultaneously over several files. Because of this, a special situation was created to make sure that the transactional *Atomicity* was always verified.

Test E writes three files to the disk, two of them containing two random numbers, and the third file containing the sum of these numbers. Like in the known examples of *undesired interleavings* for testing concurrency control algorithms, the objective is that any possible execution of a transaction that reads the three files will always see a consistent version of the files; that is, the number in the third file will always be the sum of the two other numbers.

For that purpose we created a flexible and configurable test file, where the tester can specify how many updating and checking threads he wants to run, an optional *sleep* timer value for the updating threads and even a number of loops for each thread. If we use high values for these variable with this test, we can see many transactions aborting and retrying, the updating threads generating random numbers and updating the three files, and the reading threads always asserting that the sum they do is correct.

This was also a good test to see our journaling technique at work. We broke the execution several times by using the SIG_TERM shortcut (*ctrl+c*), and observed that when the application restarted, the Cache Manager checked for consistency in the journal file, and replayed it when necessary.

The results from these executions were the main proof of concept for our work.

There is not a single thread execution where the *sum* verification fails, even after an unexpected and abnormal termination of the system, meaning that our prototype managed to

eliminate all the desired interleavings and offered a consistent concurrency scheduler that is very easy to use for file system access.

6.4 Massive Stress Testing

In order to be able to evolve and thrive, we wanted TFS to be as rock-solid as possible. As it is expected, debugging TFS algorithms and architectural mechanics was not an easy task.

The solidness tests were performed by a *bash* script that looped the tests execution, incrementing a loop counter, and measuring the total time without crashing. The project was also tested while attached to the Valgrind memory checking tool, which to eliminate all the operational errors of the most unpredictable situations.

The tests were executed through Valgrind several times in both test machines for eight consecutive hours, resulting in not a single assertion or memory access errors.

6.5 Sample Bugs Found

There is not a simple way to measure when a project with this considerable dimension is bug-free and rock-solid. The role of the testers is to keep testing the most random and rare situations until they can find the next problem.

In the current version of TFS (alpha 0.7.1), the only real bug we found was a rare concurrency situation (average 1 for each 20.000 loops) when accessing the transactional hashtable, and only when the application is run in a multi-core CPU. Running through Valgrind (which means the threads are *serialized* to a single core), the prototype does not generate this bug, running for an undefined number of time and loops. Although we know that primitive operations over hashtables cannot occur concurrently, we implemented mutex locks with the finest-possible grain size to control the accesses to this structure, but apparently not completely serializing each primitive correctly.

The solution for correcting this bug is extremely difficult to find, even when debugging all the internal mutex lock and unlocks, since the situation itself is extremely rare and hard to reproduce.

However, a possible workaround for this bug is to implement a function that acquires a mutex-lock, returns the hashtable pointer, and leaves the mutex locked. The primitives that insert, search and remove elements from the transactional hashtable inherit the responsibility of

unlocking the mutex right after they work over the structure. Hopefully this workaround will be implemented soon for the *beta* version, eliminating the bug.

Apart from this synchronization problem, there are no other bugs that we have found in the current version.

7. Future Work

Although we think that our objectives of contribution to the scientific community have been achieved, we are confident that the *TFS* project will not end with this thesis. In this chapter we will not only provide information on how to extend and evolve TFS, but also about how can the current versions of the prototype become of great use to existing projects.

At first, we will focus on the use of the current prototype version, without any further development. Second, we will name and explain some of the possible evolutions of our current version.

7.1 Using our TFS prototype in the Real World

Since *TFS* prototype already provides a transactional layer for accessing a file system, many now-a-days applications can make use of it. The application file consistency inherited by the atomicity of TFS transactions is a plus, together with the caching effect and concurrency support.

We will explain how TFS would well-integrate and contribute to two example applications; a FTP server and the GNOME Desktop Manager. Please not that these are only two examples of a fast benefit from TFS, and we do not wish to provide the implementation instructions; we will instead specify some technical details for a possible implementation.

7.1.1 The Transactional FTP Server

A File Transfer Protocol (FTP) server is surely a great example for exploring the capabilities of a transactional access to the shared files. The FTP server supports multi-threaded concurrency, providing *read* and *write* access to the same set of files. In the majority of simple FTP servers, the behavior of opening two clients which attempt to change the contents of the same file concurrently is undefined. Usually users have *write* permissions on private directories, or only a single user has *write* permissions and all the others are only allowed to *read*, eliminating the concurrency problems.

It would be complicated to implement a transactional ftp server from scratch, and would lead the programmer to develop something similar to a database transactional model, locking on

files and parts of files instead of rows or entire tables. With the TFS layer, the FTP programmer should have a lot less effort in implementing the transactional server, since each client TCP socket (a good unique identifier for a transaction) can perform a *TfsStart()* call before executing commands, and a *TfsCommit()* call when the remote user requests a "*bye*" command. After that, the programmer simply had to make sure that the file IO accesses used must now have an extra parameter: the transaction identifier returned by TfsStart. TFS will then manage consistency over concurrency, and restart the transaction whenever necessary. Note that the event of rolling back a transaction that is coupled with a TCP socket requires a simple trick of storing the user-commands requested in a private ordered structure, so that the already performed commands are re-performed whenever TFS makes the thread jump to TfsStart. The transactional layer dynamics would be totally transparent to the remote FTP user and its FTP-client software. We think that this use of TFS would prove useful since the FTP clients would always get a consistent view of the remote directory.

If we imagine a FTP mirror like a Linux distribution binary repository, this consistency improvement is a must, since the mirror must maintain consistent file content (binaries) and a consistent set of applications and their required libraries. Also in this case, a second great advantage of TFS for the FTP server is that the server would gain a caching capability without implementing anything else for it.

7.1.2 The Transactional GNOME Session Saver

The GNOME Desktop Manager is one of the most used graphical environments for working on Linux operating system. The user can configure a huge number of settings, from his desktop wallpaper image, his GTK theme, the session startup tools using *gnome-session-properties*, to even configuring it in a *key-value* registry editor using *gconf-editor*. All these settings get saved correctly in the user *home* directory, some in hidden files and others in hidden directories. The main problem arises because of the lack of *Atomicity* in the session-save or session-load primitives of GNOME; if the user changes his desktop environment in any way, a new setting must be changed in some of the sparse profile files, maintaining a loadable profile for the next user login event. But if the profile gets partially saved or simply inconsistent and corrupted, by the event of a system or application crash, the user might not even be allowed to login.

Unfortunately, a considerably large number of users have sadly reported these kinds of errors, from loosing their configuration to being unable to login, leaving them no easy choice but to delete the entire profile files and login again in an empty, generic environment.

Our prototype transactional model for file access offers the full ACID properties, from which the *Atomicity* inherited by the *journaling* technique would be very useful to GNOME. Note that in this case, neither the caching or the concurrency features would be of benefit, but the atomic file updates alone would fix a severe problem that has been *haunting* GNOME for decades.

We have spent quite a few time researching through the *gnome-session* package source code, and even contacting the GNOME development team for a clean and simple approach to plug TFS into the GNOME package. We have learned that the session-saving process is rather antiquated and unsafe: the session-saver signals all GNOME modules connected to the gnome-session with *DBus* (a flexible inter-process communication engine) informing that the session is terminating. If each module (and hopefully every module) is up and running and receives the signal, it will attempt to perform the write task by its own, using other separate techniques for helping in formatting the saved data.

Since the save of the file is not centralized by a single running process, it is therefore very complicated to plug this architecture into TFS. However, if the gnome-session daemon *requested* the data to save in each file, instead of signaling the logout and terminating, the use of TFS would be trivial to implement. Whenever a GNOME profile was to be read of written, the daemon should start a TFS transaction, perform every file access using TFS and finally try to commit the TFS transaction. In the event of saving the profile, as soon as the daemon gets the *commit* confirmation, all the transaction effects are stable in storage, inside the journal file. Similarly whenever the user attempted to log-in, even after an unclean shutdown, if the daemon uses TFS for reading the profile files, TFS would check for a consistent journal file and pre-process it before accessing the files, passing to the upper-level (the daemon) a consistent view of the profile files.

It is therefore possible to overcome the consistency problem of GNOME's profile files, independently from the underlying file system, by using an intermediate transactional layer for file access; proving the architectural concept that resulted from our study and investigation.

7.2 TFS Evolution

Our prototype is an embryo of what it could become. Our study and research proved the benefits of an approach such as TFS, and our prototype served as a proof of concept for the entire research. However, even though we aimed at a prototype, TFS has a path to walk towards its own success; there are numerous features and improvements that our prototype can have.

We want this project to flourish, such as we want to help in that evolution making it easier. In this sub-chapter we will contribute with our knowledge on how could this embryo prototype evolve, and what are the advantages of each mentioned evolution.

7.2.1 Extending the TFS API

TFS mechanics have been implemented. The interactions between the Cache Manager and the TFS Layer are intuitive and working, which leaves the API in a ready-to-grow state. The API itself could be optimized through profiling execution results and studying the less-optimized functions, but it could also be augmented improving its flexibility and use cases. Our implementation of the prototype led to an API of fourteen primitives, but there are still a lot of unimplemented functionalities. We will name and explain a few ones, while trying to explain the best approach we know to its implementation.

- ***File Deletion*** – The prototype does not have a *TfsRemove* function implemented. Since our implementation caches the effects for each transaction (*replay-logging*), implementing the TfsRemove primitive would be as simple as blocking the file in the transactions open file linked-list. The metafile should remain untouched, but the transaction that think it has deleted the file, can no longer access it. However, a *deletion* Boolean flag should be added to the transactional file structure, and should be set to positive after the first deletion. Using this flag, if a transaction deletes a file named "A" for example, but after that the transaction opens "A" again, the TFS layer knows that it has been virtually deleted and should ask cache to use a new, empty, temporary file named "A_". Finally, the Cache Manager should implement a *move* primitive to be able to put the temporary file in the place of the correct one. In order to validate these effects, a few rules must apply:

a) The metadata open counter must be 0 at the time of commit.
b) After the deletion, a blockwriteset *meta*-element should be added (like for CRW mode) which indicates the *move* of the temporary file to the new file, therefore committing all the effects needed at commit time.

- ***File truncation*** – Our API lacks the implementation of the transactional equivalent to the *ftruncate()* system function, used to diminish the length of an existing file. In our current implementation, this additional function would require three simple steps:

  a) A new blockwriteset meta-element indicating the FUID and the new length.
  b) The function itself could request a CTL exclusive lock for all of the virtual blocks starting from the new length to the old length, that is, the part of the file that is about to be deleted.
  c) At commit time, in the *forEach* loop that processes the transactional blockwriteset, the presence of a *truncate meta-element* should check for the file open counter, and if all is correct it should request Cache Manager to truncate the file, using a new *Cache_Truncate* function, to maintain the categorization and modularity of the both layers.

7.2.2 Implementing a Specific Transactional Manager

Another evolutionary perspective of TFS might be related to the maintenance of transactional consistency: the transactional engine. We used an STM engine not only to fulfill our transactional file system objective, but also to allow a transactional memory engine to perform persistent storage, the *Durability* property that does not exist on normal STM engines.

This STM usage can be replaced by a totally new transactional manager, specific for TFS needs. The resulting speedups would not become much visible since CTL is indeed fast open to specific customizations and improvements, but the code would become more specific, and would overcome the CTL's read/write sets limitation. This would allow the triumph over the file growth limitation in the *Concurrent Read Write* open mode.

The specific TFS transactional engine would need to implement an event system similar to CTL's Handler System in order to associate events before and after the outcome of a transaction, calling Cache_RefreshBlock for the transaction write set before the transaction aborts.

If we keep our caching architecture, there are many possible ways of implementing a transactional engine based on the *global version clock* algorithm for example. The manager could work similarly to CTL, keeping two linked lists for transactional read and write sets, and at each element of the list, TFS could keep the triplet containing the file unique id, the virtual block number and the versioning clock value from when the element was added to the list. Using this technique, the cache could instantly free unneeded blocks or simply re-fetch previously fetched blocks to new memory locations without problem. Even if the memory address containing the block changes, the read/write sets elements are the referred triplet, and not the memory location.

This would result in a much simpler implementation, slightly more efficient and definitely less memory-demanding.

7.2.3 Blending the API in the Linux Kernel

We have studied before the advantages of having an intermediate layer for a transactional file system instead of implementing a fully transactional file system. Even so, the later still offers some advantages since the Cache Manager would disappear, melting the file system journal file with the transactional journal file, and the caching effect would become a responsibility of the operating system kernel in real-time.

Also, a major benefit from centralizing the TFS library is that the transactional manager would be running at maximum in one central unit, allowing different applications to obey the concurrency semantics when accessing the same files or directories. We have explained before one problem of using our lib in two different programs: the CTL libraries inside each executable do not share their global versioning clocks or their cached virtual block addresses. This downside would disappear if the file system itself had the transactional manager built-in.

This approach is a very demanding task. Some of the intermediate modeling phases are:

- ***Designing the file system data structures*** – More than designing a standard file system data structures, which imply fragmentation minimization, disk space management, techniques for reducing the metadata limits (like maximum file name length) and throughput optimization, the developers should also base their data structures on

transactional buffers for the redo\undo-logs. The design of the data structures would probably be the most difficult part.

- ***Designing a combined journal file for structural consistency and transactional atomicity*** – The journaling feature should work not only for standard file system primitives, but also for the transactional model demands. This implementation must handle two separate journal files, or a single file but with support for both types of processing cells.

- ***Developing tools for the file system*** – The file system itself must have a pack of user-tools to create, maintain and reshape the internal data structures. Commands like "mkfs.tfs" and "fsck.tfs" would be very important in order to achieve a flexible file system.

- ***Developing the modular API that integrates with the kernel, optionally in user mode*** – Two main paths are available to reach a mountable file system; developing the VFS required structures, or implementing a FUSE (or LUFS) interface. The former seems a bit more complicated since the VFS structures are pretty large, but offers the best performance between the two approaches, avoiding the overhead introduced by the FUSE or LUFS interface mechanisms. FUSE however is the fastest and easiest implementation option for a mountable file system.

Although this TFS evolution would be quite larger than our TFS prototype, it has its own downsides. If the file system itself is the transactional manager and the applications simply use the file system, it is impossible to an already compiled application to define each transactional boundary; that is, each *start* and *commit* calls. Without recompiling the applications, the solution we suggest is to implement an administration interface for the transactional engine, where the administrator can define which applications behave transactionally, and even more important: *when*. The interface could allow defining a path containing wildcards, and when an allowed application opens the first handler obeying that wildcard, the transactional scope is started. Similarly, when the application closes the last handler that matches the wildcard, the file system itself should attempt a commit call. However this is our suggestion for a possible approach on how to benefit from a transactional file system without having to recompile the client

applications. If the later is not necessary (the applications *need* to recompile), then a development and installation of a TxF-like library offering the *start*, *commit* and *abort* primitives and a per-application patch before compilation would solve this transactional boundary problem.

8. Conclusion

As a final balance of this dissertation, we believe that we have succeeded in each objective, achieved the expectations, and proven the motivations.

We have studied, stated and shown the advantages of creating transactional semantics for file systems. Also, we have succeeded in proving that one of the limitations of transactional memory engines is superable: the support for persistent storage primitives.

We have started from studying the origin of the two main cores of our project, the file systems and the transactional models, and we have researched and presented all the important facts that lead to the future union of these two entities.

In now-a-days World, the central subject of this work is becoming more and more popular, yet still in a very embryonic state. At this precise moment, Microsoft Corporation has already taken the first steps towards achieving a transactional semantic for NTFS through the use of the Transactional Coordinator for databases, while other investigators have also succeeded on integrating a database with a file system using FUSE. We believe, and honestly wish to have made that clear to the reader, that other challenging and more efficient approaches are possible.

We have emerged from the standard Linux file access functions all the way to a transactional file access context, using an existing excellent transactional memory engine to manage our consistency verifications. Also, we have shown how could we not have used a transactional engine at all, and developed the needed functionalities internally to TFS.

Last, but not least, we have proudly followed our beliefs not only when we evaluated and chosen one of the possible approaches for this project's profile, but also when we focused our minds for the most contributive-possible approach, using and developing only free and open-software tools towards contributing to the natural evolution of the scientific community.

9. Bibliography

[1] **Silberschatz, Abraham, Korth, Henry and Sudarshan, S.** *Database System Concepts - Fifth Edition.* Columbus, Ohio, USA : McGraw-Hill, 2005. 0-07-295886-3.

[2] **Warne, John.** *ANSA, An Extensible Transaction Framework: Technical Overview.* UK : Cambridge, 1993.

[3] **Moss, J. Elliot B.** *Nested Transactions: An Approach to Reliable Distributed Computing.* Massachusetts : The MIT Press, 1985. 0262132001.

[4] **Cunha, Gonçalo.** *Consistent State Software Transactional Memory.* Lisboa : FCT-UNL, 2007. Msc.

[5] **Cachopo, João and Rito-Silva, António.** *Versioned boxes as the basis for memory transactions.* Lisbon, Portugal : INESC-ID/Technical University of Lisbon, 2006.

[6] **Garcia-Molina, Hector and Salem, Kenneth.** *SAGAS.* Princeton : ACM Press, 1987.

[7] **Lomet, D.B.** *Process structuring, synchronization, and recovery using atomic actions.* New York, USA : ACM, 1977.

[8] **Herlihy, Maurice and Moss, J. Elliot.** *Transactional Memory: Architectural Support for Lock-free Data Structures.* New York, NY, USA : ACM Press, 1991. 0163-5964.

[9] **Herlihy, Maurice, et al.** *Software Transactional Memory for Dynamic-Sized Structures.* Boston, Massachusetts, USA : ACM, 2003.

[10] **Herlihy, Maurice, et al.** *Composable Memory Transactions.* New York, USA : ACM Press, 2005.

[11] **Herlihy, Maurice, Moir, Mark and Luchangco, Victor.** *A flexible framework for implementing software transactional memory.* New York, NY : ACM Press, 2006.

[12] **Moir, Mark.** *Hybrid Transactional Memory.* Burlington, MA : ACM, 2005.

[13] **Harris, Tim and Fraser, Keir.** *Language support for lightweight transactions.* New York, USA : ACM Press, 2003.

[14] *Transactional Locking II.* **Shavit, Nir, Dice, Dave and Shalev, Ori.** Tel-Aviv, Israel : ACM, 2006. Proc. of the 20th International Symposium on Distributed Computing (DISC 2006).

[15] **Dias, Ricardo.** *Cooperative Memory and Database Transactions.* Departamento de Informática, Universidade Nova de Lisboa. Lisboa : MSc Thesis, 2008.

[16] **Schönhart, Sabrina, et al.** Beginnings of the UNIX filesystem. [Online] February 6, 2008. http://cs-exhibitions.uni-klu.ac.at/index.php?id=216.

[17] **McKusick, M.K., et al.** *A Fast File System for UNIX.* Berkeley, CA, USA : ACM, 1984.

[18] **Schonhorst, Brad.** Evolution of the Unix File System. *New York City *BSD User Group.* [Online] June 6, 2006. [Cited: December 10, 2007.] http://www.nycbug.org/files/FFS.pdf.

[19] *Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast Filesystem.* **McKusick, Marshall Kirk and Ganger, Gregory R.** Monterey, California, USA : USENIX, 1999. Proceedings of the FREENIX Track: 1999 USENIX Annual Technical Conference.

[20] **Galli, Ricardo.** *Journal File Systems in Linux.* Balearic Islands, Spain : Novática and Informatik, 2001.

[21] **Wright, Charles P., et al.** *Extending ACID Semantics to the File System.* New York, NY, USA : ACM, 2007.

[22] **Garcia, João, Ferreira, Paulo and Guedes, Paulo.** *The PerDiS FS: A Transactional File System for a Distributed Persistent Store.* Lisboa, Portugal : INESC/IST, 1998.

[23] **Tweedy, Stephen.** *Journaling the Linux ext2fs Filesystem.* United Kingdom : LinuxExpo, 1998.

[24] **Reiser, Hans and MacDonald, Joshua.** *Reiser4 Transaction Design.* California : Namesys, 2001.

[25] **Larus, James and Rajwar, Ravi.** *Transactional Memory.* Wisconsin, Madison : Morgan & Claypool Publishers, 2006. 1598291254.

10. Annex A – APIs

10.1 The TFS API

```
TFSThread   TfsStart                                                    ();
```

Description	Starts a new TFS transaction
Parameters	None
Returns	The new transaction unique TFS ID

```
TFSFILE     TfsFopen        (TFSThread Self, const char *path, const char *mode);
```

Description	Opens a file for TFS operations
Parameters	***Self*** – The transaction TFS ID
	Path – The path to the wanted file
	Mode – The open mode (see Table 5)
Returns	A unique TFS File identifier or a *flag* in case of error (see Table 6)

```
int         TfsFclose                                   (TFSThread Self, TFSFILE fp);
```

Description	Closes a TFS File
Parameters	***Self*** – The transaction TFS ID
	Fp – The unique TFS File ID
Returns	1 on success

```
long        TfsFread    (TFSThread Self, void *ptr, int size, int nmemb, TFSFILE stream);
```

Description	Read data from a TFS-opened File
Parameters	***Self*** – The transaction TFS ID
	Ptr – Memory location to copy contents to
	Size – Size of each member to read
	Nmemb – Number of members to read
	Stream – The unique TFS File ID
Returns	Total bytes read on success or *EOF* if end of file reached

```
long        TfsFwrite   (TFSThread Self, void *ptr, int size, int nmemb, TFSFILE stream);
```

Description	Write data to a TFS-opened File
Parameters	***Self*** – The transaction TFS ID
	Ptr – Memory location to copy contents from
	Size – Size of each member to write
	Nmemb – Number of members to write
	Stream – The unique TFS File ID
Returns	Total bytes written on success or *EOF* if end of file reached

```
int         TfsFprintf              (TFSThread Self, TFSFILE stream, char *format, ...);
```

Description	Simulates the *fprintf* call for TFS Files
Parameters	***Self*** – The transaction TFS ID
	Stream – The unique TFS File ID
	Format – Format for writing data (see *fprintf*)
	… – Optional arguments referenced in *format*
Returns	Total bytes printed on success or *EOF* if end of file reached

```
int          TfsFscanf          (TFSThread Self, TFSFILE stream, char *format, ...);
```

Description	Simulates the *fscanf* call for TFS Files
Parameters	***Self*** – The transaction TFS ID
	Stream – The unique TFS File ID
	Format – Format for reading data (see *sscanf*)
	… – Optional arguments referenced in *format*
Returns	Total bytes read on success or *EOF* if end of file reached

```
int          TfsFputc           (TFSThread Self, int c, TFSFILE stream);
```

Description	Writes a character to a TFS-opened File
Parameters	***Self*** – The transaction TFS ID
	C – The character to put, cast to an *unsigned char*
	Stream – The unique TFS File ID
Returns	The character written or *EOF* if end of file reached

```
int          TfsFgetc           (TFSThread Self, TFSFILE stream);
```

Description	Gets the next character from a TFS-opened File
Parameters	***Self*** – The transaction TFS ID
	Stream – The unique TFS File ID
Returns	The character cast to an *unsigned char* or *EOF* if end of file reached

```
long        TfsFseek        (TFSThread Self, TFSFILE stream, long offset, int whence);
```

Description	Changes the current file cursor to a new offset
Parameters	***Self*** – The transaction TFS ID
	Stream – The unique TFS File ID
	Offset – The new offset value
	Whence – The offset referential (see Table 7)
Returns	The new cursor offset

```
int         TfsFrewind                  (TFSThread Self, TFSFILE stream);
```

Description	Changes the file cursor to the beginning of file
Parameters	***Self*** – The transaction TFS ID
	Stream – The unique TFS File ID
Returns	1 on success

```
long        TfsFtell                    (TFSThread Self, TFSFILE stream);
```

Description	Gets the current cursor offset for the file
Parameters	***Self*** – The transaction TFS ID
	Stream – The unique TFS File ID
Returns	The current cursor offset

```
int         TfsCommit                   (TFSThread Self);
```

Description	Try to commit an existing TFS transaction
Parameters	***Self*** – The transaction TFS ID
Returns	1 on success

```
void        TfsAbort                    (TFSThread Self, int retry);
```

Description	Aborts an existing TFS transaction
Parameters	***Self*** – The transaction TFS ID
	Retry – Boolean integer to indicate if transaction should be retried
Returns	Nothing

10.2 The Cache Manager API

```
fileUniqueID  Cache_Fopen                 (const char* path, int allowNewBlocks);
```

Description	Opens a file inside the Cache
Parameters	***Path*** – The path to the desired file
	allowNewBlocks – Boolean integer to indicate if the cache should append new blocks when the file grows, or if it should maintain the block number limit and augment each block internally.
Returns	The Cache File unique ID

```
int         Cache_Fgrow         (fileUniqueID key, long newSize, int allowNewBlocks);
```

Description	Grows the Cache File in memory
Parameters	***Key*** – The Cache File unique ID
	newSize – The new desired file length
	allowNewBlocks – Boolean integer to indicate if the cache should append new blocks when the file grows, or if it should maintain the block number limit and augment each block internally.
Returns	1 on success

```
int          Cache_RefreshBlock                    (fileUniqueID key, int BlockNO);
```

Description	Reloads a single block content from the original file path
Parameters	***Key*** – The Cache File unique ID
	BlockNO – The number of the block to refresh
Returns	1 on success

```
int          Cache_SyncBlock                       (fileUniqueID kev, int BlockNO);
```

Description	Writes a single block content to the original file path
Parameters	***Key*** – The Cache File unique ID
	BlockNO – The number of the block to synchronize
Returns	1 on success

```
int          Cache_Fclose                          (fileUniqueID key);
```

Description	Closes a file opened by Cache Manager
Parameters	***Key*** – The Cache File unique ID
Returns	1 on success

```
void         *Cache_GetBlockAddressInMem           (fileUniqueID key, int BlockNO);
```

Description	Returns the memory location to the desired file block
Parameters	***Key*** – The Cache File unique ID
	BlockNO – The number of the block to return
Returns	The pointer to the block structure (notice that this location starts with the *BlockNO* variable, followed by the real block data. Use the function below to automatically return the data location)

```
void        *Cache_GetBlockDataFromPtr                    (void *BlockAddress);
```

Description Returns the *data* location of a block pointer

Parameters ***BlockAddress*** – Pointer to block

Returns Pointer to data location

```
void        Cache_GetFileSizeByKey                        (fileUniqueID key);
```

Description Gets a Cache File's length

Parameters ***Key*** – The Cache File unique ID

Returns The Cache file's length

```
void        Cache_StableState                             ();
```

Description Force Cache to reach a stable state

Parameters None

Returns Nothing