Universidad Nacional Autónoma de México

Posgrado en Ciencia e Ingeniería de la Computación

Estructuras de Datos Persistentes en Sistemas Distribuidos

T E S I S

QUE PARA OBTENER EL TÍTULO DE:
MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

PRESENTA:
DAVID MÉNDEZ LÓPEZ

DIRECTOR DE TESIS:
DR. SERGIO RAJSBAUM GORODEZKY
INSTITUTO DE MATEMATICAS

MÉXICO, D.F. AGOSTO 2013

To Family (Poup, Mario, Oscar, Maggie, Lucy, Emma, Grandma, Jimena), Friends(Hugo, Fernando, Adriana, Jessica, Griselda Hector, Regina, Carlos, Max) and mentors (Sergio Rajsbaum, Maurice Herlihy) much was learned. To my mother, long gone, but little did she knew the influence she had To you reader, may you find mastery in your passions.

Contents

1 Introduction **1**

1.1 Background . . . 2

1.1.1 Transactional Memory . . . 2

1.1.2 Persistent Data Structures . . . 3

1.2 Objective . . . 4

1.3 Results . . . 4

1.4 Related Work . . . 5

1.5 Thesis Structure . . . 5

2 Persistent Data Structures **7**

2.1 Semi-persistence and Full-persistence . . . 8

2.2 Version List . . . 8

2.3 Persistent Linked Data Structures . . . 8

2.3.1 Fat Nodes . . . 9

2.3.2 Node Split . . . 9

2.4 Persistent Random Access Data Structures . . . 10

2.4.1 Vectors . . . 11

2.4.2 Hash Maps . . . 11

2.5 Persistent Fixed Point Access Data Structures . . . 12

2.5.1 Stack . . . 12

2.5.2 Queue . . . 13

2.5.3 Remarks . . . 14

3 Transactional Memory **16**

3.1 Software Transactional Memory . . . 17

3.1.1 Designing an STM . . . 17

3.2 Current State of TM . . . 18

4 Transactions and Persistence **20**

4.1 Incremental Persistent Data Structure Model . . . 21

4.1.1 Incremental Version List . . . 21

4.2 Examples . . . 24

4.2.1 Vectors . . . 24

4.2.2 Hashmaps 25
4.2.3 Stack 26
4.2.4 Queue 27
4.2.5 Linked List 28
4.2.6 Tree 29
4.3 Analysis 29

5 Conclusions **31**
5.1 Results 32
5.2 Future Work 32
5.3 Acknowledgments 33

Bibliography **34**

Abstract

In the last ten years, the necessity for parallel and distributed computing research has reached a very high level of importance. The state of the art of this research has started to be used in the most recent computer arquitechtures, and is becoming part of programming language's standard libraries. One of the most popular lines of research is Transactional Memory Systems. The goal of transactional memory is to reduce the complexity of programming concurrent programs. This systems borrow the approach database management systems have at handling concurrency. A way of implementing and expanding the capabilities of transactional memory systems is by utilizing persistent data structures. This kind of data structures log and store versions of themselves. In this thesis we show the implementations of incremental persistent data structures which are built to be utilized by transactions in a software transactional memory system in order to implement a broader object synchronization. The result of this thesis is a catalogue of incremental persistent data structures algorithms and show their viability inside a transactional memory system.

Chapter 1

Introduction

In the last decades, the development of parallel and distributed computing technologies has been growing in large rate. The necessity of this technologies goes in hand with the soon to be, fall of Moore's Law [8]. First proposed in 1975, Moore's Law states the following[7]:

> The complexity for minimum component costs will increase at factor of two at a rate of two years.

This means that the computing power and integration of digital cicuits will be doubled every two years. Most computer processors are manufactured using the integrated circuit technology known as *CMOS (Complementary Metal-Oxide-Semiconductor)*. The impact of Moore's Law applied to CMOS is that the signal dissipation of a CMOS digital circuit is not affected by the size (or length of the circuit it self), but only affected by the frequency of the signal. This means that for years, as we reduced the size of the circuits, we could also increase the frequency of the processors, since the signal needed to travel less distance. Eventually the same power dissipation ocurred on smaller and smaller areas. Heat dissipation becomes harder and harder, and eventually the costs and maintenance becomes too great. We were arriving to the limit of how many digital components in a limited space a circuit can have, before the energy needed for it to function overheats the circuit. The physical limit of the integrated circuit materials is about to be reached in the coming years.

The understanding of this fact lead engineers and scientists to search for new technologies and new ways of incrementing the computing speed of processors. One of the ideas was not only solving a computing problem by making the process go faster, but with the help of other processes. This means having two computing processes cooperate to solve a problem. In other words, if you cannot increase the speed of a processor, lets put two processors together. This is easier said than done, the reality of this very much complex.

The focus of distributed programming is to find the ability to program alongside concurrency. Concurrency in this context means how much two or more processes share in an execution environment. Researchers and engineers had to come up with the semantics of how a shared object behaves when operations are executed by one or more processes. By this we mean what is or should be the correct behaviour of shared objects.

The main problem we want to solve in the field of distributed programming is that of process syncronization. What we mean by process synchronization is organizing the execution of processes, that share objects or communicate with each other, in a way that their task inside a system is not affected. By affecting the task of a process we mean that the process shouldn't behave in a non specified way or produce an unexpected result.

Different ways have been studied and implemented that try to solve this problem. Most of the work has focused on designing and constructing data structures and algorithms that can handle process synchronization. One of the the first solutions for synchronization was *Mutual Exclusion*. Mutual exclusion tries to synchronize processes by only allowing one process access to a shared object at a time. This of course creates a bottleneck and turns parallel code into sequential code, because it blocks other processes in their execution. Researchers looked for ways of achieving synchronization without blocking processes. This in time shone light into a problem called the *Consensus* problem. Consensus refers to processes deciding a value of a shared object that is being accessed concurrently. To solve this what has been the most popular way of doing it is with primitive atomic operations.

1.1 Background

For several years, the most common tools in parallel and distributed computing have been locks, like monitors and semaphores, to solve mutual exclusion. As well as atomic operations, like compareAndSet to solve the consensus problem. Although useful in certain aspects locks and atomic operations have several disadvantages. The complexity of building distributed systems based on any of the two is quite considerable. The disadvantages get even more considerable the larger the system gets.

The main disadvantage for locks is that by having bigger and more complex systems, the more difficult for a programmer will be to decipher the amount of possible locks and interactions between the processes. By not being able to control this, the risk of deathlocks or starvation to occurr increases. Also in real life implementations a process can be terminated, by physical reasons or by the operating system, while it holds a lock.

The disadvantages of atomic operations are each one only operates in one space of memory. Trying to do atomic operations between multiple spaces of variables is complicated and in some cases it is imposible without using locks.

Eventually programming with this tools and concepts became very complicated and requiered a lot of expertise on the side of the programmers. [10] A new programming model emerged from the need to simplify this. Trying to part ways from the notions of critical sections in programs and more in atomic blocks. This model is called Transactional Memory.

1.1.1 Transactional Memory

In the recent years, there has been an effort to create something broader and more easy to learn than traditional distributed programs. This effort tries to make parallel and distributed programming more aproachable for programmers without parellel programming expertise. One of most succesful ideas has been that of transactional memory. Transactional memory basically takes the idea of database transactions, and tries to apply them in parallel computing, simplifying the programming interface.

The concept of transactional memory originated in the early 90's as a computer architecture for programmers to create their own read-modify-write operations [1] in order to help create lock-free synchronization more efficient and easier to use [18]. Eventually the same idea was used in a software to overcome the inflexibility of a hardware implementation. [17] Software transactional memory came to fruition with the same goal of making non-blocking objects easier to implement.

The idea of transactional memory, in both hardware and software form, takes the approach database management systems use to handle concurrent operations. We look at a sequence of operations being applied in shared resources as transactions. The same as in database systems if two transactions conflict with each other, then either one aborts and tries to execute the operations again, or both abort. Deciding what is a conflict depends on the transactional memory system implementation, as we will see in later chapters. Synchronization is achieved by detecting modifications of shared variables or registers by transactions and allowing or not allowing a transaction to commit. A transaction commits when its modifications to the shared resources are seen system wide. The point of transactions is that they appear to happen atomically. This means that the operations performed by the transaction appear to take effect instantaneously.

Transactional Memory seems like a very convenient tool, in the last year this concepts and research have started seeing uses in real world applications. Hardware transactional memory is now being used in newest Intel processors[2]. And software transactional memory is now used in the libraries of several programming language.

The challenge to overcome, right now, in transactional memory is the lack of full object synchronization.

1.1.2 Persistent Data Structures

Persistent data structures imply the idea of having data structures that remember and store within themselves previous versions. This is that for every basic operation of the data structure (i.e. an update, adding element, deleting element) we create a new version. The main idea of this data structures is not making copies of itself, but storing changes of itself in a space and time efficient way. Not only do we want to go back and see different versions of the data sturcture, but also create new versions from older ones. As we will see in the next chapter creating new versions from older ones can get very complicated.

The study of persistent data structures has been pretty passive in the last decades. Brief work has been done sporadicaly, but never has a really good application been found. The first approach on this subject came from the field of computational geometry. [4]. Several years later the study of this structures went into more depths and several good algorithms were published. [1] This algorithms focused on creating persistent linked data structures

[1] atomic operations composed by several smaller register operations.

[2] http://www.realworldtech.com/haswell-tm/

(lists, trees, etc.). The disadvantage of this algorithms was that they require a new implementation from the ground up.

The bulk of the study on this kind of data structeres happened in the early 90's, after that not much was done. In this thesis we compile a list of persistent data structure implementations, that range from lists, trees, vector arrays, maps, stacks to queues. We approach the problem in different ways, settling in a way that doesn't require for a completely space efficient persistent data structure. What we do is give methods for adding persistence to any data structure. We also give algorithms for persistent data structures that aren't linked data structures. The research of this data structures gave us enough insight to create a new categorization for this data structures, based on common behaviour when trying to add persistence.

This thesis focuses on the implementation of transactional memory systems implemented in software (STMs). Especially using a type of data structures known as persistent data structures. Next chapter we will study the importance of this technology in the field of parallel and distributed processes.

1.2 Objective

The bulk of the work of this thesis was made at a research visit in Brown University with Dr. Maurice Herlihy, one of the most prominent researchers in the field of parallel and distributed computing. The goal of the research project is to strengthen and expand the capabilities of software transactional memory systems (STM), utilizing persistent data structures. Persistent data structures, as we will see in later chapters, have functions and uses that seem to go well with STMs.

The objective of this thesis is to show how much benefit there is in utilizing persistent data structures on a transactional system. Also we want a new, and maybe better, way of implementing object synchronization in broader way than what current STMs are capable of doing. Another thing this thesis has for an objective, is to create a list of persistent data structures, and give an API that can be used inside a transactional system.

The importance of this work will be, that we will have a base for implementing object wide synchronization from which we can improve and refine and create a production worthy implementation. And also this is important because no one has studied persistent data structures in the environment of transactional memory systems, and there hasn't been any deep work on the different types of data structures and how they can be implemented in a persistent way. This will undoubtely open many new questions and areas of research.

1.3 Results

The results of the thesis is a data structure synchronization system for Software Transactional Memory that is flexible and non-restrictive. The system comes from working in persistent data structures and searching for applications and new implementations. We show examples of how it works and is adapted for several datastructures. Most of the examples are just basic groundwork ideas and not necessarily the most optimal implementations. We will show that the better implementations are very data structure dependent. This means that for each type of data structure we will need to modify the algorithms in order to have better performance.

The importance of this system include giving new tools and capabilities to transactions inside an STM. Aside from very fast rollbacks when a transaction aborts, we are giving transactions the ability to look at past versions of a data structure and do operations to it without interfering with other transactions.

Another thing that has been brought up but was not covered completely in the project is the ability to merge versions of data structures in order to give a very granularized serialization of two concurrent transactions.

1.4 Related Work

The related work from the peristent data structures perspective is very few. Other work in the field include the use of transactions in the cloud [5]. This paper shows how one can use persistent linked data structures stored across a distributed system. Other work, although not very academic, is the use of persistent data structures inside programming languages like Scala and Clojure [2]. This programming languages use persistent vectors and hashmaps, and some of the examples we give in this thesis take into account that implementation.

Current work in STM include several implementations of STM models in production level programming languages. Specially the Scala STM library. Most of the work done in this thesis has been planned to be used in the ScalaSTM library. The reason we use Scala is because of the sintaxis flexibility it brings. It lets us create the sintactic structures for STM, without having to modify the compiler or interpreter.

One of the issues this thesis tries to address is object synchronization in STMs. There are several attempts to have an efficient way to do this, almost all attempts having several drawbacks [20]. The problem with object synchronization is that sooner or later you encounter the need for locking. One method that is now a very prevalent design is dynamic STMs[21]. This model allows for objects to be created inside transactions, and handles synchronization with what is called *Obstruction freedom* which guarantee that a transaction will always make progress. In the normal implementation objects made by a transaction must be garbage collected if it aborts. Also contention detection between shared object is very coarse grained. This two last issues is something we address in this project.

1.5 Thesis Structure

After this brief introduction the thesis talks about persistent data structures in general, we give a brief background on them and then we proceed to talk about the importance of version lists, followed by examples of different persistent data structures. We show a classification of data structures based on the similarities of their implementations.

Following persistent data structures we talk about transactional memory systems. First talking about the background and the motivation for the research in transactional memory. We go a little more in depth in software transactional memory and talk about the current state of STMs.

In the next chapter we talk about the work done in the project, and presenting the incremental persistent data structure model. We give a list of implementation examples for several common data structures. We analyze the pros and cons of this model and expected performance in transactional memory scenarios.

Finally we give our conclusions and final thoughts of the incremental persistent data structure model. Of course, we write about the future work of the project, proposed improvements, questions still unanswered and possible branching work.

Chapter 2

Persistent Data Structures

Persistent data structures are basically, in a very ample way, data structures that let us do operations on past versions of it. The core goal of their study is to not simply copy the data structure and store it, but to create versions and store them in an intelligent and spatialy optimal way. Persistence in a data structure is an idea that started being talked about in the early 1980's. M. H. Overmars' article called *Searching in the past* was the first step in the study of persistent data structures. The necessity of this work came from the field of computational geometry, specifically the d-dimensional range search problem[4]. In computational geometry dynamic data structures are utilized to keep track of objects such that queries (this queries include point membership, range search, triangulations, etc.) can be answered efficiently. In some algorithms objects represented by the data structures need to be deleted or sometimes more objects are added to the problem. Doing this operations completely deletes or modifies the information of the data structure and most of the time we need to return to a previous state. For example the point location problem in a triangulated polygon is a problem that in some algorithms require going back to previous versions of a data structure [19]. The cited algorithm computes the point location by removing points of the triangulation until we are left with the convex hull. We then insert the points we removed, calculating inside what triangle the point is. It's obvious that removing and then re inserting a point in this situation can be done by going into the past of the data structure.

The data structures and algorithms presented by Overmars were designed for very specific computational geometry problems. Years later researchers Tarjan, Driscoll and Sarnak stepped in where Overmars left and continued to study persitent data structures. They created a more general approach for creating this structure, but only for linked data structures.

After that, not much work was done for nearly two decades. On this thesis we take from where Tarjan and company left and create more types of persistent data structures. We then focus on creating persistence from already implemented data structures. One of the goals of the research was to find ways of making a data structure persistent with least amount of modifications possible. This was because we want to implement persistent data structures in already functioning runtime environments. Creating data structures from the ground up requires a higher investment and testing in order to be used in a larger scale.

2.1 Semi-persistence and Full-persistence

Persistence in a data structure is normally achieved in two steps. One much more simple to design and implement than the second one. We define a *semi-persistent data structure* as a data structure on which we can only read past versions of the data, modifications are not permitted or requiere to make a full copy of the data structure. *Full-persistence* is something that requires much more work and thought. Full-persistence not only lets you read past versions, but also to create new versions from past versions, we can say branch out versions.

The creation of versions is dictated by how the implementation is done. Normally we want the common operations of the data structure be the ones that create new versions (Delete, adding an element or update). Utility operations (function map, etc.) are normally more expensive and we don't want them to create new versions.

In order to keep track of versions, persistent data structures use an auxiliary data structure known as the Version List in semi-persistent data structures and a Version Tree in full-persistent data structures. As we will see Version handling is critical for creating persistent data stuctures without building everything from the ground up.

2.2 Version List

One of the most important things we discovered about persistent data structures is that empowering the version list leads to simpler core data structure implementations, because adding persistence to a data structure can normally be done in two ways. The first one is creating the data stucture from the ground up with persistence in mind and with a very simple version list. Tarjan and Sarnak's linked data structures take this approach. The second way is trying to adapt an existing data structure to be persistent. This is the approach we use for this thesis.

When we try to adapt an existing data structure algorithm or design the complexity of implementing persistence resides in the version list. As we will see when we take a look at persistent data structure examples, adding persistence to a data structure adds an overhead in both time and space compared to non-persistent.

2.3 Persistent Linked Data Structures

A linked data structure is finite collection of nodes, each containing a number of fields. Each field is either an information field or a pointer field. Information fields contain data, and a pointer field has a reference to another node, or null if there is no node. This type of data structures include linked lists, trees and directed graphs. We will briefly explain Driscoll, Tarjan, Sarnak's method of creating persistent linked data structures[1]. One of the main characteristics from this data structures is that most of the complexity lies inside the nodes of the data structure. The version list is a very simple numbered list. An orther operation must be overwritten when we create a fully persistent data structures, this is mainly because the the order of versions is no longer numerical but positional inside the version list.

We assume that the only operations that create versions for linked data structures are: *insert, update, delete*. Each of this operations are broken down into access node operations and modify field operations. So what we want is to minimize the number of this operations that we add as overhead of creating persitence.

There are two popular methods for making linked data structures. One is more space efficient, Fat Node Method, the other one is more time efficient, Node Splitting.

2.3.1 Fat Nodes

First we define what a node is. In this context node is a structure that has a set amount of fields, with at least one information and one pointer field. What a fat node does is that it records changes to the original fields, creating new fields inside the node. Sets of nodes with one pointer field can be viewed as linked lists, nodes with two can be seen as binary trees.

A fat node creates an arbitraty number of values for each field. Modifiying a field in a fat node is done by creating a new field for the value that was modify. Also each field is tagged with a version stamp. The version stamp is created in when a new insert, update or delete operation is done and it is assigned to the modified fields by that operation. On an insert operation only one new pointer field is created in the data structure, plus the creation of a new node. On an update operation, only one information field is created. A deleting operation is can be a little more complicated, creating several new fields for several nodes. This can be seen if the underlying data structure is a tree that uses some sort of balancing.

Traversing the fat nodes requires the use of the version list. In order to traverse we need to know on which version we are traversing. Initial access to the data structure is done either by an array or by a structure that maps the version number to the head node (or root) of the data structure. Some versions can map to the same head node, but if it was modified in a new version then it must map to the appropriate node. Starting from the head node we traverse through the pointer fields. When we select which pointer to traverse we have to choose which version. We finally choose, by comparing the version stamps, the field with greatest version that is not greater than the version we are traversing in.

The only difference between semi-persistent and full persistent is how we compare version numbers. In a fully-persitent environment we no longer have a version list, but a version tree. The order query in a version tree is no longer numerical but also positional. This problem is solved in O(1) worst case time by Dietz and Sleator [22].

2.3.2 Node Split

The obvious problem with fat nodes is that they can grow to aribitrary sizes. Finding the version of a given field we want to access can start to get slow, compared to the non-persistent data structure, the more we update the data structure. A more practical approach is with something called Node Spliting.

Node splitting instead of having arbitrarily large nodes, we have fixed size. This node now only handles multiple versions of pointer fields, information fields only have on version. We define only a set amount of versions for each pointer field. There are three different kinds of new fields we have copy pointer, parent pointer and extra fields.. The copy pointer is reference to a copy of the node, that copy has newer versions of the other fields. The parent pointer is a reference to the parent node of the current version, this is necesarry for doing traversing in versions that are not in the original node. If there was no parent node, in order to know who was the parent of the node we have to go traverse the copy nodes in order to find out. The inverse copy pointer is used for full-persistence, it holds the pointer to the node that created it. The extra fields are used to carry an extra version of any of the other fields, be it a pointer field or a pointer field. As in the fat node method fields have a version tag including the parent pointer.

We create a new node when the following happens: there is an update of the value of an information field of the node, and when the extra pointers given to the node are filled up. We create a new node with the information and pointer fields copied (we put the new value in the information field if it was created from an update). Then we put the correponding parent pointer and version stamp on the node. The tricky part is what happens when one makes a copy of a node that has children. If the child node does not have an extra field to log the new parent, then we have to make a copy of the child node.

Traversing the split nodes is very similar to the fat nodes. We search for the highest version that is not greater than the the version we are traversing in, for each node we traverse their copies until we find the appropriate field.

As we have seen this methods for persistence go very deep into the data structure implementation (i.e. special node implementations). But the version list used is very simple and very straight forward. In order to do less intensive alterations to a data stucture in order to get persistence, we can add more complexity to the version list/tree.

2.4 Persistent Random Access Data Structures

Random access data stuctures are data structures where any of its elements can be accessed in constant time. Examples include arrays, vectors and hasmaps. We are going to treat arrays the same as vectors , because arrays are just vectors that stay the same size.

The most common implementation for persistence in this type of data structures comes from the work done in the Clojure programming language [2]. This programming language uses persistence but it calls it immutability. The main difference is that with inmutable vectors or hashmaps there is no version control, everytime an operation is done in an immutable object a new reference is made and the old one remains unaffected. Obviously as one creates new references this are not full copies of the data structure, internally all references from the same immutable data structure share as much as possible. What we do here is take the immutable data structures and add a version list. The version list can be the same as the one from linked data structures.

2.4.1 Vectors

The idea of the persistent vector comes from how memory pages are handled in logic memory. The internal structure of the vector is actually a collection of arrays each of size 32. Each of this arrays are referenced by another array of size 32, forming a balanced tree structure. The root of the tree is a reference to an array of size 32 which has 32 children arrays. The bottom level arrays contain the references to the objects stored in the persistent vector[16].

The reason arrays of size 32 are used is because on how the mapping of the index of the persistent vector to the correct array inside is done. The get operation is actually done by using bitwise operations of the original index. The index is represented in a 32 bit chain, we divide the index number in chunks of 5 bits. Because 5 bits are needed to count to 32. Essentianlly each chunk of 5 bits is the index for a level in the array tree. The right most bits are the index for the bottom level, the next 5 bits to the left are for the next level, and so on. Since the tree's height ,is at practice, never higher than 7 the look up for an object inside the persistent vector is always O(1). Also having a size 32 array is convenient because if the addresses are properly aligned they fit in a cache block.

Inserting in a persistent vector requires a little more work. Vectors only consider the 'cons' or append method, which inserts element at the end of the vector. Instead of creating a new array and copying the previous elements to it when we add more than 32 elements, we add a new leaf to the array tree, this new leaf will be referenced by the next entry of the root array. Whenever a root array is filled, that is, all it's 32 entries are pointing to other arrays, we have to create a new root array. This new root will point to the previous root on it's first entry and a new array in it's second entry, thus creating a new level in the array tree. In practice this will only happen 5 times, which makes the tree height 7, which means the maximum number of elements is 32^7, which is a very big number.

An optimization is used in order to optimize the insertion of a number. A tail array is used to buffer insertions. When this array is full it's only then when it appends it as a new leaf inside the array tree.

For the update operation a technique is used called Path Copying which we will talk about next section.

2.4.2 Hash Maps

To achieve persistent hashmaps we use a special variety of trees calle trie or prefix tree[23]. As with the persistent vector each node has up to 32 children. Leaf nodes contain a binding of the Hashmap. A binding is the pair of key and value objects stored in the hasmap. Non-leaf nodes are either IndexNodes or Colission Node. Index nodes depending on what level they are the lead to the leaf node with the binding we are looking or to the next index node.

Building the trie comes again from taking the hashcode created from the objects we are using as keys, and dividing their bit representation in chunks of 5 bits. The root of the tree is always an index node and the first least significant 5 bits of the hash code give us the index of what children to search for the binding. If the hash code has chunks different than 0 we create the index node at the corresponding level. Whenever there is a hash collision (there is already a leaf node with a binding with the same index) a Colission node is placed instead the leaf node and it either stores colliding values, or just marks the collision for error management.

In order to make it persistent we use a non destructive operation called path copying. When adding a binding inside the trie in order to create another version that doesn't interfere with previous version, we copy the index nodes the hash index points to (including the tree root) and only modify the new leaf node reference with the new binding. This method is used to handle updates in persisten vectors. Only the path of arrays taken by the index are copied, with only the last array modified with the new value.

2.5 Persistent Fixed Point Access Data Structures

Fixed point access data structures are data structures where contention happens only in one or two places. An example of this data structures are stacks and queues. The main characteristic of this data structures is that they can rely heavily on the version list in order to implement persistence. Normally one would implement a stack or a queue based of a list, but for the approach we are doing they are both using vectors to store the data. The reason we use vectors is because when you implement persistence you never want to destroy data. Vectors have no destructive operation, and update and append both take constant time. For both data structures we give more complexity to the version list in order to handle persistence.

2.5.1 Stack

We now give an example of a fully persistent stack. There is no previous work on either a persistent stack or a persistent queue, so this algorithms are completely new, and with more work can be improved.

The main aspect of the stack is the version list node. This node holds three extra fields, aside from version number and next version, previous-push node, consec-pops and top reference node. This means that the version node always points to where the top reference of the stack should be in that current version. The previous-push node reference is there to help compute valid tops for the versions. The consec-pops field is a counter how many consecutive pops have occured. The parent node is a reference to the previous version. Most of the computation of the stack will actually be handled by the version list.

The version list will actually be pointing to entries of a vector. This vector can be an immutable vector like the one described in the previous section. There will be never be the need for the vector to be updated because every entry will be the top of the stack of some version.

We start the stack with an initial version that is the one pointing to the empty stack, and with all fields set to null. The push operation is actually pretty simple, we take the value we are going to push inside the stack and append it to the vector, and calling the entry of the vector the new top of the stack. It is also very important that getting the new index of the vector and appending the value happen atomically, if it's not done that way we are

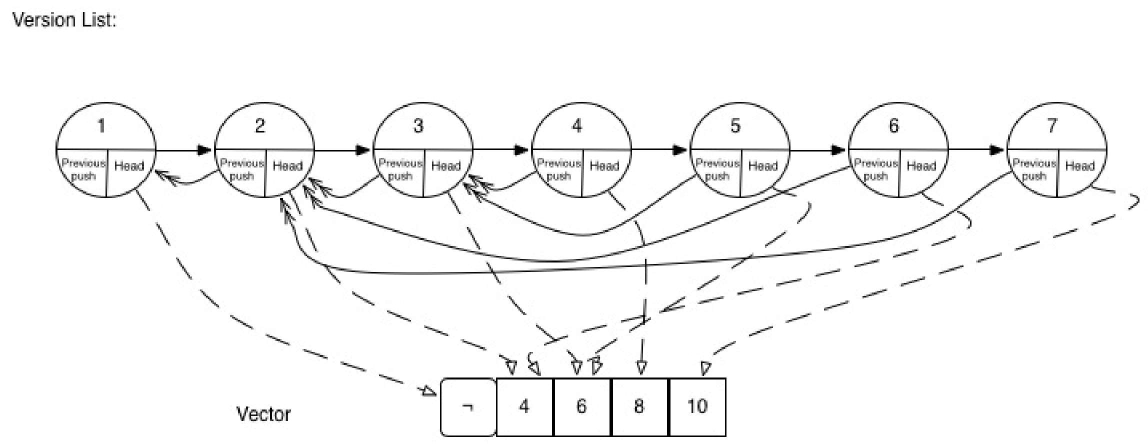

Figure 2.1: This diagram represents the execution of a persistent stack after the following operations: push(4), push(6), push(8), pop(), pop(), push(10).

going to have a bad time, because in case of a concurrent push, we can get the wrong top reference. After we have done both steps, we create the new version node with the correct top reference, if the parent's consec-pops is zero we set the previous-push node field to the parent node, else we copy the parent's previous-push node.

The pop operation has to first compute the next top pointer from the previous elements of the list. The operation goes as follows: first we take the value that the parent node's top pointer points to. We now compute the new top pointer. If the parent node's consec-pop field is zero we copy the top pointer from the previous-push node as the new top-pointer, we then set that same previous-push node as a the new previous-push node. If the consec-pops field is greater than zero we traverse the previous-node pointers one for each consec-pops. After we find the previous-push node we set it as the new previous-push node and set it's top pointer as the new top pointer. Finally we set consec-pops as the parent's consec-pops plus one.

The stack operations have an increase of time complexity because now the computation is done inside the version list. The push operation always does the same number of operations, it's complexity is still O(1) time. The problem is with pop operations. The fact that we must traverse the previous push nodes depends on how many consecutive pop operations there has been. In practice this number does not grow to large. Aside from that we obtain persistence while still having a low tradeoff in time complexity.

2.5.2 Queue

The persistent queue implementation is a little more simple. It still uses the version list to compute the head and the tail of the queue. The way we see the persistent queue is like an ever growing vector and each version has a unique head pointer and a tail pointer.

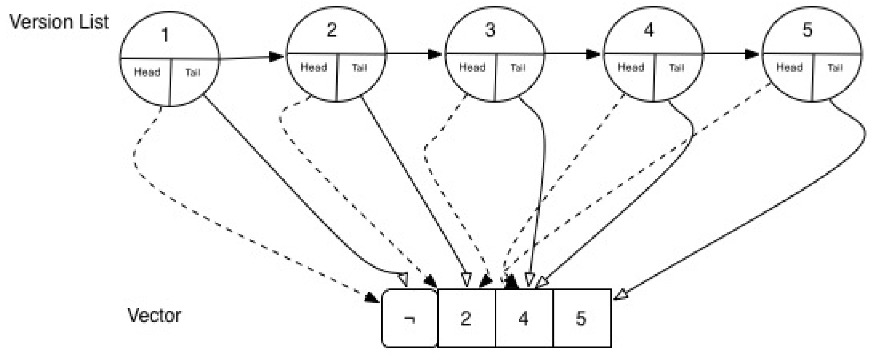

Figure 2.2: This diagram represents the execution of a persistent queue after the following operations: enq(2), enq(4), deq(), enq(5).

The same as the persistent stack we augment the version list node with more fields. And modify the enqueue and dequeue algorithms. The nodes for the queue now hold reference to the head and to the tail. As in the same way that the stack works, the underlying data structure where we stor the data is vector, and again we ca use an immutable vector.

The enqueue operation pretty similar to the push operation. We want to get the new tail pointer and add the value to enqueue to the vector in one atomic step. We then create the new version node with new tail pointer and we copy the head pointer from the parent version. If the parent version is the empty queue we set tail and head pointer with the same value.

The dequeue operation has to do a little extra, it first takes the value that is the head pointer of the parent version. We then have to compute the next head pointer, this is done using the fact that a head pointer was a tail pointer at some point, so we search for the tail pointer that was the same as the parets head pointer and we take the previous one. This may look a little unnecesary, but remember that this is full-persistence, there can be other versions' data inside the vector, simply decrementing an index is not possible.

The important improvement we have on the time complexity is on the side of the dequeue operation. Dequeue's time now depends on how big is the difference between the number of enqueue's and dequeue. If the diference is great the dequeue operation will take longer time.

2.5.3 Remarks

It is convenient to remark that the work on persistent stacks and queues is just a first approach. Better and more efficient ways to achieve full persistence can be found. All of this methods obviously increase the data structure time and space complexity. Although not by a lot (for indepth analysis go to [1]) we have seen that attempting to add persistence will add overhead to any data structure. Our goal is to minimze this, but not worrying too much because we will be adding more capabilities to transactions inside a transactional memory system that otherwise would not be possible.

Chapter 3

Transactional Memory

Transactional memory came from the need of making multiple variable atomic operations.[18] The main weakness of atomic operations is the inability of trying to do two or more operations across multiple variables atomically without blocking. As we have seen that is the main roadblock in order to make a comprehensible nonblocking data structure. By comprehensible we mean that it is relatively easy to understand, and doesn't require in-depth concurrent and distributed computing knowledge to implement. Non-blocking data structures take whole chapters in books in order to explain.

The original idea was to use an instruction set extension that allowed programmers to do their own atomic operations. In order to do it across multiple variables, they took the idea from how database management systems handle concurrent queries. Queries are handled inside transactions. The transactions execute the queries, if there is a conflict with another transaction, one of them aborts and rollsback the operations done. If a transaction finishes without conflicts it is said to commit, and the operations done are reflected in the database. The idea is to make multiple variable atomic operations into transactions if there is a conflict accesing one of the memory registers (by conflict we mean the original value was changed by another operation) it aborts and rollbacks. And with this transactional memory was born. This was quite the breakthrough because as we will see it changes paradigms in multiprocessor programming.

The approach taken here is now called hardware transactional memory and it started a whole new line of research all over the world. Years later it inspired to have this transaction approach not only in hardware, but in programming languages.[17] This was inspired because hardware transactional memory can only synchronize memory registers, obviously we want to synchronize more complex data structures. Software transactional memory shifts the paradigm of multiprocessor programming because we now don't reason concurrent programs by thinking where are the critical sections of the program. We now try to think in what parts of our program we want them to be executed in one atomic step. Code complexity goes down, and parallel computing becomes easier for mainstream programmers.

3.1 Software Transactional Memory

A transaction is a sequence of operations done by a single thread, this operations must appear to take effect instantaneously and atomically. The transaction at the end either commits (the operations take effect) or is aborted (the effects are discarded), if a transaction aborts operations on shared objects or variables must be rolledback. Rolling back in some case requires for inverse operations to be done in the shared objects or variables. This is sometimes a problem because there are operation where the inverse is very hard or expensive to do (i.e deleting a node in a balancing tree).

A very important point to make is that transactions are linearizable [24]. And not only that, they must have the safety property of opacity[12]. Opacity guarantees three things (1) all operations done by all commited transactions appear as if they happened at some single point during the transacation's lifetime, (2) no operation done by any aborted transaction is ever visible to other transactions, and (3) every transaction always observes a consistent state of the system.

Software transactional memory has the task of taking transactions, and instead of trying to synchronize memory registers and now try to synchronize variables in a programming language. At the very begininng software transactional memory had very little practical use. Memory for variables had to be preassigned for each transaction and also transaction size to be static. This limited what algorithms could be performed by a transaction. Algorithms that used dynamically growing data structures could not be handled.

The problem was locking, eventually, when people tried to have some dynamic object concurrency, locks had to be used. Coarse-grained locks made systems unscalable, and fine-grained locks made guaranteeing correctness and deadlock-freedom increasingly difficult. Eventually a way to implement dynamic objects was found [21]. Dynamic Software Transactional Memory (DSTM) is now used for several STM implementations. The main difference with DSTM is that it has the property of *obstruction-freedom*, instead of the stronger property of lock-freedom. By obstruction-freedom we guarantee that a transaction will always commit. This is the delay or the abortion of a transaction doen't prevent other transactions to commit. This lead to simpler and more efficient implementations of STMs.

3.1.1 Designing an STM

When implementing an STM there are 4 big design points one must address [13]:

- **Conflict Detection:** Conflict detection refers to at what step in a transaction are we going to detect conflicts, either on encounter, or on commit. Detection in encounter means that we detect read/write conflicts when they occur during the execution of the transaction. This requires special infrastructure in variables or shared objects to allow for detection, but conflicts can be detected earlier. Detection on commit makes the transactions a little bit faster but when we try to commit and detect a conflict, one of the transactions must abort and roll back, wasting a lot of time if the transaction was long. Most STMs use *single-writer multiple-reader* conflict detection protocol .

- **Locking scheme:** This refers as to how the transaction is going to commit their operations. A transaction must eventually acquire every location that has been updated by itself. There are two types of acquisition (1) eager, at the time the operation is done in that location, and (2) lazy it tries to acquire on commit time.
- **Contention Management:** This refers to how we handle a transaction when a conflict is detected. There are two types of contenciton management (1) timid, which refers to aborting inmediatly when we detect a conflict, and (2) greedy we try to complete the transaction, this helps in write/write conflicts, at least one of the transaction commits.
- **Read visibility:** This refers to letting other transactions know that someone read a shared variable. Making reads visible helps in conflict detection, but can sometimes creat fals contentions if the reader was aborted or doomed to abort.

Some of this STM designs favor short transactions others favor long ones. The state of the art STMs [13] use a combination of this design decisions for when it's mor convenient. For example SwissTM (the STM model used in Scala) is lock-based invisible read model with commit-time conflict detection in read/write conflicts and encounter-time conflict detection for write/write conflicts. It has two-phase contention manager that is greedy for long transactions and timid for short ones.

3.2 Current State of TM

In the last couple of years, the use of transactional memory has been getting more and more popular. Although research has improved HTM and STM into really promising ideas for market level products, there are still many improvements to make. One of the problems that still present a challenge is an efficient way to do full object synchronization between transactions. This will allow for more complex data to be shared by transactions in a non-blocking way, and not only that, but presented in a way that is friendly to programmers.

One of the newest and most useful ideas that try to achieve this is transactional boosting[]. Transactional boosting seeks to utilize current state of the art concurrent data structures inside transactions. This is done modifying this data structures in a very minimal way by adding locks called abstract locks. Abstract locks let us detect conflicts between transactions, but also letting non transaction code acces the data structure, and linearizing with the transactions. With this we have to keep in mind that operations outside transactions cannot be aborted. The main condition this concurrent data structures must comply is that every method must have an inverse, and a reasonably efficient at least. Abstract locks detect conflict, by checking the conmutativity of operations going on in the data structure. In order for a transaction to acquire the lock the operation it's doing must commute with other operations being done in data structure. If the abstract lock conflicts the caller must be delayed.

The work in this thesis looks to address the issue of object synchronization from different approach. The incremental persistent data structure model also looks for flexible and concurrent data structure friendly way for transactions to work. Conmutativity in operations is something our model is also looking at. The reason is because this model seeks to add more capabilities to a transaction inside an STM. This capabilities include working on past versions of a data structure and create a local copy. Creating checkpoints for a transaction, and constant time rollbacks for a transaction.

In the next chapter we will explain this model and give some examples of it's applications, followed by a brief analysis of model.

Chapter 4

Transactions and Persistence

We will now present an object synchronization model for transactional memory that uses persistent data strucrures. As we have seen achieving good and efficient object synchronization is a considerable challenge. Most of the work in this modeled started with the basic persistent data structure knowledge. After looking into what makes the persistent data structures work, and seeing the importance of the version list, we decided that our approach had to go in the direction of making a robust version list. This also goes in hand with the fact that from the very beginning we wanted a systen that does not require to create completely new data structures. We want something that can use something already that esixts. Although using completely persistent data structures built from the ground up gives this system better performance overall.

An important remark we make in this chapter is that the ability of merging versions in persistent data structures is a problem that we realized is hard to solve in a reasonable and efficient time. A complete solution was left outside the scope of this thesis. The solution given here is the simple one, but less efficient.

The importance of merging versions is that this gives us a very fine grained object synchronization. What we mean for merging versions is that, assume we have two transactions that just did a sequence of operations with a persistent data structure, each transaction has their own version of the data structure at the end; merging the two versions should give us a new version of the data structure where both sequences of operations were executed. Of course the merging can only happen if the sequences of operations are compatible. Right now the only criteria for compatibility we have proposed is commutativity of operations. Commutativity of operations in a data structure means that no matter what order 2 sequences of operations are done, the result is always leaves the data structure in the same state. This explains the difficulty of merging versions right now, because determining commutativy in single operations is very straight forward, but determining commutativity of sequences of operation of arbitrary length gets very complicated. We tried, and failed, several approaches for a method to determine commutativity without actually executing both operation sequences in both orders and comparing the state of the data structure. For the moment this was left out the scope of the thesis, but obviously is a point of supreme importance. For now, we assume that the transactional memory system has some mechanism that limits concurrency to transactions whose operations can be merged in a meaningful way. We focus here on how that merging is accomplished.

Working from the start with transactions in mind, gave us an easier time implementing this model. Version searching became much easier. One thing about making persistent datastructures with robust version lists is that finding and comparing the order of versions takes more time, and considerable thought has to be put. Thi is becuase of the fact that version lists no longer are lists in the full sense of the word, but trees and in some cases directed acyclic graphs. We fortunately bypass a lot of this starting with the fact that we assume transactions handle the versions they own. This means that transactions, for each persistent data structure, have a reference to the versions they own, or to versions that are shared.

This model was inpired by a model of concurrency control in memory called Conversion [15]. This memory model takes the file repository approach, letting processes work in a local copy of the memory and then converge their operations into the same into the shared part of the memory.

4.1 Incremental Persistent Data Structure Model

The incremental persistent data structure is way of making data structures persistent, but trying to minimize the amount of versions the whole data sturcture has to handle. We use the idea of logging operations of the data structure from a commited version, and eventually compiling the changes into a commited version where the operations done are reflected. The use of persisten data structures guarantees that this process is lock-free, because commiting the data structure is basically just making the version visible outside the transaction.

We use the version list in order to log the operations of the data structure, as we will see in the examples section some operations start having an increase of time complexity in the order of the length of the transaction. The big advantage having the operations logged has is that we can use any data structure underneath the version list. Since there is not a big amount of commited versions one can use the naive persistent data stucture construction, which is making a complete copy of the data structure for each version. Eventhough this is something we don't want because having a well implemented persistent data structure beneath our version list, speeds up the performance and space.

4.1.1 Incremental Version List

The core part of this system lies in the implementation of the incremental version list. We call it list eventhough it technically is a directed acyclic graph, but that would be a long name an sound funny. In theory every edge of the acyclic graph is labeled with the operation that created the version. The DAG is composed by 2 types of nodes: (1) *incremental nodes* and (2) *commit nodes*. Incremental nodes log the operation done on the data structure. It normally has fields for the parameters of the operation and for the result.

In most cases extra fields ar added for certain data structures. This new fields help us to be able to do data structure operations starting from an incremental node versions. Commit nodes are more like traditional version nodes. The main component is a pointer field that points to the underlying data structure. Commit nodes have 2 subtypes: Shared commit node and local copy commit node. Shared commit nodes are visible to all transactions and possibly visible to non transactional code. Local copy nodes are extactly that, nodes that are only visible to single transaction. Operations done in this version of the data structure must not affect the native data structure.

Commit nodes are created in two different ways. The first one is done by compiling the operations logged by the incremental nodes. This is made by actually executing on the native data structure the incremental node operations in the same order, creating a new commit node and setting the reference to the data structure in the new state. Executing this operation with a native data strucutre that is not persistent, requires for a complete copy to be made before executing the operations. This step is the one that benefits on having a good and efficient persistent data structure, because it does not require to copy the native data structure, the operations can be executed directly without the worry of interfering or modifying the previously commited version. The second way of creating a commit node is by merging versions. Starting from 2 incremental nodes from 2 different sequences of operations, we want a commit node that references a version where both sequences of operations appear to have happened. Right now the only way to do this is by going back to the closest common commit node of the two sequences, and executing on the native data structure one sequence after the other. The resulting data structure is then referenced by the new commit node.

The important aspect of the merging of versions is, that is not something a transaction by it's own does. A contention manager is the one that decides if two transactions' operations can be merged and if they are merged. We only concentrate on the task of merging two versions. In the current design the only merge criteria we handle is commutativity of sequences of operations.

Shared commit nodes are the most important part of the version list. This is because they are the ones that reflect the valid states of the data structure in the concurrent environment. Shared commit nodes are created by either upgrading a local copy node to a shared commit, or by the merging of versions. In the current design when a transaction tries to commit it creates a local copy node of the data structure once the contention manager decides if the transaction is to be commited then that local copy node is upgraded to shared commit node in a single step, achieving the goal of transactional memory, having sequences of code happen in a single atomic step. Of course, if it is decided that the transaction should abort then the roll back is pretty easy to execute, we discard the local copy node and the incremental nodes.

In every use of this data strucutre model the first version is always the empty data structure, and it is always a shared commit node.

A final remark about implementing this model is that the only condition we ask for native data structures is that they implement read-only exploring interface, this means we are able to explore the whole data structure without modifying it. This property we call it

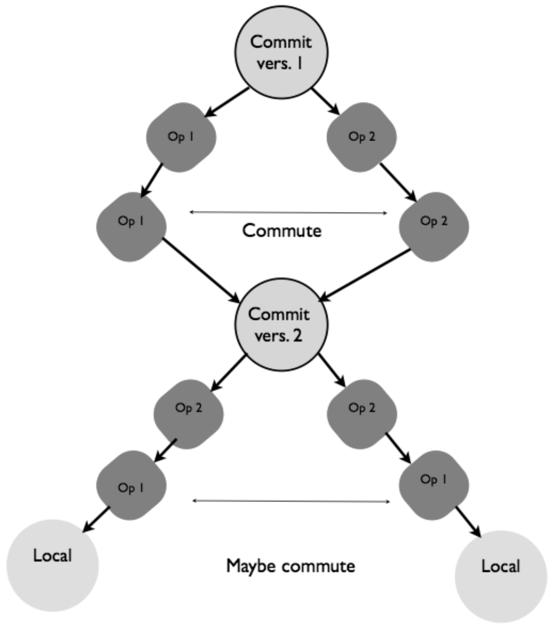

Figure 4.1: An abstract view of the Version List DAG.

transparency.

4.2 Examples

We now present a list of several examples of data structures utilizing the incremental persistent model. Each data structure has unique way of implemting the incremental nodes. In practice, instead of actually labeling the edges or pointers of the nodes, we use object oriented programming and add a subclass for each type of operation for the incremental nodes. For each data structure we override their basic operations so that it's diffent if it was called in an incremental node or on a commit node. Also we need to override because we need to simulate the behaviour of the native data structure in the incremental node sequence.

We exclude the merge and the make local copy because in this cases is the mostly the same for each data structure.

4.2.1 Vectors

$$\text{Vector} \left\{ \begin{array}{l} \text{operations} \left\{ \begin{array}{l} \text{update} \\ \text{append} \\ \text{lookup} \end{array} \right. \\ \\ \text{incremental nodes} \left\{ \begin{array}{l} \text{Update=>(value,index)} \\ \text{Append=>(value,newIndex)} \end{array} \right. \\ \\ \text{lookup(index)} \left\{ \begin{array}{l} \text{We search the incremental nodes starting from the most recent.} \\ \text{If we find the node with the index we are looking for we return value.} \\ \text{If we reach the commit node we do the normal search in the native data structure.} \end{array} \right. \\ \\ \text{Update and Append} \left\{ \text{We insert the corresponding incremental node.} \right. \end{array} \right.$$

This implementation is pretty straight forward, just add the operations to the list, and if there is a read we search the the incremental nodes first. This is a drawback, because lookup in a vector is constant, and now it depends on the size of the incremental node list, which in turn depends on the size of the transaction.

4.2.2 Hashmaps

- Hashmap
  - operations
    - lookup
    - newBinding
    - removeBinding
  - incremental nodes
    - NewBinding=>(key,value)
    - RemoveBinding=(key)
  - lookup(key)
    - We search the incremental nodes starting from the most recent.
    - If we find the node with the key we are looking for we return value.
    - If we find the key in a RemoveBinding node we throw an exception.
    - If we reach the commit node we do native search in the native data structure.
  - newBinding
    - We first check for a collision. Traversing the incremental nodes.
    - If we find a RemoveBinding node with the same key we add NewBinding node
    - If we find NewBinding with the same key, we flag for collision.
    - If we reach the commit node we check for collision there
    - If there is no collision we add the NewBinding node.
  - removeBinding
    - We insert the Remove Binding incremental node.

The same as vectors, looking up a key takes a much greater time than the original data structure. We go from a near constant time look up to possibly linear time.

4.2.3 Stack

- Stack
  - operations
    - push
    - pop
    - peek(lookup)
  - incremental nodes
    - Push=>(value,previousTop)
    - Pop=>(value,previousTop,popsInCommited)
  - peek
    - We search the incremental nodes starting from the most recent.
    - Case Pop node go to previous top and repeat.
    - Case Push node: return value
    - Case commit node: do peek in native data structure.
  - push
    - We create Push node.
    - We check parent version node type:
    - Case Pop node: copy preiousTop
    - Case Push Node: Set previousPush to parent, add Push node
    - Case commit node: se previous top to parent, add Push node
  - pop
    - We create Pop node.
    - Check parent version node type:
    - Case Push node: copy previousTop and set value to Push Node value
    - Case Pop node and popsInCommited == 0: repeat on parent
    - Case Pop node and popsInCommited == x: set previousTop to commit node, value = (native top - x) and popsInCommited= x+1
    - Case Commit node: set value to native stack top and popsInCommited = 1.

Like the stack example in chapter two we do something similar to achieve persistence. We have to keep track of previous top pointers inside the version list. On this version list again we have an increase of complexity in both operations. Calculating the overhead is tricky because, for example pop's complexity depends on how many consecutive pop's have happened.

4.2.4 Queue

- Queue
  - operations
    - enqueue
    - dequeue
  - incremental nodes
    - Enqueue=>(valueIn,previousDeqs)
    - Dequeue=>(valueOut,previousDeqs)
  - enqueue
    - We create the enqueue node.
    - We insert to the version list.
  - dequeue
    - We create a dequeue node
    - We look at the value of previous dequeue on the parent
    - We look at the size of the native data structure
    - Case size greater than dequeues:
    - value to dequeue is native head pointer minus previousDeqs.
    - Case dequeues greater than size:
    - we take the (previousDeqs -size)th enqueue node in the version list, from top down.
    - set valueOut to valueIn.
    - Increase previousDeqs by one.

The design for the queue can be improved by having previousDeqs as a global variable, or a global head pointer. The drawback of this design is that the dequeue operation now can potentialy take non constant time. Searching for the proper enqueue node takes time dependending on how many total dequeues we have done previously.

4.2.5 Linked List

- List
  - operations
    - insert
    - update
    - delete
  - incremental nodes
    - Insert=>(value,newIndex)
    - Update=>(newValue,Index)
    - Delete=>(index)
  - get(index)
    - We search bottom up in the incremental list.
    - We use local variable offset and set it to 0.
    - If at some point index is the index of the (currentnode index + offset) then return the value
    - Case Insert node: if index > currentnode index + offset,
    - increase offset, continue to parent node.
    - Case delete node: if index >= current node index + offset
    - decrease offset, continue to parent node
    - Case Update node: continue to parent node
    - Case commit node: return the native get(index + offset)
  - insert(index,value)
    - We insert the corresponding incremental node.
  - delete(index)
    - We insert the corresponding incremental node.
  - update(value, newIndex)
    - We insert the corresponding incremental node.

This data structure is different from the others because the operations that create versions actually take constant time. The change comes when we want to do a lookup in the data structure. The overhead we get from using the incremental nodes depends on the number of operations done to the list. The lookup algorithm works by having an offset for the index we are searching, this is explained by the fact that if you modify the list before the index you are looking for it shifts the index values for all the nodes to the right, if you delete you decrease the indexes, if you insert you add the indexes. At the end we take into account how many times the indexes have shifted and do a search in the native list taking into account the offset.

This method adds a less overhead to the original data structure in it's lookup, than the other data structures. Since the search in a list is linear , adding the incremental list only adds a linear factor to the search. The advantge we gain is in the other operations, deletions, insertions and updates are done in constant time.

4.2.6 Tree

The implementation of this model on trees is very bare-bones, since there are many different types of tree implementations. The approach we take is to try to interfere as less as possible to any balancing or restructuring operation that happens inside the tree. The lookup operation is pretty throw away because it depends on what we are looking for in a tree, and that changes. As same as with the Linked List, Insert, Update and Delete operations take constant time, but all lookups are hit with a linear time overhead, that depends on the amount of operations done previously to the data structure.

- Tree
  - operations
    - insert
    - update
    - delete
  - incremental nodes
    - Insert=>(newValue)
    - Update=>(oldValue,newValue)
    - Delete=>(delValue)
  - lookup(value)
    - We search bottom up in the incremental list.
    - Case Insert node: if value == new value return node found
    - Case Update node: if value == oldValue return node not found
    - if value == newValue return node found
    - Case Delete node: value == delValue return node not found
    - Case Commit node: do native lookup on tree data structure.
  - insert(newValue)
    - We insert the corresponding incremental node.
  - delete(delValue)
    - We insert the corresponding incremental node.
  - update(oldValue, newValue)
    - We insert the corresponding incremental node.

4.3 Analysis

The incremental persistent data structure model, as we have seen has some advantages for achieving object synchronization. It is flexible enough that any data structure works with it, but it works better with fullpersistent data structures.

The overhead we add the data structures varies from type to type. Some have big drawbacks but others don't. As we can see random-access data structure the look up time for a piece of data increases by linear time that depends on the number of operations done previously. The use of this model will benefit mostly short transactions. One idea that comes to mind is using a method that tells us if the key, or index we are looking for is on the incremental list, if it's not then we search the native data structure directly. Also we can do something about decreasing the time when we search the incremental list. An unexplored idea is arranging the incremental nodes in a binary search tree, so looking for an index there takes only logarithmic time. If we explore commutativity of sequences of operations of this type of data structures we find that they share the property that most operations commute, the only when they access the same index do they not commute. Detecting commutativity here is easier.

Performance for single-point access data structures have a similar setback. Time complexity for pops and dequeues increases depending on how many consequtive removal operations have happened before. This of course makes each pop the worst case time linear on the number of operations. This is obviously very bad since that operation natively takes constant time. Not only that, commutativity of the operations of a Stack in most cases is not possible, only sequences that leave the stack the same commute, and detecting that requires the execution of both sequences. Queues on the other part their commutativity depends on the size of the queue and the difference between dequeues and enqueues.

In the other hand linked data structures have a more apparent benefit using this model. Insertions and updates take constant time, and lookups don't increase their complexity. Commutativity for lists depends on what range of nodes are affected by the two sequences, if none of the nodes intersect then the sequences commute.

On thing we can see in several data structures is that the merging of versions of a data structure, in some cases should be very fast to do a merge. This comes because intuition tells us that we are operating on the same data structure. If operations don't conflict with each other, we should not need to re do operations. Although this statement, pretty much goes out the window if the only method to detect commutativity is to execute the operation sequences and then compare the two states.

Chapter 5

Conclusions

The main conclusion this thesis arrives is that this model is just the first step of utilizing persistent data structures in a transactional memory system. The problem of object synchronization is very unexplored using this approach. Using the incremental persistence model makes ahieving persistence much more easier and straightforward, but it has it's costs. The thing we must analyze as we try to experiment its use is what environment will it be more useful. By useful we mean that the utility we get overweights its costs.This utility are operations like having quick commits and rollbacks, and having local copies of data strucutres for transactions.

It is clear this model has better performance in short transactions, but we don't want it to be limited to those scenarios. More work is needed in order to minimize the overhead we have by utilizing this persistence model instead of regular concurrent data structures. A way to improve this is by modifying the model and having ad hoc solutions for each type of data structure. As we saw in the last chapter every data structure has unique behaviour when we try to postpone operations, simulating another data structure's behaviour with a list, is in some cases non-trivial and not efficient. An idea is to utilize something beside a list to log operations. For example using a binary search tree in case of the vector.

The merging model can also be improved, but to do that an ad hoc solution must be given for each data structure. This is simpler if the native data structure is a well implemented persistent data structure. For example in a linked data structure a merge of versions might me achieved by changing the version stamps of pointer fields. This wasn't studied in depth, because we wanted to use data structures already implemented in standard libraries. We didn't want to make data structures from scratch.

One big point this thesis purposely mention little about is operation sequence commutativity problem. This is because of the difficulty associated with the problem. The problem is if there is a way to determine if two operation sequences of the same version of a data structure commute, without having to execute both sequences and then compare states. The tentative answer is no, if we only have the incremental version list. My proposed idea is having a lock, similar to the abstract lock used in transactional boosting. This idea hasn't been explored yet, we only focused in this thesis to laydown the groundwork for an object synchronization model.

The importance of the work done in this thesis is that now we have something from which one can start studying persistent data structures inside a transactional memory system. Previously there was none, including a wide variety of persistent data structures. Also of great importance is that now we have one more purpose to study persistent data structures, given that better persistent data structures improve the performance of this model.

5.1 Results

The concrete results we deliver in this thesis are the following: We created and implemented two new never before seen persistent data structures. The persistent stack and persitent queue presented in chapter 2 are as far as I'm concerned have never been studied. Our implementation works well when full persistence is required, going back into past versions and creating new states. The important aspect is that doesn't require a complete overhaul of the original data structure, only the internal logic is overriden to use the versionlist.

The incremental persistent data structure is 2 in 1 model that adds persistence to data structures and a new object synchronization method for STMs. It also enables a fast commit and rollback operations for transactions that use this data structures. The peformance of this system is at the moment dependent on the size of the transaction.

5.2 Future Work

A big caveat of the incremental persistence model is trying to implement the version merge. And we are not speaking about the actual algorithms that merge two structures, what we mean is the determining and the detection of the pre-conditions needed to do a correct merge. The only precondition we have right now is commutativity, but maybe there is a weaker condition, one more simpler to detect. This work is going to be part of the contention manager that uses this model. Once we are able to do this, we can focus on having better and more efficient ways of merging data structures. Another thing that is the future, is improving the performance of the model, not only on short transactions. This has to be usable both scenarios. Ideas for this include better uses of the version list, and maybe not just restrict us to just a list. For different data structures different ways of logging operations.

One task that is pretty much not studied is the creation of normal persistent data structures. This will benefit us, because this model works best with fully persistent data structures. Most of the recent work done in this area has been non academic and has come from software engineers. One thing that look very important to study are the famous concurrent data structures like skip lists and concurrent red-black trees. Having a good persistent version of this data structures would be very useful.

Finally, one of the ideas that came up during the research, was instead of an incremental list that logs the operations, is using a repository system approach, with non persisten local copies. This would work by having transactions work on a local copy of the data structure and when it commits it merges the changes into the main persistent data structure branch. This would require very efficient version merging and conflict detection.

5.3 Acknowledgments

I would like to acknowledge the department of Computer Science of Brown University for all the acomodations and free meals they provided during the two month research visit. Specially I would like to thank Dr. Maurice Herlihy for guiding the bulk of the work, focusing on what is important and discarding the pointless work.

I would also like to acknowledge the help and input from the faculty of the Computer Science and Engineering Graduate Programm of the National Autonomous University of Mexico. Helping improve and clarify a lot of the algorithms seen in this thesis. Things like using a search tree instead of a list when we use incremental operations in a vector, were ideas from Dr. David Flores. They're critique helped in the shaping of the conclusions we arrived in this thesis.

A lot of the inspiration for this model came from the model talked about in this article *Conversion: Multi-Version Concurrency Control for Main Memory Segments*. Our approach evolved from the work done by this model.

Bibliography

[1] J. R. Driscoll, N. Sarnak, D. D. Sleator, R. E. Tarjan, *Making Data Structures Persistent* , Journal of Computer and System Sciences, Vol. 38, No. 1, 1989: pp 86-127.

[2] RichHickey. *Persistent Data Structures and Managed References*. http://www.infoq.com/presentations/Value- Identity-State-Rich-Hickey 2009. Online Presentation.

[3] Amos Fiat, Haim Kaplan. *Making Data Structures Confluently Persistent*. Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2001: pp 537-546

[4] M. H. Overmars. *Searching in the past, parts I and II*. Technical Reports RUU-CS-81-7 and RUU-CS- 81-9, Department of Computer Science, University of Utrecht, Utrecht,The Netherlands, 1981.

[5] Anton Tayanovskyy, Adam Granicz. *Global Transactions in the Cloud with Persistent Data Structures*. IntelliFactory. 2010.

[6] Gilbert, S., Lynch, N.: *Brewer's conjecture and the feasibility of consistent, available, partition tolerant web services*. ACM SIGACT News 33 (2002) 51–59.

[7] Moore, G.E. , *Progress in digital integrated electronics*, Electron Devices Meeting, 1975 International. Vol. 21. IEEE, 1975: pp 11 -13.

[8] Ikka Tuomi, *The Lives and Death of Moore's Law* First Monday Volume 7, Number 11, University of Illinois (2002).

[9] G. M. Amdahl. *Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities*. AFIPS Conference Proceedings, pp 483–485, 1967.

[10] Nir Shavit, Maurice Herlihy, *The Art of Multiprocessor Programming*, Morgan Kaufman Publishers, 2008.

[11] Martin Odersky, Lex Spoon, Bill Venners *Programming in Scala, First Edition* http://www.artima.com/pins1ed/index.html , 2008.

[12] R. Guerraoui, M. Kapalka, *On the Correctness of Transactional Memory*. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (pp. 175-184). ACM, 2008.

[13] A. Dragojevic, R. Guerraoui, M. Kapalka *Stretching Transactional Memory*. ACM Sigplan Notices, vol. 44, no. 6, pp. 155-165. ACM, 2009.

[14] M. Herlihy, E. Koskinen *Transactional Boosting: A Methodology for Highly-Concurrent Transactional Objects*. Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 207-216. ACM, 2008.

[15] T. Merrifield, J. Eriksson *Conversion: Multi-Version Concurrency Control for Main Memory Segments*. Proceedings of the 8th ACM european conference on Computer Systems, EuroSys, vol. 13. ACM 2013.

[16] K. Krukow *Understanding Clojure's PersistentVector implementation*. http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/ , 2009. Blog Post.

[17] N. Shavit, D. Touitou *Software transactional memory*, Springer-Verlag Distributed Computing Journal 10.2l, 1997: pp 99-116.

[18] M. Herlihy, J.E. Moss, *Transactional memory : architectural support for lock-free data structures*, Proceedings international symposium on computer architecture '93, pp 289-300. 1993.

[19] David G. *Optimal search in planar subdivisions*. SIAM Journal on Computing 12, 1983

[20] Virendra Marathe, William Sherer, M. Scott, *Design tradeoffs in modern software transactional memory systems*. Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems, pp. 1-7. ACM, 2004.

[21] M. Herlihy, V. Luchangco, M. Moir, W. Scherer *Software transactional memory for dynamic-sized data structures*.In Proceedings of the twenty-second annual symposium on Principles of distributed computing, pp. 92-101. ACM, 2003.

[22] P. Dietz D. D. SLEATOR *Two algorithms for maintaining order in a list*, Proceedings, 19th Annual ACM Symp. on Theory of Computing, pp. 365-372. 1987.

[23] K. Krukow *Understanding Clojure's PersistentHashMap*. http://blog.higher-order.net/2009/09/08/understanding-clojures-persistenthashmap-deftwice/ , 2009. Blog Post.

[24] M. Herlihy and J. M. Wing. *Linearizability: a correctness condition for concurrent objects.* ACM Transactions on Programming Languages and Systems(TOPLAS). 12.3, pp: 463-492. 1990.