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

Consistent State Software Transactional Memory

Gonçalo Vasco Trincão Bento da Cunha

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

Lisboa
(2007)

This dissertation was prepared under the supervision of
Professor João Lourenço,
of the Faculdade de Ciências e Tecnologia,
Universidade Nova de Lisboa.

To my mother, my father and my sister.
To Luzia.

[This page was intentionally left blank]

Acknowledgements

I would like to, first of all, show my appreciation to my supervisor, Professor João Lourenço for his guidance and support. Even with his very tight time schedule, he always found the time to discuss the problems and progress of the thesis. I also appreciate his friendship and his remarkable human skills.

To Dave Dice, Ori Shalev and Nir Shavit who sent us the TL2 source code, which is the base of this thesis.

To my good friends Américo Rio and José Reis, with whom I have shared my ups and downs during our usual coffee breaks and Friday lunches.

To my close family, my mother, father and sister, my very special thanks for their love, support and confidence in me.

To Luzia, who always stood by me and gave me the strength to embrace this project.

[This page was intentionally left blank]

Summary

As the multicore CPUs start getting into everyone's computers, concurrent programming must start covering, not only the scientific and enterprise applications, but also every computer application we all use in our daily lives.

Since the introduction of software transactional memory [ST95, HLMWNS03], this topic has had a strong interest by the scientific community as it has the potential of greatly facilitating concurrent programming by hiding the concurrency issues under the transactional layer.

This thesis builds on the TL2 STM engine [DON06], which is one of the top performing to date. We have explored different design alternatives focusing on performance and safety. With our research we have achieved performance improvements and better safety properties of the engine. We have also achieved a much better understanding of the design alternatives and their impacts.

During the course of this thesis we have come across several tough concurrency bugs and we have created a list of testing patterns, which proved to be useful in finding and reproducing several problems.

This thesis describes the cutting edge of STM engine technology, elaborates on the design of a new STM engine and reports on the experimental results obtained.

[This page was intentionally left blank]

Sumário

Agora que os processadores multicore começam a ser comuns até nos computadores pessoais, a programação concorrente precisa de começar a contemplar, não apenas as aplicações científicas e empresariais, mas também todas os programas que usamos no nosso quotidiano.

Desde a introdução da memória transaccional por software [ST95, HLMWNS03], este tópico tem sido alvo de um forte interesse pela comunidade científica. Isto acontece porque tem o potencial de facilitar consideravelmente o desenvolvimento de programas concorrentes, escondendo as suas dificuldades na camada transaccional.

Esta tese baseia-se no motor transaccional TL2 [DON06], que é um dos protótipos de investigação com melhor performance até à data. Durante este estudo explorámos arquitecturas diferentes para o TL2, com o foco na sua velocidade de execução e propriedades de segurança, conseguindo a estes níveis melhoramentos em relação ao protótipo original. Obtivemos igualmente um entendimento mais detalhado das alternativas de implementação dos motores transaccionais, bem como dos seus impactos a nível de funcionalidade e desempenho.

Durante a investigação deparámo-nos com vários problemas de concorrência complexos, o que nos levou a sintetizar uma lista de padrões de teste que provaram ser úteis na tarefa de encontrar ainda mais problemas no protótipo.

O presente trabalho expõe alguns dos tópicos do que se faz hoje em dia no que diz respeito à investigação em sistemas de memória transaccional, descreve em detalhe as alterações que fizemos ao TL2 e apresenta os resultados experimentais obtidos.

[This page was intentionally left blank]

Contents

[This page was intentionally left blank]

List of Figures

Chapter 1

Introduction

This Chapter shows the motivation for transactional memory, highlights the main contributions of the thesis and outlines the structure of this work.

1.1 Introduction

On the past decades, the density of transistors in computer chips has grown in an exponential way. Gordon Moore, back in 1965, has observed that the number of transistors per square inch doubles every 18 months and the rate of growth has been relatively stable since then. For a long time the CPU clock frequency has accompanied this trend, growing side by side with the number of transistors. However, since the appearance of CPUs with GHz clock speed, the climb rate of clock frequency has been slowing down, even though the number of transistors is still rising.

This situation where the number of transistors increases but the clock speed does not, has left room for the CPU manufacturers to add more instruction sets (e.g. SSE, VT) and to increase the number of cores each chip has. However, regular single threaded applications can't take advantage of the increasing number of cores and developers must change the way applications are implemented to keep providing more complex features without decreasing its overall performance.

To take advantage of this new reality, applications must be concurrent. There is, however, a natural resistance from many programmers to create concurrent programs because of the additional effort and complexity it brings [Sut05]. In part this resistance is due to the lack of a suitable framework to deal with concurrency. The usual synchronization constructs (locks and condition variables), while simple on the paper, may become unpredictable and error prone in complex systems. Coarse grained locks on large data structures do not scale and fine grained locks are prone to difficult problems in larger systems, such as deadlocks and priority inversion. Also locks and condition variables do not compose [HMPJH05]. Software Transactional Memory (STM) is gaining popularity, as it helps to solve some of these problems. It can prevent deadlocks and priority inversions and it is composable.

1.1.1 Transactional Model

Transactions are known for a long time on the database community. They have been very well studied, have standard semantics and are a standard feature of database management systems.

The transactional model is a good paradigm to deal with concurrency because it can hide most of its complexity. With the transactional model, a programmer only has to define which section of its code is a transaction. From a developer perspective, the transactional model ensures that the section runs "as if" it was running alone in the system. The transactional layer is responsible to manage the concurrency of the several transactions and avoid concurrency hazards.

The transactional model defines four properties [Dat94]:

- *Atomicity.* The transactions are atomic, they have a "all or nothing effect". Either all its actions take effect or none of them does.
- *Consistency.* Transformations preserve consistency, meaning that the global state moves from one consistent state to another.
- *Isolation.* Transactions are isolated from each other, even if they run concurrently. For any two distinct transactions T1 and T2, T1 may see T2 updates, or T2 may see T1 updates, but not both at the same time.
- *Durability.* After a transaction commits the changes will persist even if there is a system crash.

A transactional system with these properties allows programmers to continue using the sequential programming mental scheme with minimal changes. They basically don't have to care about concurrency control as the transactional model deals with it.

A sample of a transaction is shown on Figure 1.1. The transaction occurs between BeginTransaction and Commit. After the commit returns successfully, the application knows that all its effects have taken place. Before the commit occurs, the transaction may still be undone.

If, instead of committing, the transaction had aborted (Figure 1.2), the application would know that none of its updates had taken place.

```
BeginTransaction()
DoAction1()
DoAction2()
DoAction3()
CommitTransaction()
```

Figure 1.1: Simple Transaction

```
BeginTransaction()
DoAction1()
DoAction2()
DoAction3()
if(something)
    CommitTransaction()
else
    AbortTransaction()
```

Figure 1.2: Transaction that may abort

The transactional model ensures that: either all or none of *DoAction1*, *DoAction2* and *DoAction3* are performed; the consistency properties of the shared state are preserved after the transaction finishes; even if there are other concurrent transactions running, the transaction runs as if it was alone on the system; and after the transaction commits, its updates persist even if there is a system failure, in other words, the changes are written to a persistent storage.

1.1.2 Transactional Memory

Transactional memory aims at replacing the traditional concurrency control mechanisms used in most programming languages: locks, condition variables, semaphores, etc, are among the most commonly used.

Naturally, the transactional memory and database requirements are different [HM-PJH05], e.g., the consistency and durability properties are more relaxed and there is a much stronger performance requirement on memory transactions.

The durability requirement implies that changes to the shared state have to be persistent after the transaction commits, meaning they have to be written to a persistent storage. This requirement is stronger than necessary for transactional memory (locks and condition variables also don't have this property) and it brings a big performance penalty, therefore it has been removed from the transactional memory requirements.

The consistency requirement is well suited for databases, which have a standard well known data organization—tables, rows, cells, etc. and have standard integrity rules, like referential integrity. Memory transactions, on the other hand, can't have consistency verification in the same way as database transactions because the data structures are much more flexible—the concepts of primary and foreign keys are not used in memory structures. This requirement has been dropped from all STM engines we know of.

To summarize, the main requirements for transactional memory are: Atomicity, Isolation and Performance.

The need for performance has already made some authors propose to drop some interesting features. Early STM engines [HLMWNS03,HF03] were implemented using non-blocking synchronization [HLM03] and, as a consequence, they had the additional feature of supporting arbitrary thread failures, however later tests with lock based STMs ([Enn06,DS06,DON06,SATH$^+$06]) showed that the former were less performant. So more and more STM proposals are based on blocking techniques.

Also, the performance requirement has led to the use of optimistic concurrency in detriment of read/write locks. Optimistic concurrency allows transactions to run without synchronization and, on commit, validate if the transaction conflicted with others. However, transactions running with optimistic concurrency control, may exhibit strange behaviors due to conflicts with other transactions (like dereferencing invalid pointers), which must be detected and handled by the STM engine.

This work is based on the TL2 STM engine. TL2 is a top performing engine, with interesting features, like the integration with standard non-garbage collected memory languages such as C and the early detection of inconsistent states.

1.2 Contributions of this Thesis

The focus of this thesis is the STM engine technology. After an initial survey of the existing STM engines, we have extended TL2 by implementing new features; we have implemented and evaluated several performance oriented design options; and we have created testing tools to ease problem finding in the STM engine.

The features implemented in the STM engine can be summarized as:

- *User Called Aborts* — For the application to programmatically abort a transaction.
- *Automatic Transaction Retry* — To automatically retry a transaction that finds itself in an inconsistent state. This changes the semantic of the commit from "at most one" to "exactly one" successful execution.
- *Transaction Nesting* — To enable the partial rollback of a transaction and to increase reusability of the software stack by allowing transactions to be started at several layers.

Related to performance, we have made several changes to the original TL2. We can outline:

- We have achieved a better performance by increasing the lock granularity to the object level and thus reducing the overhead by having fewer lock acquisitions.
- We have decreased the overhead also by reducing the number of instructions executed on the most used transactional method (load).
- We have introduced a novel modification to TL2's algorithm to decrease the cache coherency overhead by reducing the number of accesses to shared state.

Other contributions of this thesis were:

- We have increased the safety properties of the original STM engine. This novel modification ensures that the transactional code always sees a consistent view of memory. The original prototype did not ensure a consistent view when a piece of allocated memory was leaving the transactional space (being free'd).
- We have built a lightweight tracing engine. A specialized lightweight tracing engine is indispensable to debug the STM implementation because the lock granularity on a memory transaction is very fine grained and the number of interleavings is huge, making it virtually impossible to use traditional debuggers to find STM implementation bugs. The novel tracing engine built on this thesis records

events in order of occurrence and does it with an exclusive section of a single CPU instruction.

- In addition to the changes made on the prototype, we have proposed a novel classification scheme for the consistency of the transaction memory view.

- Finally, we are also the first to present a type of hazard that may occur on lock based STMs that use the undo log strategy. This hazard may occur because transactional writes may take place when a transaction is in an invalid state.

1.3 Outline of the Dissertation

This dissertation is divided in the following chapters:

- *Chapter 1.* This Chapter has shown the motivation for the use of transactional memory, it has highlighted the main contributions of the work and outlined the structure of this thesis.

- *Chapter 2.* This Chapter shows the main features associated with transactional memory alongside with the main design approaches.

- *Chapter 3.* This Chapter describes the TL2 STM engine, in which this work is based. It also describes in detail the improvements we to TL2 which originate this thesis.

- *Chapter 4.* This Chapter describes some of the problems found while implementing the changes in the STM engine and synthesizes the tests used to reproduce those problems.

- *Chapter 5.* This Chapter describes the tests and experiments made to evaluate the prototype and shows the achieved results.

- *Chapter 6.* This Chapter summarizes the main results of this investigation and brings out some pointers for future directions.

Chapter 2

Software Transactional Memory

This Chapter shows the main features associated with transactional memory alongside with the main design approaches.

2.1 Introduction

Transactional memory aims at providing a higher abstraction level for concurrent programming. It aims at replacing locks and condition variables as the concurrency control mechanisms, providing a semantics which makes it easier for the programmers to reason about. One of the most important properties provided by transactional memory is the serializable isolation level, where the concurrent events occur "as if" they happened sequentially. Another very important property is atomicity, which means that either all the effects of a transaction are visible (transaction commits) or none of them are (transaction aborts).

Taking these properties as a starting point, there have been a flurry of design approaches and of proposed new features. In the following sections we describe some of these new features proposed in the literature and we outline several design approaches, discussing their main advantages and drawbacks.

There have been several syntaxes used in STM prototypes. The first approach was used in [HLMWNS03] and it uses methods defined in a library to control the transaction progress (Figure 1.1 of Section 1.1.1). The second approach was used in [HMPJH05] and uses a new keyword *atomic* to represent a transaction, the authors call it an atomic block (Figure 2.1). The third approach used in [HLM06], places the transaction code inside a function or method, the transaction function is called via a transaction manager (Figure 2.2). [1]

```
atomic{
   x++;
   y++;
}
```

Figure 2.1: Transaction defined by an atomic block

Despite the syntax used, the guarantee that is always provided is: If the transaction returns successfully, then all the actions of the transaction were executed and the transaction did not interfere with other transactions. Otherwise, if the transaction did not return successfully, then none of its actions have taken effect.

[1]The examples shown in this thesis will use pseudo-code in a C/Java like fashion.

```
int TransactionXPTO(){
    a=TxLoad(x);
    b=TxLoad(y);
    return 1;
}

// ...
CallTransaction(TransactionXPTO);
```

Figure 2.2: Transaction defined by a function or method

2.2 Features

In addition to these basic features—atomicity and isolation, some others have been proposed to ease the developers job and extend the number of situations where transactional memory can be used.

2.2.1 Conditional waiting

The conditional waiting functionality is available when using *condition variables* and it allows a thread to wait until a certain condition is true. This idea may as well be applied to transactions [HMPJH05].

One example is crossing a one-way bridge, which allows for only one car at a time. This example designed using *condition variables* is shown on Figure 2.3.

```
Init(){
  NumCars=0;
}

Synchronized EnterBridge(){
  while (NumCars >= 1) {
    wait();
  }
  NumCars++;
}

Synchronized LeaveBridge(){
  NumCars --;
  NotifyAll();
}
```

Figure 2.3: One-way Bridge using condition variables

Plain transactions are not enough to implement efficiently the conditional wait functionality, as they don't have waiting primitives.

To encode this logic using transactions it is necessary to implement a waiting primitive. Two proposals have shown up: one with guarded atomic blocks (Figure 2.4) used in [HF03] and another with a new *retry* primitive (Figure 2.5) used in [HMPJH05].

When using a guarded atomic block, the execution is deferred until the condition is true. When using the *retry* primitive, the transaction is executed as usual, but when a *retry* primitive is called the transaction is rolled back and only restarted when any of the variables read in the previous execution of the transaction have changed. Using the example in Figure 2.5, if a transaction retries, it will be restarted only when the variable NumCars has changed (by a commit of another transaction). The *retry* primitive is more flexible than the guarded atomic blocks as the logic before a *retry* is issued can be more complex than a simple expression, e.g., it is not easy to use atomic blocks with guards to wait for all the elements of an array to have a specific value.

```
Init(){
   atomic{
      NumCars=0;
   }
}

EnterBridge(){
   atomic(NumCars<1){
      NumCars++;
   }
}



LeaveBridge(){
   atomic{
      NumCars--;
   }
}
```

Figure 2.4: One-way Bridge using atomic blocks with guards

```
Init(){
   atomic{
      NumCars=0;
   }
}

EnterBridge(){
   atomic{
      if(NumCars<1)
         NumCars++;
      else
         retry;
   }
}

LeaveBridge(){
   atomic{
      NumCars--;
   }
}
```

Figure 2.5: One-way Bridge using atomic blocks with *retry* primitive

2.2.2 Composability

Composability is an important property because it facilitates the creation of several software layers, where the lower layers hide the complexity of their implementation from the upper layers.

An example where locks are not composable is a concurrent list implementation with three access methods: add, delete and get [HMPJH05]. These methods are internally synchronized and therefore enough to manage the list. However, if one application has two list instances and needs to move an element from one list to the other, without having visible the intermediate state where the element is in neither of the lists, these three methods are not enough. The list implementation needs to expose the lock and unlock methods to allow this operation. However, the abstraction is broken as the list must expose its internal lock and unlock methods.

Memory transactions can aid in achieving better composability by using the extensions described ahead.

2.2.3 Composing alternatives in transactions

Some situations require a way to express alternatives in transactions. Let's say there is already a library representing a transactional queue and a given thread can use the send/receive API calls to interact with the queue. The receive method blocks when the queue is empty.

Let's suppose one given thread needs to wait for data on two or more transactional queues. The problem is that since the receive method is blocking, if one queue is empty, the thread will block and will not be able to receive from the other queue even if it has data.

The basic requirements for this feature are: to be able to wait for several events; the absence of active polling; and that they are composable, as we want to reuse the existing receive method from the transactional library API.

The result as proposed by [HMPJH05] is a new primitive that expresses an alternative and it is called *orElse*. It is used in conjunction with the *retry* primitive and it allows the transaction to execute an alternative path if the previous path did a *retry*. On the example of Figure 2.6, the method *ReceiveFromEither* tries to receive from one of the queues, if the first receive issues a *retry*, it rolls back and tries from the second queue. If the second queue is also empty, the whole transaction is rolled back and restarted when any of the variables read have changed.

```
Receive(Queue q){
  atomic{
    if(NumItems==0){
      retry;
    } else {
      // return data from queue
    }
  }
}

ReceiveFromEither(Queue q1, Queue q2){
  atomic{
    do {
      d = Receive(q1);
    } orElse {
      d = Receive(q2);
    }
  }
}
```

Figure 2.6: Composing alternatives in transactions

2.2.4 Transaction nesting

A feature long known on databases is transaction nesting, which enables the creation of sub-transactions (transactions inside other transactions).

The nesting feature is imperative to enable composition of transactions. When a software system is composed of several layers; the lower layers need to be atomic; and the upper layers need to re-use the operations form the lower layers, then the transactions need to be composable. In other words, the upper layers are transactions and need to reuse the lower layer operations, which are also transactions.

An example is a list implementation with the same three methods: add, delete and get. The operation move, which transfers an element from one list to another, should be composed of a delete followed by an add, but in an atomic step. As such, it needs to start a transaction and, when it calls the methods delete and add, they also start a (sub-)transaction.

There are three types of transaction nesting [ALS06]: *flat nesting*, *closed nesting*, *open nesting*. Their main characteristics are:

- *Flat Nesting* — The effects of the sub-transactions are only visible to other transactions when the main transaction commits. This mode works by "inlining" the sub-transactions with the main transaction. When a sub-transaction aborts the whole transaction is rolled back, including the main transaction.

- *Closed Nesting* — The effects of the sub-transactions are also only visible to other transactions when the main transaction commits. In this mode, the updates of the sub-transactions are recorded separately from the main transaction and they only become part of the main transaction when the sub-transaction commits successfully. When a sub-transaction aborts, only the effects of the sub-transaction are undone.

- *Open Nesting* — The sub-transactions are considered to be independent of the main transaction and their effects are visible to other transactions immediately after the sub-transaction commits successfully. The aborts of the sub-transaction don't cause aborts on the main transaction and even if the main transaction aborts, the sub-transaction effects will not be undone.

2.2.5 Irrevocable actions

Since transactions may abort at any time (by explicit call from the application or due to internal engine issues) it is important that every action made within the transaction can be reversible. However, some types of actions are not reversible, like writing to a socket or deleting a file, consequently they should not be allowed. STM engines to date are only capable of reversing memory operations, therefore I/O operations must not be used inside a transaction, otherwise the operation may be executed multiple times, or it may be executed even if the transactions aborts.

On the example of Figure 2.7, the *PowerOffRemoteServer* function, powers off some server on the network. If the execution was allowed, and if the "some condition" was false, the transaction would rollback, but the remote server would have already been shutdown by the transaction.

It is not easy for an STM engine to prevent I/O operations (if not even impossible in some programming languages), this is why most STM engines leave this responsibility to the programmer. One exception is the STM engine STM Haskell [HMPJH05]. It is implemented in the programming language Concurrent Haskell, which has a very expressive type system and it splits all actions in STM actions and IO actions. The compiler statically prevents IO actions from running inside memory transactions.

2.3 Design approaches

We now describe some concepts and design approaches which have been reported in the literature. These are some of the fundamental building blocks of the STM engines—

```
TransactionBegin();
  PowerOffRemoteServer(ServerName);
  if(some condition)
    TransactionCommit();
  else
    TransactionAbort();
```

Figure 2.7: Irrevocable action inside transaction

the synchronization techniques used, the granularity of the locks, the algorithms of the transaction log, the contention management strategies, etc.

2.3.1 Blocking vs. Non-Blocking Synchronization Techniques

When a thread executes a transactional access (read or write), it uses the primitives of the STM engine to mediate the access. The access must be mediated by the engine to allow the synchronization of the concurrent accesses to shared data-structures.

The synchronization techniques used by STM engines may be based on blocking or non-blocking synchronization techniques. Traditional blocking synchronization (or lock based) usually have better performance. The alternative non-blocking synchronization technique guarantees progress of the system as a whole independently of the interleavings or any thread failures.

Blocking Synchronization

Blocking synchronization techniques use locks to prevent other threads to access the same data. Locks are acquired before any access to the shared data and released after the operation is complete. Threads that find a locked item must wait until the lock is released to access the data, usually contention management algorithms are used to avoid deadlock and starvation problems (described ahead). However, if a thread owning a lock fails, the lock can no longer be (safely) released and the system may freeze.

Non-Blocking Synchronization

Non-blocking synchronization techniques work by making a private working copy of the items and, when the changes are finished, atomically swap the working copy with the old one. The operations must be restartable because threads may collide with each other when accessing the data.

STMs using non-blocking synchronization inherit the advantages of this technique: they don't suffer from deadlocks and are resilient to thread failures. If a thread fails within a transaction, its effects are discarded without affecting the global state. On the contrary, with blocking STMs, if there is an unforeseen exception or thread failure while holding the locks then: either the locks stay acquired, hence making the system halt; or the locks are released by a manager but the dirty data will become visible to the other threads.

Non-blocking synchronization algorithms can have either of these three progress guarantees [HLM03, HF]:

- *Obstruction freedom* "is the weakest guarantee: a thread performing an operation on the data structure is only guaranteed to make progress so long as it does not contend with other threads for access to any location (...). This requires an out-of-band mechanism to avoid livelock; exponential backoff is one option."
- *Lock freedom* ensures that at least one thread always makes progress hence "the system as a whole makes progress, even if there is contention. (...) This is sufficient to prevent livelock, although it does not offer any guarantee of per-thread fairness."
- *Wait freedom* "adds the requirement that every thread makes progress, even if it experiences contention." "It ensures that every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads."

A *wait free* algorithm is always *lock free* and a *lock free* algorithm is always *obstruction free*, but not the other way around. Independently of the progress guarantees, non-blocking synchronization mechanisms are never subject to deadlocks.

An example of how DSTM, a non-blocking STM implementation [HLMWNS03], atomically updates the objects on commit is shown in Figure 2.8. Every transactional object is a pointer to a Locator object. The Locator has a pointer to the status of the transaction updating the object; an old and a new copy of the object data. When the transaction is active, the data on the *old* pointer is the most recently committed data and the *new* pointer is the working copy. When the transaction commits, it updates the status from *active* to *committed*, from this moment on the most recently committed data is under the *new* pointer.

Synchronization Techniques on STMs

In the context of transactions there are several arguments defending the usage of both blocking or non-blocking approaches [HLMWNS03, Enn06, DS06]. However there has

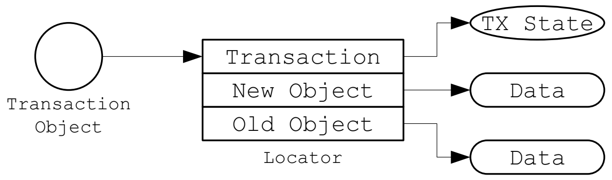


Figure 2.8: DSTM Object Structure

been a shift towards blocking STMs. While early STM implementations started using the lock-free property [ST95], later, implementations using obstruction-free property [HLMWNS03] were proposed and lastly the implementations using blocking synchronization [Enn06, DS06, DON06, SATH$^+$06]. This shift is due to the search for simpler implementations and better performance—and blocking STMs are closer to those goals [Enn06, DS06].

2.3.2 Transactional data and granularity

It is important that the variables read and written by transactions are not simultaneously accessible by non transactional code. Otherwise harmful interleavings could occur due to unsynchronized access to these variables.

We have seen two strategies for dealing with this issue: the first is to leave to the programmer the responsibility of avoiding non transactional accesses to transactional variables, this is the strategy of [Enn06, DS06, DON06]; the second is to define a special type of transactional variables, which may only be accessed by transactions, this is the strategy of [HLMWNS03, HLM06].

Another issue when defining transactional data is the size of its granularity. In other words, the engine needs to define the smallest amount of data a transaction may access. The main strategies are block level (word-based STMs) and the object level (object-based STMs). With block level granularity, the transactional data grain has the size of a memory word or the size of a cache line. With object level granularity, the transactional data grain has the size of an object.

Word based

Word based STMs have been first proposed by Harris and Fraser in [HF03]. These STMs have more room for concurrency as the granularity is finer. In lock based STMs each word has its own lock and each lock is acquired independently of the others, therefore collisions are less frequent. However the overhead is high because the transactional API must be called once for every word the transaction accesses.

For instance, one transaction that needs to access all the fields of a forty byte object (assuming word size of four bytes) it must make ten transactional calls to read the whole object.

Object based

Object based STMs [HLMWNS03] have one metadata structure shared among all the fields of the object—they have a coarser granularity than word based STMs. Simultaneous accesses from different transactions to different fields of the same objects will conflict. The transactional API is called once for every object (instead of every field).

Object based STMs are less suitable for accesses to transactional variables that are not objects—like single variables and arrays. Word based STM, on the other hand, can access single variables and access arrays word-by-word.

2.3.3 Concurrency Control

There are two main types of concurrency control mechanisms used on blocking STM engines [SATH$^+$06]: read-write locks and versioned write locks. Read write locks are one of the most commonly used concurrency control methods and there are several implementations available for most programming languages (e.g., pthread_rwlock for C; ReentrantReadWriteLock for Java).

Versioned write locks, on the other hand, use an optimistic concurrency control for reads and revocable two phase locking for writes. Their main advantage is the fact that reads do not need to acquire a lock, therefore they don't need to write to the shared lock. This is an advantage as it causes less cache coherency traffic on the shared bus and less remote cache invalidations(hence less cache misses).

Read Write Locks

Read write locks have two types of lock acquisition, one for read and another for write. Readers acquire the lock in shared mode before reading the value; writers acquire the lock in exclusive mode before writing to the value. Reads can be shared among other reads and writes are exclusive. The main disadvantage of this method is the fact that it needs to acquire a lock when reading, this causes extra cache invalidation traffic on the bus. Also upgrade from read to write locks are expensive [SATH$^+$06] as the writers need to wait for the readers to finish.

Versioned Write Locks

Due to performance reasons, most lock based STM systems use optimistic concurrency control for reads with versioned write locks [Enn06, DS06, DON06, SATH$^+$06]. Each

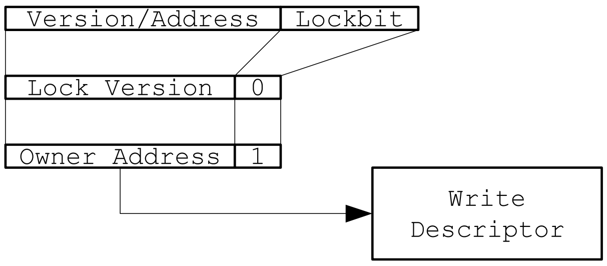


Figure 2.9: Versioned write lock

variable has a version number that is incremented for every commit updating it. When a transactional read is performed, the version number of the variable is recorded in a transaction local data structure (the read set). On commit, it is verified if the versions of the variables read have changed by comparing the actual version number with the one recorded on the transactional read. If any of the version numbers have changed, the transaction is in invalid state and must be rolled back, otherwise it may commit. This prevents non-serializable orderings from committing.

The implementation of versioned write locks is usually made using a single word (32 or 64 bits depending on the architecture) to hold the field indicating whether the lock is held or not; the version number; and a pointer to the transaction holding the lock (Figure 2.9).

The field (lockbit) indicating the status of the lock (acquired or not) uses a single bit. The remaining bits (31 or 63 depending on the architecture) either hold the lock version number (if the lock is released) or hold a pointer to a descriptor of the transaction holding the lock. The performance advantage of the versioned write locks is due to the fact that readers just read the lock value, they don't have to change the lock status (like it happens with read write locks).

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

2.3.4 Recovery Strategies

The STM engine needs to record information about every update that each transaction tries to make. In case a transaction commits, it may need to apply the changes and when a transaction aborts it may need to undo the changes previously made. Depending on the engine implementation, it may also need to register every read on a transaction read log to validate the transaction on commit. The transaction update log is also called write set and the transaction read log is also called read set.

On blocking STMs there are two main recovery strategies undo log and redo log.

Their behavior differs mainly in the way that the update log is managed.

Undo log

In the undo log strategy, when a write occurs, the shared variable is locked (to prevent other transactions from accessing the dirty data) and the shared value is replaced with the new tentative value. The previous value is recorded on the transaction private undo log. This strategy is also called "in-place update".

Reads check whether the variable is locked. If it's not, they record the lock version number on the read set and access the variable. If it is locked, the transaction may either wait or abort.

When a commit is performed, the engine validates the read set by checking if all reads still have the same lock version. If it is still valid the locks are released and the undo log is discarded, otherwise the transaction aborts.

When an abort is performed, the shared variables are re-written with the values kept in the undo log and locks are released, therefore restoring the data to the state it had before the transaction began.

Since these transactions change the shared variables as soon as a write is performed, the write locks must be acquired immediately. Some authors [DS06] call this strategy "encounter locking" mode.

Redo log

In the redo log strategy, when a write occurs, the shared variable remains unchanged and the new tentative value is written on a transaction private write set. This strategy is also called "out-of-place update".

A transactional read can't look directly to the shared variable as it may have already been written within the current transaction; instead it must first look aside into the transaction write-set. If the variable is not on the write set, the read must check whether the variable is locked, if it's not locked, it accesses the variable directly and records its version number on the read set, otherwise the transaction must wait or abort.

The advantage of this technique is that lock acquisitions may be delayed until the commit phase. When a commit is performed, the shared variables are locked and, if the read set is valid (the version numbers haven't changed since the actual read), their values are replaced with those stored in the write set. If it is not valid the transaction aborts.

When an abort is performed, only clean-up work needs to be done.

Some authors [DS06] call this strategy, where the locks are acquired at commit time, "commit time locking" mode.

Undo vs. Redo log

The undo log strategy has the main advantage of speeding up reads, because they look directly into to the shared variable, they don't need to look aside into the write set. However, locks stay acquired for a longer period, which lasts from the first write on the variable until the transaction commits.

The main advantage of the redo log strategy is that locks are acquired only when performing the commit—while the values are being copied from the redo log to the shared location. This minimizes the lock hold time and therefore the overall contention. Reads, however, are slowed down by having to look aside to the write set and check if a previous write by the current transaction has updated the variable.

2.3.5 Transaction States and Global Version Clock Algorithm

When using versioned write locks a number of hazards may occur due to the existence of invisible reads. A transaction may read a dirty value—one that is being changed by another transaction; a transaction may not see a valid memory snapshot because the state changed between two consecutive reads; and it may see an old snapshot. When such things happen it means the transactions have collided and at least one of them has entered a state where it cannot commit and therefore it must undo all the changes it made.

One example is shown in Figure 2.10, where the system has an invariant of $x == y$ but due to the interleaving of T2, transaction T1 does not observe the invariant on step 8.

	T1	**T2**
	// invariant x==y	
1	**atomic** {	
2	_a=x;	
	....	....
3		**atomic** {
4		x ++;
5		y ++;
6		}
	....	....
7	_b=y;	
8	*// invariant x==y*	
	// is not observed	
9	}	

Figure 2.10: Transaction sees inconsistent state.

We divide transaction state into three categories: *updated consistent*, *obsolete consistent* and *inconsistent*.

Updated Consistent State

A transaction running in an updated consistent state is one in which all reads have returned the latest committed values and these values haven't been updated since the read. Transactions that try to commit with an updated consistent memory snapshot will succeed.

Obsolete Consistent State

A transaction is in a obsolete consistent state if it has an outdated memory snapshot. It may happen because another transaction updates one of the read variables. A transaction running in a obsolete consistent state has a correct behavior but it has a past view of the system and therefore it will fail to commit. Read-only transactions may commit with an obsolete consistent state because they are only retrieving information from the system.

Figure 2.11 shows an example of a transaction running into an obsolete state. On step 6, transaction T1 is consistent but obsolete, commit will fail.

	T1	**T2**
1	**atomic** {	
2	_a=x;	
	....	....
3		**atomic** {
4		x ++;
5		}
	....	....
6	_b=z;	
7	// update something	
8	}	

Figure 2.11: Transaction in consistent but obsolete state.

Inconsistent State

A transaction is in an inconsistent state if the read values do not correspond to a valid memory snapshot. It may be due to reading dirty values or read-after-write hazards e.g., two consecutive reads to the same variable returning different values because it was updated by another transaction. Transactions in this state may have unpredictable behaviors because, since the system state is not consistent, all program invariants may be broken. All inconsistent transactions will eventually abort.

Concurrency Control and Transaction States

When using read write locks (assuming a strict two phase locking model), a read is guaranteed to always have an updated consistent state and the read variable can't be changed until the commit—the transaction has acquired a read lock and no other transaction can update the value while the read lock is being held.

When using versioned write locks there is no such guarantee—transactions may read variables while they are being written and therefore some hazards may occur. While the transaction is being run, the loads may see dirty variables and/or see inconsistent states as a result of unfortunate interleavings. Although these transactions will eventually abort, they may suffer erroneous behaviors while running—they may enter endless loops in the middle of the transaction, dereference invalid pointers, have divide by zero exceptions, fail assertions that otherwise would be valid if the transaction was being run in a consistent state, or even worse, provoke memory corruption. Such transactions are invalid and must be aborted—however the abort may happen at a very later stage in the transaction.

Figure 2.12, inspired on [HF03] shows an example of a system that has an invariant $x = y$. Transaction T1 is interleaved with T2 and reads different values of x and y, as a result it observes an invalid state and enters an infinite loop.

	T1	T2
	//x=0 and y=0	
1	**atomic** {	
2	_a=x;	
	. . . .	. . . .
3		**atomic** {
4		x ++;
5		y ++;
6		}
	. . . .	. . . .
7	_b=y;	
8	**if**(_a != _b)	
9	**while**(true){}	
10	}	

Figure 2.12: Infinite loop caused by improper ordering. Transaction T1 is inconsistent.

Another potentially hazardous behavior is memory corruption, which may have effects outside of the transactional space and the results may be undo-able. One example is shown on Figure 2.13, where an application has for transactional variables: a pointer to a dynamic size array (*main_table*), the array itself and an integer specifying the array size (*main_table_size*). Let's suppose a STM engine for a C like programming language, which makes the writes in-place and allows transactions to run in inconsistent states. One transaction is updating one of the array elements and another is switching the array to another one of a smaller size.

	T1	**T2**
	// main_table_size =100	
1	**atomic**{	
2	_size=main_table_size	
	....	....
3		**atomic**{
4		main_table_size=10
5		main_table=new_table(10)
6		}
	....	....
7	_table=main_table	
8	_offset=_table+(_size−1)	
9	∗_offset=1	
10	}	

Figure 2.13: Memory Corruption when accessing indexed data.

The problem occurs when T1 reads the size of the old table and the pointer to the new table (with a smaller size). When the write occurs on line 9, the table offset (*_offset*) is larger than the array dimension. Therefore, if the write is not prevented by the STM engine, it will change memory outside the array. It may corrupt memory allocated for other transactions, or even for non-transactional use. Even tough the old values will be restored when the transaction aborts, the value has already flickered, which may cause problems if some other thread reading the changed value. One way to avoid the problem is to revalidate the read set before every write, but the performance cost may be too high.

The problem described on Figure 2.13 does not happen with the redo log strategy because the shared variables are only changed after the read set is validated. Also, with programming languages like Java, this error would throw an exception that could be caught by the STM engine.

The transactional engine may detect all the mentioned hazards and prevent the user code to run in inconsistent state by re-validating the whole read set at every transactional load. However the size of the read set increases with the number of loaded variables and re-validating the read set at every transactional load becomes expensive for longer transactions. The cost is $O(n^2)$ where n is the number of transactional reads. This is why several systems opt to revalidate the read-set periodically or only at commit time.

TL2 [DON06] uses a different algorithm to avoid this problem. Instead of using a version counter per variable, it uses a global version clock algorithm with the clock being incremented for every commit. When a transaction starts, it reads the global version clock into a thread local storage, which is the *transaction timestamp*. Transactional loads check if the lock version of the variable is smaller or equal to the *transaction timestamp*. If it is the case the transaction proceeds, otherwise it immediately aborts.

The global version clock algorithm detects when the read set is not consistent and prevents user code to run in an inconsistent state. This approach is more efficient than revalidating the read set at every transactional load, as it requires only the lock of the loaded variable to be checked. The cost is $O(n)$ where n is the number of transactional reads.

The validation made by the global version clock algorithm is weaker than the entire read set validation. Validating the entire read set detects transactions running in inconsistent and obsolete states, while the global version clock algorithm only detects inconsistent states (which is not a problem since obsolete states do not cause the transaction to have erratic behaviors). Obsolete states are detected at commit time. This algorithm also has more false positive aborts: situations were the transaction shouldn't abort but it does. An example is show on Figure 2.14 where, on step 5, transaction A detects a conflict reading variable x and aborts.

	T1	**T2**
1	**atomic** {	
	....	....
2		**atomic** {
3		x ++;
4		}
	....	....
5	_a=x;	
6	}	

Figure 2.14: False positive with the global version clock algorithm.

2.3.6 Lock Placement

There are two main strategies for lock placement on lock based STMs. The first is to place the lock next to the data. The second is to place the lock on a separate table.

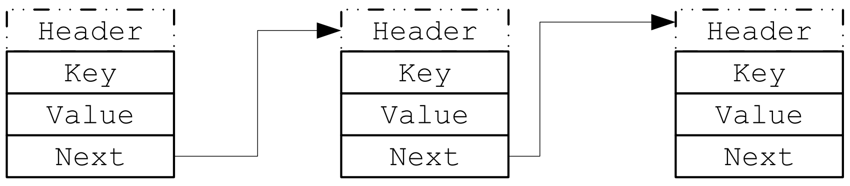


Figure 2.15: Sample list implementation using adjacent locks

Lock adjacent to the data

Placing the lock next to the data has the advantage of having a higher chance that both, the lock and data, will stay on the same cache line. This may yield a better performance, by reducing the number of cache misses.

However, placing the lock and data next to each other, requires changes to the structure of the objects on the heap (example in Figure 2.15). Also, the object handling is different. Without compiler support, the developer holds a pointer to the header and, to access the data itself, the pointer is incremented by the header size. These changes limit the possibility of reusing of existing libraries, as they change both the structure of the objects and the way they are handled. Some STM engines [HPST06] have used compiler support to automatically add an extra header.

Placing the lock next to the data also brings a difficulty when using non garbage collected programming languages. If the object is deleted and freed, the header is also deleted. If other threads are still referencing the freed object, they may end up reading, or even worse, writing to freed memory. Even tough the transaction will abort and the old value are restored, writes to freed memory must be prevented by the STM engine.

On non garbage collected languages, STMs must use either a specialized malloc and/or free implementation [Enn06, SATH$^+$06] or a memory system that does not allow memory still referenced by any transaction to be reused for non transactional use [DS06, DON06]. On garbage collected languages this is not a problem, since the runtime system only releases the object's memory when there are no more references to it.

Lock in a separate table

Placing the locks on a separate table does not require changes to the heap objects, nor to the way they are handled. With this technique it is easier to re-use existing libraries, as it only requires the memory accesses (reads/writes) to be instrumented.

To map a variable to the lock in the lock table, it is necessary to have a hash function that maps the variable address to a position in the table.

One practical problem with the lock table approach is the fact that several memory addresses may be mapped to the same table position (lock collision). While not being a correctness problem (provided that the STM engine is prepared for this matter), it may result in a performance degradation. If there are collisions on the lock table, some variables will be seen as locked, but in fact the lock was made for another variable whose address maps to the same position in the lock table. As a result, there may be a performance impact due to an increased number of unnecessary aborts.

2.3.7 Contention Management

Obstruction free and lock based STMs have the need for an out-of-band mechanism to solve transaction collisions and avoid livelocks and deadlocks (for lock based STMs). In some systems, the responsibility of guaranteeing progress is left to a contention manager [HLMWNS03], which decides which transaction should abort, in case a conflict occurs. When a transaction tries to update a transactional variable and it detects conflict with another transaction, it informs the contention manager about the pending conflicts. The contention manager uses heuristics to implement conflict resolution policies. Simplest heuristics may be: to always abort the conflicting transaction; to always abort the requesting transaction; or to back-off and retry a few times before aborting the conflicting transaction. More complex contention management heuristics are described in [WNSS05].

2.3.8 Support for Long Running Transactions

Some types of applications require the use of large transactions—for instance a transaction that transverses a large transactional list to count the number of elements. These large transactions suffer from having a higher probability of colliding with other transactions. In the list example, if a node is added or deleted while the list transversal is occurring, a collision may occur and one of the colliding transactions must either i) wait or ii) abort. Both strategies (waiting or aborting) have severe drawbacks when dealing with large transactions: waiting for a large transaction to finish may slow down the system or even halt it because every other transaction is waiting for the large transaction to finish; and aborting may lead to a livelock on the large transaction as it is never able to finish successfully.

The STM engine [CRS06] uses a technique to prevent the problem of long running read only transactions. This technique, which is common on database systems, registers the history of changes of the transactional values and it allows read-only transactions to succeed, even if there are writing transactions running in parallel. With the history of changes, transactions can look at the old values and therefore they can consult the memory snapshot as it was when the transaction started.

2.4 A Bit of History

Software transactional memory has come a long way since its introduction by Shavit and Touitou in 1995 [ST95]. Many new ideas have been proposed and many have been abandoned. We describe now some of the most significant events in Software Transactional Memory history.

The initial STM implementation of Shavit and Touitou used the lock-free property, which provided resilience to thread delays, failures and avoidance of deadlocks and livelocks. Later, in 2003, the obstruction freedom property was introduced on transactional memory by Herlihy et al., [HLMWNS03], which proved to make the STM implementations simpler and more efficient. In their paper, they also created the first STM engine that allowed the dynamic creation and deletion of objects within the transaction. Whereas the prior implementations required the variables and transactions to be statically defined in advance.

Another interesting advance was made in 2003 by Harris and Fraser [HF03], where they proposed, among other things, the first word based STM engine that did not require additional space allocation per each variable and it was therefore more convenient to use.

Nowadays most STM implementations use lock based algorithms, which are not resilient to thread failures nor delays, however they have a significantly better performance [Enn06,DS06,DON06]. Figure 2.16 taken from [NSS] illustrates this trend in the design of the STM engines.

Since the appearance of lock based STM engines, there has been a broad range of ideas regarding design options. An interesting study made by Saha et al., [SATH$^+$06] in 2006 shows that versioned write locks have much better performance than read write locks. They also pointed out that the undo log had lower overhead than the redo log strategy due to the avoidance of look asides.

Later, Dice et al, introduced a STM engine in 2006 which used a redo log strategy with the difference that locks were acquired only at commit time. This solution had better performance than the undo log strategy when contention is higher. There is no consensus on which strategy is better, each one has better performance on specific scenarios.

In 2005 Harris et al. [HMPJH05] proposed a new set of features (*conditional waiting* and *composing alternatives*) for transactional memory. These new features have proved to extend the range of problems that transactional memory can address.

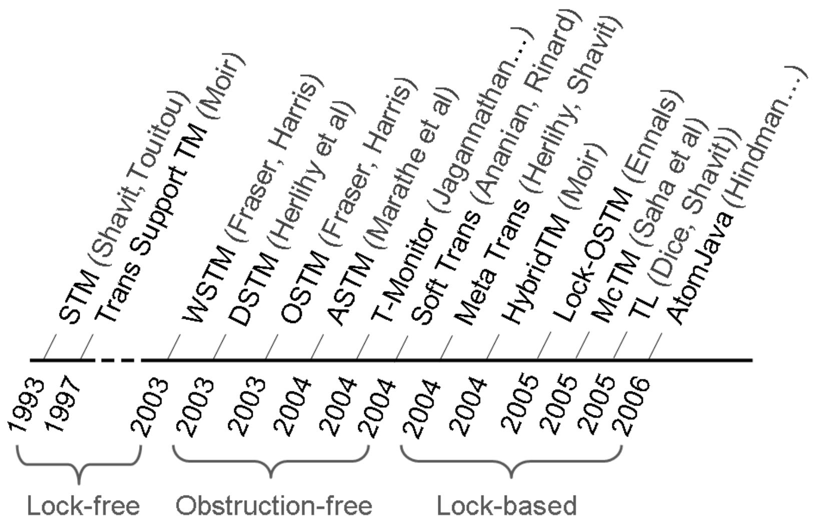


Figure 2.16: STM History

Nowadays, much of the work is still towards optimizing the performance of the STM engines. Harris et al., in 2006, proposed several new optimization strategies in [HPST06]. The integration of the STM engines into the compiler has also been suggested as a way to increase its performance. Both Harris et al and Adl-Tabatabai describe it on [HPST06, ATLM+06].

Chapter 3

Prototype Development

This Chapter describes the TL2 STM engine, in which this work is based. It also describes in detail the changes made to TL2 which originate this thesis.

3.1 Introduction

Our prototype began with a port of TL2 to Linux/X86/GCC (TL2 was originally developed for Solaris/SPARC/SUNPRO C compiler). Afterwards, several modifications were made to implement new features, aiming at achieving better performance than the original TL2, increasing its safety and facilitating the debugging of the engine. Some of the added features were: user called aborts, automatic transaction retry and transaction nesting. Performance modifications were related to log management strategies, transactional granularity, partial state validation and algorithmic changes to optimize the load operation. Test related modifications were the creation of a new event tracing engine.

In the next section we describe the features and implementation options of the original version of TL2. The followings sections describe the modifications made to TL2.

3.2 Original TL2

TL2 was chosen among other STM implementations because it has two interesting features: i) it doesn't require the use of specialized malloc/free implementations (even for non garbage collected languages), allowing memory to be recyclable between transactional and non-transactional space; ii) by using the global version clock algorithm it prevents transactions from running in inconsistent states.

The original version of TL2 was word based; used a redo log logging strategy; and, depending on compile time options, it allowed locks to be placed on a separate table or adjacent to the data.

3.2.1 Lock Structure and Placement

The default working mode of TL2 places locks on a separate table. The structure of the lock table in TL2 is shown in Figure 3.1. The addresses of the transactional variables are hashed to get the lock position in the lock table. TL2 also allows the locks to be placed adjacently to the object.

The lock itself has two fields, one containing the status (acquired or not) which is located on the least significant bit and another containing the remaining bits. If the lock is 0, then the lock is released and the remaining bits hold the version number of the variable. If the bit is 1, then the lock is acquired and the remaining bits hold the address of the write set entry of the transaction holding the lock. Having one less bit to hold the pointer is not a problem since the addresses of the write set entries are, at least, four bytes aligned, therefore the last two bits are unused.

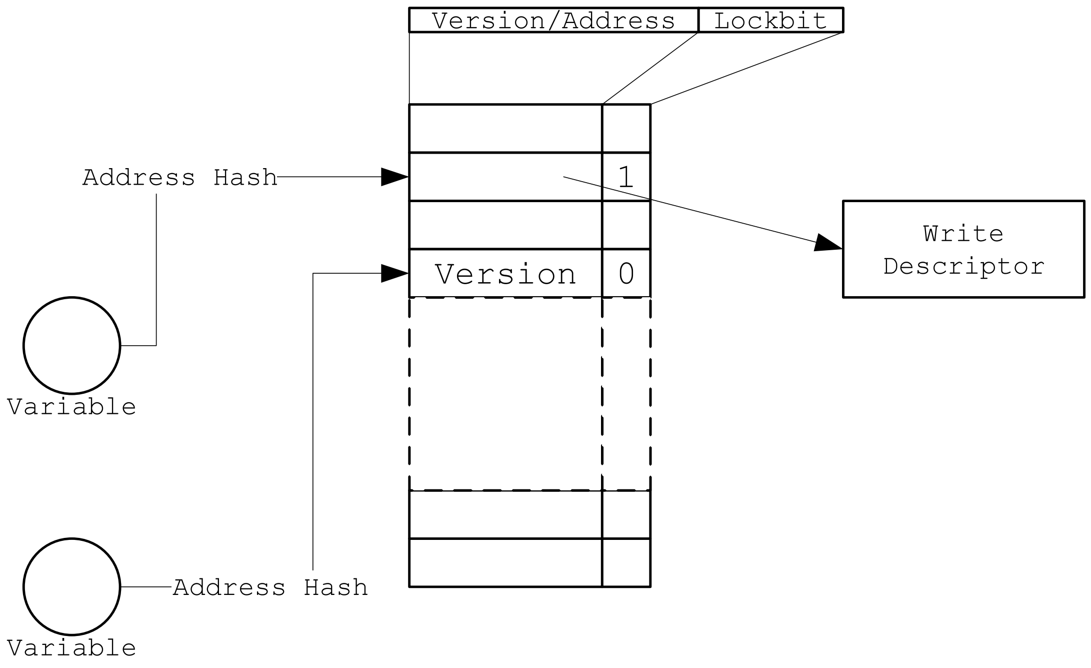


Figure 3.1: Lock Table in TL2

3.2.2 Word Mode

This is the mode that was originally implemented in TL2 and the API is as show on Figure 3.2. The accesses are either reads or writes made to single words and they are all intermediated by the STM engine. Word based mode can also be used to access objects, treating object fields as words.

```
TxStart(Thread *t);
TxCommit(Thread *t);
intptr_t TxLoad(Thread *t, intptr_t *addr);
void TxStore(Thread *t,
             intptr_t *addr,
             intptr_t value);
```

Figure 3.2: Simplified API of TL2

3.2.3 Redo Log Strategy with Consistent State Validation

The original TL2 works in redo log strategy with consistent state validation and it follows the algorithm bellow:

When a transaction starts, it reads the global version clock into the *transaction timestamp*.

On a transactional load, the transaction first checks if the variable is already in the write set, if it is, then the variable has already been written in this transaction and it returns the value on the write set; otherwise it logs the read on the read set and returns the value. The consistent state validation is performed by checking the lock version before and after reading the variable.

For the read to be valid, three constraints must apply (Figure 3.3):

- Lock isn't held—to avoid collisions with other writing transactions.
- Lock version is the same in both checks (before and after the read).
- Lock version is lower or equal to the *transaction timestamp* (t→rv)— to avoid seeing inconsistent states (otherwise the variable may have changed since the beginning of the transaction).

If constraints apply, the read is successful and the read set of the transaction continues to be consistent. Otherwise it aborts.

```
TxLoad(Thread *t, intptr_t *addr){
  lock_version1=GetLock(addr);
  value=*addr;
  lock_version2=GetLock(addr);
  if(is_version(lock_version1) &&
       lock_version1==lock_version2 &&
       lock_version1<=t->rv){
       ...
    return Value;
  } else {
    abort();
  }
}
```

Figure 3.3: Consistent state validation in Redo Log/Word based mode

On a transactional store, the transaction simply logs the write on the write set. When a transaction commits, the variables in the write set are locked, the read set is validated, the global version clock is incremented, the new variable values are copied from the redo log to their positions and finally the locks of the written variables are released with an updated version corresponding to the new global version clock number.

Aborts simply discards the redo log.

3.2.4 Contention Management

Since TL2 is a blocking/lock based STM, it has to deal with deadlock/livelock problems. On TL2, when a transaction finds a locked variable, it aborts and retries the transaction. The retry is delayed for a random amount of time which is exponential with the number of retries.

3.3 TL2 Port to X86

Original TL2 was developed for Solaris on SPARC/64bit with SUNPRO C compiler. We had to port it, since the hardware and software we had available was Linux on X86/32bit with GCC compiler. On the port to this architecture several modifications were done as described ahead.

3.3.1 Solaris to Linux

TL2 uses the Solaris scheduler control mechanism *schedctl* [sch, DS07], which allows threads to request the kernel to avoid being preempted for a brief period. TL2 uses this mechanism while the locks were being held, to try to avoid the situation where a thread looses the CPU while holding locks, forcing other threads to abort or spin for the lock.

On the port to Linux, the TL2 scheduler control was removed, since this is not available on Linux.

3.3.2 SPARC to X86

TL2, as well as several other STM implementations, use native *Compare And Swap* (CAS) instructions to implement the locking mechanism. Since SPARC has a different instruction set than X86 and since these instructions aren't available as portable libraries in C, there was the need to port them using GCC inline assembly for X86 [int]. The implementation was inspired on the Ennals STM engine [Enn06].

TL2 uses a SPARC non-faulting load instruction [WG], which allows a thread to read a memory location like a regular load, except for the fact that it does not throw an exception (e.g., segmentation fault) when the memory location is not in the process address space, instead the instruction returns zero. Original TL2 used this instruction to read the lock values and to load transactional variables due to the fact that transactions may try to read memory locations already released by other transactions. An example is a thread trying to read a list node while another is trying to release the very same node. It may happen that the first transaction gets a pointer to the list node and, before reading it, the second transaction removes the node from the list, commits and free's the node. When the first transaction tries to read the node, it has already been released and it could result in an exception.

Since the non-faulting load instruction is not available on X86, the solution was to implement a fault handler with *setjmp/longjmp* instruction pair. On the beginning of the transaction, *setjmp* saves the processor context into a memory buffer; if an exception (e.g., segmentation fault) is caught, *longjmp* restores the processor context and the execution continues as if *setjmp* had just been called, in other words the transaction is restarted.

The use of *setjmp/longjmp* adds a limitation to the prototype. The instruction *setjmp* saves the caller context in a buffer, which includes the stack pointer but not the stack contents. Therefore *longjmp* must be called inside the same or on a deeper stack frame than *setjmp*, otherwise *longjmp* may find an invalid stack frame. The consequence is that the function starting the transaction can only return after it commits or aborts.

3.3.3 64 bit to 32 bit architecture

TL2 was implemented for a 64 bit architecture. Porting it to a 32 bit architecture was not trivial because of the use of versioned write locks. Since, on a 32 bit architecture, the version numbers are only 31 bits long (1 bit is reserved to indicate whether the lock is held or not) and they are incremented at every commit, the roll-over time for the version counter is drastically reduced. On a 1CPU/1Ghz machine, assuming each transaction takes 10.000 cycles to complete, the roll-over time is $\approx$ 6 hours.

One option to solve this problem would be to handle the overflow, by reseting the counter when there are no active transactions (eventually forcing every transaction to abort); another is to use a wider version lock to avoid the problem, at least as a practical concern. The option chosen was to use 64bit version locks, by using the X86 SSE instruction set. SSE instructions can atomically load, store, and CAS 64bit words and the roll-over time is increased from $\approx$ 6 hours to $\approx$ 3 million years, using the same 1CPU/1Ghz machine.

3.4 New Features

We now describe the features we introduced into the prototype.

3.4.1 User Aborts

Sometimes it is convenient for the programmer to abort explicitly a transaction and undo its effects. An example (Figure 3.4) is a transaction wishing to receive a message from a queue and after receiving the message discovers that the message is not "interesting", so the thread needs to undo every change to the queue. This functionality was added to the prototype. The algorithm is described in Section 2.3.4.

```
atomic{
    msg = Receive();
    if(!is_interesting(msg)){
        abort();
    }
}
```

Figure 3.4: User specified abort

3.4.2 Automatic Transaction Retry

This extension allows an aborting transaction to be retried automatically. If a transaction detects a collision with another transaction, one of them must abort. The example on Figure 3.5, shows a transaction that found a collision on step 3 and is going to abort. The aborting transaction has several options regarding the control flow: i) it may continue its control flow inside the transaction, ignoring any further transactional writes (continues to step 4); ii) it may abort, with the control leaving the transactional block (goes to step 6); or iii) it may automatically retry, placing the control flow on the start of the transaction (restarts on step 2).

```
1 // ...
2 TxStart();
3   a=TxLoad(x); // collision is detected
4   b=TxLoad(y);
5 TxCommit();
6 //...
```

Figure 3.5: Control flow after abort

Options 1 and 2 have the semantic *at most one successful commit* and the problem is that programmers must be aware that transactions may not finish successfully, so the commit return status must always be tested. Developers must place the transaction inside a while loop to guarantee a successful commit (Figure 3.6).

```
// ...
do{
  TxStart();
    a=TxLoad(x);
    b=TxLoad(y);
  status = TxCommit();
}while(!status);
//...
```

Figure 3.6: Non retrying transaction placed inside a while loop

The option 3 (automatic retry) has a semantic *exactly one successful commit* and is a safe option to be used without placing the transaction code inside a loop. There has been two options for implementing the automatic retry mechanism: using *setjmp/-longjmp* instructions if they are available, calling *setjmp* on the beginning of the transaction and calling *longjmp* if the transaction aborts; or placing the transaction code within a function/method [HLM06], this way the transaction is started via a call to the transaction manager, which in turn automatically calls the transactional code inside a loop. An example is show in Figure 3.7. Original TL2 had the option 1 implemented (control flow continues inside the transaction). Our prototype implemented the option 3 (automatic retry) using the *setjmp/longjmp* instructions.

```
int TransactionXPTO(){
    a=TxLoad(x);
    b=TxLoad(y);
    return 1;
}

//...
CallTransaction(TransactionXPTO);
```

Figure 3.7: Non retrying transaction placed inside a while loop

3.4.3 Transaction Nesting

We have added to our prototype support for transaction nesting. Our implementation is partially based on *closed nesting*.

When a transaction starts, it creates a new transaction descriptor and all load and store operations are recorded in the transaction log of that descriptor. When a transaction commits, the transaction log is concatenated with the parent transaction.

When a user explicitly aborts a sub-transaction, the changes made by the sub-transaction are undone and the sub-transaction log is discarded. In other words, only the effects of the sub-transaction are undone. However, when a sub-transaction finds a collision with other transactions the whole transaction is rolled back (including the main transaction) and retried because the variable that caused the collision may have also been read by the main transaction. One way to avoid this, would have been to validate the entire read-set of all transactions when a sub-transaction aborts.

The use of large nested transactions along with high collision rates, suffers the same problem as non nested transactions—high abort rates.

Nesting was implemented for all combinations of the prototype—undo log and redo log.

3.5 Performance Related Changes

The use of concurrent programming aims at taking advantage of the multiple cores that computers have to improve the application throughput or decrease its processing time. However, programming concurrent applications introduces overheads related to the synchronization of tasks and adds contention to shared structures.

The overhead is related to the additional instructions that have to be executed to perform the synchronization and it increases with the number of synchronization calls. The contention is related to the simultaneous accesses to the shared structures, (either application data structures or synchronization structures, like locks) which leads to some of the tasks having to wait for a lock to be released or may cause additional cache coherency traffic. Contention increases with the number of threads accessing simultaneously the same shared structures.

The effects of the overhead of the synchronization mechanisms are mostly visible when using a low number of CPU cores because the overhead may be higher than the gain of using more than one CPU core. The effects of contention of synchronization mechanisms are mostly visible when using a high number of cores on applications concurring for the same shared structures. The probability of having simultaneous accesses to the shared structures increases, leading to higher waiting times for the locks, more cache coherency traffic and more cache misses.

Next we describe the changes we made to the prototype focused on reducing the overhead and minimizing the contention.

3.5.1 Undo Log Strategy with Consistent State Validation

The undo log strategy is a modification of TL2. In this mode the updates are made in place and the locks are acquired as soon as the variable is written.

When a transaction starts, it reads the global version clock into the *transaction timestamp*.

On a transactional load, the transaction logs the read operation on the read set and returns the value. The consistent state validation is performed by checking the lock version before and after reading the value and doing the same constraint verifications as in redo log/word based mode (Section 3.2.3). These checks are bypassed if the owner of the lock is the current transaction.

On a transactional store, the transaction verifies if the lock version is lower or equal to the *transaction timestamp*[1], acquires the lock, records the value in the undo log and stores the new value in place. If the variable is already locked by this transaction, the new value is simply stored in place, otherwise the transaction immediately aborts.

When a transaction commits, the read set is validated, the global version clock is incremented and finally the locks of the written variables are released with the lock version set to the updated global clock version. If the transaction is read only, the commit always returns successfully because the read set is guaranteed to be consistent (although it may be obsolete). Committing an obsolete but consistent read only transactions is not a problem.

Aborts increment the global version clock, copy the variable values from the undo log back to their original position and release the locks with the updated version of the global version clock number. Aborts do have to increment the value of the lock at least on read-write transactions, because the written (dirty) values may have been read by another transaction and the way to abort the other transaction is to update the clock version—even tough the value is restored same when the transaction started.

3.5.2 Object Mode

Object mode is another modification to the original TL2 and it is only implemented for the undo log strategy (described in Section 3.5.1). The API used to handle the objects is different from the word based mode. Like the STM engine of [HPST06], reads and writes are made directly to the objects after the corresponding open calls. In other words, after the object is opened, it may be accessed directly from the application without the intermediation of the transactional engine. The side effect of using this mode

[1]This is necessary to make sure that read-write variables haven't changed between a prior read and the write and to ensure that this transaction is not writing to a freed variable.

(and the usage of optimistic read locks) is that the transaction code may run inconsistent. A transaction may read a value while it is being written by another transaction. The way to prevent this side effect is to explicitly validate the object's version after accessing it.

The API of object mode is shown on Figure 3.8 and it works in the following way. Before an object is read, *TxOpenRead* is called and it records the read operation on the read set. After *TxOpenRead* is called, the object can be accessed for read directly without intermediation of the engine. However each read may render the transaction to an inconsistent state. To return to a consistent state the transaction must call *TxVerifyAddr* on the object read. *TxVerifyAddr* checks if the lock version of the object is lower or equal than the transaction timestamp. If it is, the object hasn't been changed since the beginning of the transaction and the transaction continues to be consistent, otherwise a collision has occurred and the transaction immediately aborts. The macro *TxReadField* facilitates the usage by doing everything in one step—reads the value, calls *TxVerifyAddr* and returns the value.

Before an object is written, *TxOpenWrite* is called. It checks if the lock version is smaller or equal to the *transaction timestamp*, acquires the lock and records the current object data in the undo log. If the object is already locked by this transaction, *TxOpenWrite* simply returns, otherwise the transaction immediately aborts.

```
TxOpenRead(Thread *t, void *addr);
TxOpenWrite(Thread *t,
            void *addr,
            int size);
TxVerifyAddr(Thread *t,
             void *addr);
#define TxReadField(t,addr,field)
```

Figure 3.8: Simplified API for handling objects in object based mode

When using object mode, the object fields are accessed directly without intermediation of the transactional engine. This allows the object accesses to be optimized by the compilers. In word based mode, the read accesses are behind a function call, which results in a performance loss due to the function call overhead and to the fact that the memory accesses can't be reordered by the optimizer.

In summary, the advantages over the word based modes are: only one entry in the read set and one lock verification per object; object fields can be accessed directly (without transactional API); and when accessing multiple fields of an object sequentially, only one validation is enough to verify the state consistency. The disadvantages are the coarser lock granularity, resulting in a lower potential for concurrency.

The way we envision the future implementation of object based mode with redo log strategy on this prototype is for the application to hold two pointers to the object when writing into it. One pointer for the real address of the shared object and another for the private write buffer. The first would be the shared variable itself and used as the argument to the *TxOpenRead* and *TxOpenWrite* calls. The second would be the transaction private write buffer and be used to write to the object. This way we could access the object fields directly by using the private buffer, however at a cost of having to hold to pointers.

3.5.3 Full Validation, Partial Validation and No Validation

With object based mode the use of *TxVerifyAddr* is not required for the transaction to finish correctly; it is only required to guarantee that the transaction runs in consistent memory states e.g. to prevent infinite loops. Other STMs that do not use the global version clock algorithm may also achieve this by revalidating the read set after every read, but it would have a cost of $O(n^2)$—where n is the number of transactional reads. This prototype does this validation with a cost of $O(n)$.

However, even with the global version clock algorithm, validation has a non negligible cost (discussed in Chapter 5) and therefore state validation should be minimized as much as possible. One possibility to reduce this overhead is for state validation to be made only on the places necessary to avoid unacceptable transaction behaviors like infinite loops, failure of valid assertions, etc. For instance, when accessing multiple fields of an object sequentially, only one validation is usually enough to verify the state consistency.

We divide consistent state validation in three modes: *full state validation*, *partial state validation* and *no state validation*. *Full state validation* is achieved by verifying the lock version every time a read to a transactional variable is made and therefore the transaction is always consistent. *No state validation* is achieved by never verifying the transaction state while the transaction is running. In other words, the object is opened before any of the fields are read but the engine never verifies if the state continues to be consistent. *Partial state validation* is achieved by selectively choosing the places to validate the state. This may be a feature of a compiler or a transactional access optimizer, which identifies sequential accesses to the same object and performs validation only after all sequential accesses are finished.

Next, we show three examples, one with *full state validation*, another with *partial state validation* and another with *no state validation*. They show a part of a transaction reading the key and value of a list node. After the read, an assertion is made to verify the invariant that the keys must be greater than zero. The assertion must be checked within a consistent state; otherwise, interleavings with other transactions may fail the assertion even if it is valid according to the algorithm.

The first example (Figure 3.9) shows a transaction running with *full state validation*. This approach makes the transaction always run in a consistent state—at the cost of additional checks. The assertion can be made immediately after reading the key, because the state is guaranteed to be consistent.

```
1 TxOpenRead(t, node);
2 key = TxReadField(t,node,key);
3 assert(key>0);
4 value = TxReadField(t,node,value);
```

Figure 3.9: Full state validation

The second example (Figure 3.10) shows a transaction running with *partial state validation*. The validation is performed after reading the key and value. This approach however allows the transaction to run inconsistent in steps 2 and 3. After the revalidation on step 4, the transaction is again consistent until the next transactional load. The assertion can only be made after the verification finishes to guarantee that it is issued in a consistent state.

```
1 TxOpenRead(t, node);
2 key = node->key;
3 value = node->value;
4 TxVerifyAddr(t, node);
5 assert(key>0);
```

Figure 3.10: Partial state validation

The third example (Figure 3.11) shows a transaction running with *no state validation*. The transaction code may run in inconsistent state and therefore no assertions can be issued (unless the entire read set is validated).

```
1 TxOpenRead(t, node);
2 key = node->key;
3 value = node->value;
```

Figure 3.11: No state validation

The first approach (*full state validation*) is simpler to be used by the programmer, because there is no need to keep track of which objects have to be verified for consistency.

The second approach (*partial state validation*) has less overhead due to the fewer number of validations made. Yet, it is more complex for a programmer to use it directly. It is better suited for compilers that generate the transactional code from a higher level language. The compiler may validate the transaction state after a sequence of transactional reads, or before some event that needs a consistent state — e.g., before issuing an assertion, before testing a loop condition, etc.

The third approach (*no state validation*) has even less overhead. However transactions run totally inconsistent and nothing can be ensured while the transaction is running. Only when a commit returns successfully, it is guaranteed that the transaction was successful and no harmful interference occurred.

One simplification may be done when a write lock is acquired. Since transactional writes acquire object locks, the reads don't need to be validated on objects opened for write (Figure 3.12).

```
TxOpenWrite(t, node, sizeof(node_t));
key = node->key;
assert(key>0);
value = node->value;
```

Figure 3.12: Consistent state validation for locked variables

3.5.4 Reducing TxLoad Overhead

The performance of the STM engine is mostly dominated by the efficiency of the transactional load operation. Usual applications have a number of reads which is far superior to the number of writes [HPST06] and therefore to the remaining transactional calls—stores, starts, commits, aborts, etc. Therefore, any improvement on the load operation has much more significant impact than any improvement on other operations (Amdahl's Law).

On the other hand, as the number of threads increases, the likelihood of threads colliding in the access to shared data also increases. The shared data may be the data variables themselves, locks, or any other shared structures. Reducing contention should, therefore, be focused on reducing the number of accesses to shared data or restructuring the access pattern to have less collisions.

We have designed an improvement to the *TxLoad* algorithm that reduces the number of steps on the transactional load and simultaneously reduces the accesses to the shared variables.

The load operation of the original TL2 checked the lock version before and after reading the variable (Figure 3.3). Afterwards it verified three constraints:

i) Lock isn't held.

ii) Lock versions are the same between both checks.

iii) Lock version is lower than the *transaction timestamp* ($t \rightarrow rv$).

The first two checks verify if the variable isn't dirty by ensuring that it hasn't changed between the two lock validations and that it is not being modified by another transaction. The third check verifies if the variable has changed since the beginning of the transaction.

TxLoad operation was improved by changing the commit algorithm in a way that allows the load operation to verify the lock version only once, while keeping all algorithmic properties. The idea is that, if one transaction changes the variable when another is reading it, then the lock, on the posterior validation, will be either acquired or have a version which is greater than the transaction timestamp. Therefore, the lock version validation prior to the read of the value (as per TL2 algorithm) can be avoided (Figure 3.13). The implementation of this optimization required a modification in the commit algorithm.

```
...
TxLoad(Thread *t, intptr_t *addr){
  value=*addr;
  lock_version=GetLock(addr);
  if(is_version(lock_version) &&
       lock_version<=t->rv){
       ...
    return Value;
  } else {
    abort();
  }
}
```

Figure 3.13: Optimized *TxLoad* algorithm.

The commit algorithm had to be changed to have the global version clock only being incremented after the values are safely written in their final locations. This is to avoid the situation shown in Figure 3.14 where a dirty read is not detected when *TxCommit* is interleaved with another transaction.

Note that the clock is always incremented by 2, odd numbers mean the lock is held, even numbers mean the lock is released.

The solution to the problem is to increment the global version clock after the values have been written and flushed to memory 3.15. This new setup requires redo log to be

	T1	T2	Description
1	$TxCommit$		Commit starts
2	Acquire Locks		Locks are acuired
3	Increment Global Clock		Clock is incremented to 12
	....	....	....
4		$TxStart$	TX starts and timestamp is sampled on value 12
5		$TxLoad$	Load starts
	....	....	....
6	Validate ReadSet		Read set is validated—transaction will commit
7	Apply redo log	Read value	T1 applies the redo log while T2 reads the variable value—value read is dirty
8	Release locks		Locks are released with the new version set to 12
	....	....	....
9		Read lock	T2 checks the lock version and since it is equal the transaction time stamp (12), it considers the read to be valid, although it was dirty.

Figure 3.14: Redo log mode: Reference to Non-Transactional Memory.

flushed to memory (via a CPU memory barrier [McK05]) to prevent the compiler and the CPU from re-ordering the instructions.

3.5.5 Block Size

Another experiment with the STM engine was the change of the block size. The change is only implemented in the undo log word based mode and the goal is for the STM engine to have less locks and lower overhead on acquisition and release of the locks at the cost of a coarser lock granularity. In the regular undo log mode, when a transaction writes to a variable, it locks the variable and stores its previous value in the undo log. When the block size is changed to the cache line size and a transactional store is issued, the STM engine locks the entire block (one lock is enough for the full block) and records the value of the full block in the undo log. The advantage is the fewer number of locks/unlocks and copies to the undo log but it has a coarser lock granularity with higher probability of false contention. This technique is also used by [SATH+06].

The block size must be coordinated with the application's memory allocation routine. The application must allocate a size which is multiple of STM block size and the memory must be aligned to the block size. Otherwise the STM engine, when copying

	T1	**T2**	**Description**
1	$TxCommit$		Commit starts
2	Acquire Locks		Locks are acuired
	. . . .	. . . .	. . . .
3		$TxStart$	TX starts and timestamp is sampled on value 12
4		$TxLoad$	Load starts
	. . . .	. . . .	. . . .
5	Validate ReadSet		Read set is validated—transaction will commit
6	Apply redo log	Read value	T1 applies a the redo log while T2 read the variable value—value read is dirty
7	Increment Global Clock		Clock is incremented to 14 (clock is always incremented by 2)
8	Release locks		Locks are released with the new version set to 14
	. . . .	. . . .	. . . .
9		Read lock	T2 checks the lock version (14), since it is greater than the transaction time stamp (12), the read is not valid and the transaction is aborted.

Figure 3.15: Redo log mode: Reference to Non-Transactional Memory—Corrected version

to the undo log, may read unallocated memory; and when aborting a transaction it may write to memory allocated for other uses. Therefore the use of this functionality is not safe unless the application handles this by, for example, allocating an aligned array of objects.

3.5.6 Lock Adjacent to Data

With object based mode (detailed in Section 3.5.2) we extended the prototype to support lock placement adjacent to the object, as well as, on a separate table.

To place the lock next to the data it is necessary to add to the data structure a field that will hold the lock information. Since our prototype was implemented in C programming language and C does not have any automatic way to add fields to structures/objects, it was decided to manually add a lock field to every transactional object on the application code. Although this is not suitable for a production use it is good enough to realize some experiments and evaluate the performance.

3.6 Safety Improvements

Next we describe our new quiescing algorithm. Unlike the original algorithm from TL2, this one guarantees that transactions run in consistent states even when memory is entering and leaving the transactional space.

3.6.1 New Quiesce Algorithm

Transactional variables must only be accessed by transactions, however it is desirable that variables leaving (being freed) the transactional space can be re-used by non transactional operations. To handle this transition the variables must be quiesced before they are freed (example in Figure 3.16). In TL2 the quiesce function is named *TxSterilize*.

Quiescing on the original TL2 consisted of waiting for the writes to be drained and the lock to be released. This, however, fails under some circumstances. Consider the situation where we have a list with two nodes (A and B) and three threads (TX1, TX2 and NT3) are using it. TX1 is looking for node B, TX2 is deleting node B, NT3 is not running any transaction. The following sequence of events takes place:

1. TX1: starts (with *transaction timestamp* 10) and looks up node A, which is prior to node B, reads the pointer to node B.

```
// allocate memory
new_memory=malloc(sizeof(int));
TxStart(t);
  TxStore(t, &a, new_memory);
TxCommit(t);

// release memory
TxStart(t);
  memory_to_free=a;
  TxStore(t, &a, NULL);
TxCommit(t);
TxSterilize(t, memory_to_free);
```

Figure 3.16: Creating and releasing a transactional variable (simplified)

2. TX2: starts, (with *transaction timestamp* 10) looks up node B and removes all references to node B.

3. TX2: commits (increments the global version clock to 12).

4. TX2: quiesces node B (no thread is currently locking B).

5. TX2: frees node B.

6. NT3: starts, calls malloc and receives a pointer to the *same memory where node B was previously referenced*.

7. NT3: writes to that memory.

8. TX1: follows the pointer to the late node B and reads its contents. At this point TX1 is reading memory already recycled for usage of another thread. Hence TX1 is running on an inconsistent state, which violates the idea of never seeing inconsistent memory states.

The newly designed algorithm allows memory to be recycled and to always run in consistent states. The solution found was for the quiesce operation to treat the delete as a regular write. Quiesce updates the lock version to the current value global version clock and since it is always called after the commit and before the free, updating the node version number will invalidate any future reads to it by other transactions. Quiesce does not need to increment the value of the global version clock because it was already incremented when the commit was done (quiesce can't be called inside an active transaction). With the new quiesce, the previous operations are transformed into the following, where the step 8 no longer sees the inconsistent state:

1. TX1: starts (with *transaction timestamp* 10) and looks up node A, which is prior to node B, reads a pointer to node B.
2. TX2: starts (with *transaction timestamp* 10), looks up node B and removes all references to node B.
3. TX2: commits (increments the global version clock to 12).
4. TX2 quiesces node B (node B's lock is set to 12).
5. TX2: frees node B.
6. NT3: starts, calls malloc and receives a pointer to the *same memory where node B was previously referenced*.
7. NT3: writes to that memory.
8. TX1: follows the pointer to the late node B and verifies that B's lock is greater (12) that its own transaction timestamp (10). TX2 aborts and the user code never sees an inconsistent state.

Even if some other transaction incremented the clock between step 3 and 4, the result would be the same.

Apart from solving the above problem, the new quiesce function also allows recycling of memory when using the undo log mode (described in Section 3.5.1), which could not be done on TL1 [DS06].

3.7 Debugging enhancements

This section describes the new tracing engine implemented to facilitate the debugging of the STM engine.

3.7.1 Event Tracing

The debugging of the STM engine is a complex task. The amount of concurrency of transactional applications is at the level of very fine grained locking (one lock per variable), therefore a huge number of interleavings is allowed. With this scenario, traditional debuggers are of little use, not only they interfere with the run, reducing or even eliminating concurrency, but they also don't record the interleavings of actions. When running the transactional code in a debugger like GDB, the concurrency problems related to certain interleavings, such as the one discussed in Section 3.6.1, typically disappear.

To effectively observe these problems it is necessary to: i) allow the transactions to run with minimal intrusion—measured, not only in terms of overhead on the local thread but also on the size of the locked section—a smaller locked section means that other threads wait a smaller amount of time for the lock to be released; ii) check program invariants while the transactions are being run; and iii) be able to observe the past actions of the transactions, and the order of their occurrence.

Tracing Levels

To ease the debugging of the STM engine we have developed a minimal tracing engine that selectively records events in three different layers: application layer, transactional layer; and lock and data layer.

- The application layer information is important to know which piece of code of the application was running. Examples of such information are: "insert element X on the list L".
- The transactional layer information is important to know which step of the transaction was being run. It logs which transactional primitives (*TxStart*, *TxCommit*, *TxAbort*, *TxLoad*, *TxStore*) are being run and the stage of the operation. Some examples are: *executing a transactional load of address X - checking whether variable is locked*; *executing a transactional store - variable is already lock by another transaction - going to abort*.
- The lock and data layer information is the deepest layer of tracing. It is important when the information on the upper layers is insufficient. It traces information related to the acquisition/release of locks, and to the reads and writes of shared variables.

Tracer Internal Structure

To store the information in order of occurrence (total order) we have used a circular buffer of events, where each buffer item holds one event. The access to the buffer is synchronized among all transactions with a CAS instruction on the next buffer item variable. When a transaction needs to add an event to the trace, it first atomically increments the buffer index and then writes the buffer element. When the event buffer is full the next buffer item variable is reset to the first position of the buffer and the oldest events are overwritten.

The usage of the CAS instruction, instead of a lock, was motivated on the need to reduce the overhead and contention of the tracing facility. Using the CAS instruction, the exclusivity time is only one CPU instruction. If transactions T1 and T2 are trying to trace an event simultaneously and the CAS of T1 succeeds, then T2 only has to wait until the CAS of T1 finishes. If, instead, the tracer used lock/read/modify/unlock, then it would have at the very least four CPU instructions of exclusivity—acquire the lock, read the pointer, write the new pointer position, release the lock. With a smaller exclusive section size, the other threads wait, in average, a smaller amount of time for each other. However this architecture does not prevent a high number of cache conflicts on the next buffer item variable—if many transactions are using the tracing facility and each transaction runs on a separate CPU, there is a high probability that the next buffer item variable, will constantly be switching between processor caches.

The tracing facility can trace the following type of information: *thread identifier*, *type of event*, *step* and three optional arguments. The *thread identifier* is a number that uniquely identifies the thread. The *type of event* is an enumeration that identifies the event layer and the operation being performed on that layer. Each operation is composed of one or more steps, the *step* identifies the step of the operation (e.g., within a transactional load, step 20 identifies that the variable is unlocked). The three arguments are numeric and they are operation/step specific, e.g., they may log the address of the variable being loaded, the lock version, or any other information related to the event being logged.

The tracing engine API is shown on Figure 3.17 together with an example of its usage. The example shows a trace call made on the *TxLoad* operation. The thread identifies itself with the *UniqId*, the type of the event is *_tl_tx_load*, the step is number *10* and there are three arguments provided: the address of the read variable, its value and the version of the variable lock.

```
void TraceEvent(int thread_id, int step,
                enum _TraceEventName type,
                intptr_t const volatile *addr,
                intptr_t arg1, intptr_t arg2);

TraceEvent(Self->UniqID, 10, _tl_txload, addr, value, version);
```

Figure 3.17: API and sample usage of the Tracing Engine

Naturally, the tracer can be completely disabled for performance tests. This is done with a preprocessor definition, which is enabled or disabled at compile time, thus reducing the tracer intrusion to zero when the tracer is disabled.

Tracer Output

When one program invariant is known to be broken (assertion failure, segmentation fault, etc), the tracing facility is instructed to dump its contents to a file. The event buffer can also be dumped to a file using the core file and a script for GDB which is available along with the source code.

The trace log file is a simple text file that can be imported by a spreadsheet program like Microsoft Excel or Open Office Calc and display it in tabular format. Using the spreadsheet program's capabilities, the event can be filtered by type or by thread.

There is still a large room for improvement on the tracing facility. A few interesting and useful improvements would be for the tracer to allow other types of arguments (e.g., strings); have automatic race condition detection (e.g. detect a transaction that writes to an unlocked variable); graphical display of the events and their interleavings; creation of a dependency graph of the events; having advanced filtering capabilities; etc.

[This page was intentionally left blank]

Chapter 4

Testing STM Implementations

This Chapter describes some of the problems found while implementing the changes in the STM engine and synthesizes the tests used to reproduce these problems.

4.1 Introduction

The experimental work of porting, extending and testing the TL2 STM engine, caused several complex bugs to show up. From the experience acquired in these experiments we report some of the bugs that we have faced and we synthesize a few testing patterns which aid at finding and reproducing these and other erroneous behaviors.

Parts of this chapter were published in [LC07].

4.2 Terminology

Before showing a sample of the problems found during the development of our version of the STM engine, we describe the terminology used in the examples. Figure 4.1 describes the operations made by the transactions at the transactional level: *TxStart*, *TxLoad*, *TxStore*, *TxCommit*, *TxAbort* and *TxSterilize*. Figure 4.2 describes the operations made by the STM engine at the lock and data level: read variable's lock version, read variable's value, etc.

Symbol	Meaning
$TxStart()$	Start transaction
$TxCommit()$	Commit transaction
$TxAbort()$	Abort transaction
$TxLoad(x)$	Transactional operation to read the value of variable x
$TxStore(x, a)$	Transactional operation to write the value a to variable variable x
$TxSterilize(x)$	Transactional operation to quiesce variable x before it is released (freed)

Figure 4.1: Transaction Construct Glossary—Transactional Level

Figure 4.3 shows the transformation of transactional-level operations (shown in Figure 4.1) into lock-level ones (listed in Figure 4.2). The transformation is simplified, as some internal operations were omitted, like adding elements to read and write sets or internal validation and maintenance operations. The omitted operations are not relevant to the illustration of the bugs reported herein.

$TxStart$ — In both, redo and undo log based STMs, a time-stamp will be associated to the transaction. The time-stamp will be the value of the current global version clock.

Symbol	Meaning
$a = RV(x)$	Read the value of transactional variable x
$v = RL(x)$	Read the lock version of the transactional variable x
$WV(x, a)$	Write the value a to the transactional variable variable x
$Acq(x)$	Acquire the lock of the transactional variable x
$Rel(x)$	Release the lock of the transactional variable x
$IncL(x)$	Increments the lock version of variable x to the current version of the global clock

Figure 4.2: Transaction Construct Glossary—Lock and Data Level

$TxLoad$ — In both, redo and undo log based STMs, loading a variable (memory location) is, basically, a three-step operation: i) load the variable's version counter; ii) load the variable's value; iii) load again the variable's version counter. After these steps it is checked whether that variable's version number hasn't changed between the first and the third step. If it did change, the transaction must abort immediately, otherwise the address is added to the read set. These checks are not relevant for the bugs analyzed in this thesis and were, therefore, omitted in the decomposition of the operation illustrated in Figure 4.3.[1]

$TxStore$ — In redo log mode, changes are made out-of-place. The new variable's value is stored in the redo log, and no memory changes are actually made at this point. The new value will only overwrite the original one when the transaction commits (if it succeeds). If the commit fails then the redo log is simply discarded.

In undo log mode, changes are made in-place. Once the lock of the variable has been acquired, the current variable's value will be stored in the undo log and then overwritten with the new value.

$TxCommit$ — In redo log mode, all the pending updates should now become effective. In this case, commit is a four-steps operation: i) acquire locks for all variables that will be updated; ii) validate the read-set—validating the read-set ensures that all variables read by the transaction satisfy two conditions: they are not currently locked by any other transaction; and they were not changed since the moment they were first read until all the write locks have been acquired; iii) apply all

[1]This *TxLoad* algorithm refers to version used before the performance improvement described in Section 3.5.4 was applied.

Operation	**Decomposition for a redo log mode**	**Decomposition for a undo log mode**
$TxStart()$	$ts = clock$;	$ts = clock$;
$TxCommit()$	Acq(all write-set); RL(all read-set); WV(all write-set, new); Rel(all write-set);	RL(all read-set); Rel(all write-set);
$TxAbort()$		WV(all write-set, old); Rel(all write-set);
$a = TxLoad(y)$	$v1 = RL(y)$; $a = RV(y)$; $v2 = RL(y)$;	$v1 = RL(y)$; $a = RV(y)$; $v2 = RL(y)$;
$TxStore(x, a)$		$Acq(x)$; $WV(x, a)$;
$TxSterilize(x)$	$IncL(x)$;	$IncL(x)$;

Figure 4.3: Simplified decomposition of transactional- into lock-level operations in undo- and redo log mode STMs

pending changes to memory locations (overwrite the memory locations with the values kept in the redo log); and iv) release all acquired locks, changing the version number to the incremented version of the global clock.

In undo log mode, the locks of the overwritten variables were already acquired in the *TxStore* operation and the new values were already written into the variables. It is only necessary to validate the read-set and, if successful, release all acquired locks, changing the version number to the incremented version of the global version clock.

$TxAbort$ — In redo log mode, abort simply discards the redo log, so it is a null operation in what concerns to changes to shared memory locations.

In undo log mode, new values were already written in-place. So, when aborting a transaction, the original values must be restored from the undo log, and the previously acquired locks must be released.

$TxSterilize$ — This function is called before releasing any transactional variable. Both in undo and redo log mode, it prevents all transactions from doing any further reads or writes to that variable. It does so by increasing the lock version of the variable to the current global version clock number.

4.3 Sample of Bugs Found

In the following section we describe interleavings that triggered wrong behaviors in the STM.

4.3.1 Bug 1: Reference to Non-Transactional Memory

Figure 4.4 shows a problem where transaction T1 is accessing a piece of transactional memory already released by another transaction T2. The transactions are operating on a list with three nodes: x, y, and z. T1 is iterating the list and reading the nodes keys. T2 is deleting node y from the list.

	T1	T2	T3	Description
1	$y = TxLoad(x.n)$			get the pointer to node y
	....	....	....	....
2		$y = TxLoad(x.n)$		
3		$z = TxLoad(y.n)$		
4		$TxStore(x.n, z)$		
5		$TxCommit()$		
6		$TxSterilize(y)$		
7		$free(y)$		
	....	....	....	....
8			$new = malloc()$	malloc returns the block released in the previous step
9			$*new = something$	
	....	....	....	....
10	$TxLoad(y.k)$			read y key ... inconsistent read

Figure 4.4: Undo/Redo log mode: Reference to Non-Transactional Memory.

The bug arose because T1 did not detect that its read set was inconsistent when performing step 10, as the memory freed in step 7 had already been recycled for usage on another thread (T3).

The original *TxSterilize* function from TL2, only waited for all writes to variable y to drain, allowing this harmful interleaving. To correct the problem, *TxSterilize* was changed to also increment the version lock of variable y to the current value of the global version clock. In this way, transactions that access y after the sterilization will detect that the version clock has been updated and will, therefore, abort.

4.3.2 Bug 2: Lost Update with Lock Collision

As depicted in Figure 4.5, commit starts by iterating the write set. If the variable is also in the read set then it is a read/write variable, otherwise it is a write only variable. For read/write variables, the lock version is checked against the transaction timestamp, if it is greater the transaction aborts; for write only variables, the algorithm didn't find such check necessary.

```
 1 for each i in write-set {
 2   if (i is not locked){
 3     if(i is also in read set){
 4       // read/write variable
 5       if(get_lock_version(i) > tx_timstamp)
 6         abort;
 7       else
 8         lock(i)
 9     }else{
10       // write only variable
11       lock(i)
12     }
13   }
14 }
```

Figure 4.5: Lock acquisition in redo log mode—buggy version

Figure 4.6 shows a possible interleaving on two transactions T1 and T2. T1 is updating the write-only variable y and the read/write variable x. T2 is only updating variable x.

When operating in redo log mode, the original TL2 algorithm failed with the interleaving shown in Figure 4.6 because of a lock collision. If variable x and y have identical hashes, there will be a lock collision and they will share the same lock in the lock table. With this interleaving, when T1 commits, it starts acquiring the locks according to the algorithm in Figure 4.5. The first iteration of the algorithm finds variable y and since it is a write-only variable, the algorithm goes to step 11 and locks the variable. The second iteration finds variable x and, due to the lock collision with y, it finds the variable x to be already locked and erroneously skips any verification of the lock version.

	T1	T2	Description
1	$TxLoad(x)$		
	....	....	....
2		$TxStore(x, a)$	
3		$TxCommit()$	
	....	....	....
4	$TxStore(y, a)$		y is write only
5	$TxStore(x, a)$		x is read-write
6	$TxCommit()$		
7	$\hookrightarrow Acq(y)$		lock acquisition phase
8	$\hookrightarrow Acq(x)$		
9	$\hookrightarrow RL(x)$		read set validation phase

Figure 4.6: Redo log mode: Lost Update with Small Lock Table

The correction to this problem is to always validate the lock version of read/write variables, even if the lock is already held. Figure 4.7 shows the corrected algorithm.

```
for each i in write-set {
  if(i is also in read set){
    // read/write variable
    if (i is not locked &&
        get_lock_version(i) <= tx_timstamp) {
      lock(i);
      }
    else if(i is locked by this thread &&
      get_lock_version(i) <= tx_timstamp)
      continue;
    else
      abort();
  }else{
    // write only variable
    lock(i);
  }
}
```

Figure 4.7: Lock acquisition in redo log mode—correct version

4.3.3 Bug 3: Dirty Read Not Invalidated when Transaction Aborts

Figure 4.8 shows an example of a hidden dirty read in undo log mode. Transaction T1 reads the variable x and commits, and transaction T2 writes to the same variable and aborts. In step 2, T1 reads the lock version of variable x. Then T2 stores a new value on x. In step 6, T1 reads (dirty read) the value of x after changed by T2. In step 7, T2 aborts and the old value of x is restored.

	T1	**T2**	**Description**
1	$TxLoad(x)$		T1 loading variable x.
2	$\hookrightarrow RL(x)$		
	....	....	....
3		$TxStore(x, a)$	
4		$\hookrightarrow Acq(x)$	
5		$\hookrightarrow WV(x, a)$	new value is written
	....	....	....
6	$\hookrightarrow RV(x)$		dirty value is read by T1
	....	....	....
7		$TxAbort()$	T2 aborts
8		$\hookrightarrow WV(x, old)$	old x value is restored
9		$\hookrightarrow Rel(x)$	
	....	....	....
10	$\hookrightarrow RL(x)$		lock version revalidation
11	$TxCommit()$		

Figure 4.8: Undo log mode: Dirty read not invalidated when transaction aborts

The bug in this situation was that, when transaction T2 aborted (in undo log mode) the value of variable x was restored and the lock was simply being released with the old version number. Transaction T1 was not detecting that it read a dirty value because the lock version revalidation, in step 10, returned the same value as in step 2, therefore assuming the value read in step 6 was valid. To correct this problem, when aborting a transaction in undo log mode, the lock version of every variable in the write set must be incremented.

4.3.4 Bug 4: Lost Update on Lock Upgrade

Figure 4.9 shows a problem that happened on undo log mode, when reading a variable and then modifying its value. Transaction T1 is incrementing the variable x and transaction T2 is storing a new value in the same variable. The problem was that *TxStore* was not validating if the lock version was the same as the one obtained in the first *TxLoad*—it was merely acquiring the lock and writing the value. The correction was to force *TxStore* to validate the lock version before writing to the variable.

	T1	T2	Description
1	$v = TxLoad(x)$		
	....	....	....
2		$TxStore(x, w)$	
3		$TxCommit()$	
	....	....	....
4	$TxStore(x, v + 1)$		upgrade from read access to write access

Figure 4.9: Undo log mode: Lost update on lock upgrade

4.4 Testing Patterns

Testing the STM engine aims at incrementing the probability of generating harmful interleavings. Harmful interleavings are those that improperly read and/or modify the shared data, i.e., locks and transactional variables. Such interleavings can occur during reads, writes/updates, commits, aborts, and when adding and removing variables from the transactional space.

Tests may target specific implementation options, such as the bug described in Section 4.3.2, or concurrency control errors, which depend on both implementation options and execution environment.

4.4.1 Very Short Transactions

This testing pattern aims at maximizing the interleavings between the main transactional operations, i.e., reads, writes, commits and aborts. Traditional transactions execute much more load and store operations than commits and aborts. To stress test the commit algorithm it is useful to increase the frequency of its occurrence and we can do that by using very short transactions. To improve the efficiency of the test, it should be used read-only, write-only and read-write variables as some engines treat these variable in a different manner.

An example of this pattern is a list with a header and single node. The concurrent transactions are continuously updating the node and/or reading its contents. The transactions are very short as there is only one node, therefore the ratio of time spent in the commit of the transaction over the time spent in the body of the transaction is higher than it would if the transaction was longer. We found this example useful to find the bug reported in Section 4.3.2.

This pattern works best for redo log based STM engines. Such systems only change the shared state on commit, shortening the time-window in which the transaction runs with its shared state changed and concurrency errors can only be revealed once there are changes in the shared state.

4.4.2 High Frequency of Variables Being Added and Deleted

A testing pattern aiming at stressing the variability of the transactional space, by repeatedly inserting and removing data to/from the transactional space.

Such pattern allows to detect bugs mostly related to transactions holding pointers to variables being simultaneously released by other transactions, such as the bug described in Section 4.3.1. These bugs may cause invalid memory accesses and memory being read/written after data deletion.

An example of this pattern is an implementation of a transactional list, where several concurrent transactions are continuously adding and removing nodes.

4.4.3 High Number of Updates on a Small Number of Variables

This testing pattern aims at generating a very high frequency of collisions between transactional reads and writes, also forcing transactions to abort very frequently. This pattern can be instantiated with a list that can hold a very small amount of nodes (e.g., ten) and with several concurrent (e.g., five) transactions trying to get and update the list nodes.

This test produced very good results with undo log strategy, because they change the shared data on writes (locks and data), on commits (locks) and on aborts (locks and data). Overall, this testing pattern was found to be very effective.

This pattern was particularly useful to find the bugs reported in Sections 4.3.2 and 4.3.3, as these bugs are related to read/write collisions.

4.4.4 Small Lock Table

STM engines that use a lock table to store objects/data locks, usually make use of an hashing function to map the object address to its lock within the lock table. Such hashing function may map several objects to the same table position, originating a lock collision problem. Such lock collisions may cause an improper validation of the lock state, with transactions never being able to commit and potentially running into livelocks.

This pattern aims at maximizing the function $f_L = \frac{V*T}{L}$, where V is the average number of transactional variables being manipulated by each transaction, T is the number of running transactions and L is the size of (number of entries in) the lock table. The pattern can, therefore, be instantiated by an adequate combination of i) using a smaller lock table; ii) increasing the number of transactional variables; and iii) increasing the number of running transactions. This pattern contributed significantly to find the bug reported in Section 4.3.4. Just by decreasing the size of the lock table to a very small number it was possible to have a significant number of lock collisions and reproduce the harmful interleaving.

4.4.5 More Concurrent Transactions than CPUs

If the number of transactions is less than the number of CPUs, any transactions willing to run can be immediately assigned to a CPU, and transactions will never be stalled waiting for CPU. In such case, some interleavings will be harder to reproduce, because they depend on transactions being preempted and stalled for some time. Using more threads than CPUs causes some of the threads to be preempted for large amounts of time, potentially while holding locks.

This pattern was useful to reproduce the bug reported in Section 4.3.3.

4.5 Conclusions

These testing patterns are a simple way to find bugs in STM engines. In the case of our prototype, they were helpful as we could find several problems just by using these techniques.

There is a lot of work that can be done in this area specially in finding more testing patterns.

[This page was intentionally left blank]

Chapter 5

Prototype Validation

This Chapter describes the tests and experiments made with the prototype and shows the obtained results.

5.1 Introduction

During and after developing the enhancements to TL2, several tests were created and executed to verify the engine's functionality, stability and performance. The tests were made with a red black tree and a sorted list test harnesses, which are two of the most common test sets in the literature.

Functional tests were made with a simple functional test harness consisting of several test cases, aiming at each functionality in the prototype.

Stability tests were made with the performance test harness by running several load patterns, several number of threads, several operating systems (Linux and Solaris X86), and several machines with different hardware configurations. Each version of the prototype was left running to find errors. The latest version was left for more than seven days in a row without errors.

The performance tests were aimed at validating the performance on several scenarios. First we compared the performance of the prototype implementation options. Next we tested against the original TL2 implementation. Finally against another STM engine–the Robert Ennals STM engine [Enn06]. The Ennals STM implementation was chosen because it shows great performance results and it was also as a performance baseline by TL2 authors.

5.2 Description of the Tests

We decided to use a test harness similar to the one used by the TL2 authors. The tests were made with a Red Black Tree implementation based on the one found on TL2 package, which in turn is based on the *java.util.TreeMap* implementation. However, several modifications were made to adapt it to use the different API of object mode as well as the API of Ennals STM. We have also created a variant of the tests with an implementation of a sorted list.

The tests consist on series of operations on a set. The set is either a Red Black Tree implementation or a Sorted List implementation. Both implementations have three methods—*put*, *delete* and *get*. The set elements have a key and a value and all methods are indexed by the key. Duplicate keys are not allowed and adding an element with an already existing key just updates its value.

The sorted list implementation is a standard double linked list created from scratch. The insert operation places the nodes sorted by key; the get operation runs through the list until it finds the element; the *delete* operation, first gets the element and then updates the pointers of the adjacent nodes.

5.2.1 Test Harness Implementation

The test harness is divided in two components: the set implementation (which may be the sorted list or the red black tree implementation); and the harness launcher. The harness launcher starts a number of threads in parallel. Each thread continuously loops between: i) choosing an operation to execute (*put*, *delete* or *get*) based on a random number; ii) calculating a random key; and iii) executing the selected operation on that key. The probability of *put*/*delete*/*get* operations is chosen via a command line argument. Also the key range is passed as an argument and it limits the number of elements the set can have, e.g., a key range of 1000 means that the set can have elements with keys from 0 to 999, thus the maximum number of elements in the set would be 1000.

The key range largely affects the contention on the set. With a low key range, the probability of having more than one thread reading, writing or deleting the same node increases. Therefore the key range has a big impact on overall contention. A second effect of the key range is that it increases the search time for an element, this is specially relevant for the list implementation, where the average search size (and time) is proportional to the key range.

The tests have been made using the several possible combinations of the prototype: word based mode with redo log; word based mode with undo log; object based mode with undo log. The word based modes always run in consistent states, the object based modes have three variants regarding consistent state validation: *full state validation*, *partial state validation* and *no state validation* (see Section 2.3.5). Also the object based mode has two options regarding lock placement: in a separate table or adjacent to the object. In summary all tested combination are shown in Table 5.1.

The tests were made on a 2 way Intel Xeon CPU@2.66GHz—Dual Core, making four processing cores. Although with these machines we can't evaluate the scalability of the selected alternatives, we can have a glimpse on its behavior on a small scale scenario.

On these tests the STM engine, as well as the harness, were compiled for X86 32bits on a Linux operating system with kernel version 2.6.18. The compiler used was GCC version 4.1.2, the optimization flag used was *-O3* and the debugging and profiling flags were disabled. Our tracing engine was also disabled.

5.2.2 Test Parameters

The test parameters chosen to run the harness are basically the same used on TL2 tests. It includes a small (200 keys) and a large set (20.000 keys). The small set represents a high contention structure and the large set represents a low contention structure. Each

Short name	Prototype combination
word/tab/redo	Word based mode with redo log, full state validation and lock in a separate table.
word/tab/undo	Word based mode with undo log, full state validation and lock in a separate table.
object/tab/undo/fv	Object based mode with undo log, full state validation and lock in a separate table.
object/tab/undo/pv	Object based mode with undo log, partial state validation and lock in a separate table.
object/tab/undo/nv	Object based mode with undo log, no state validation and lock in a separate table.
object/adj/undo/fv	Object based mode with undo log, full state validation and lock adjacent to the object.
object/adj/undo/pv	Object based mode with undo log, partial state validation and lock adjacent to the object.
object/adj/undo/nv	Object based mode with undo log, no state validation and lock adjacent to the object.

Table 5.1: Tested prototype combinations.

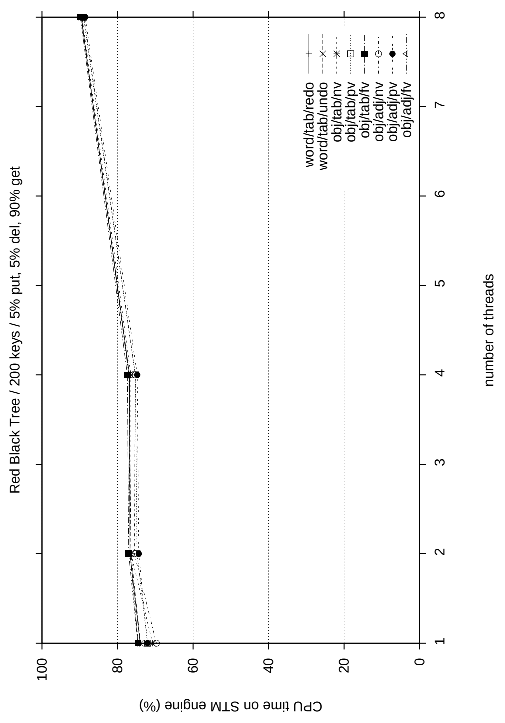
set is subject to two load patterns, one with mostly reads, other with a higher write percentage. The first pattern has 5% puts; 5% deletes; 90% gets, which we call the read pattern. The second pattern has 20% puts; 20% deletes; 60% gets, which we call the write pattern. These patterns intend to show the difference in behavior with varying proportion of reads versus writes.

The number of running threads included 1, 2, 4 and 8 threads. Until 4 threads, the intent is to study the performance increase of the test harness with the number of available CPUs. With 8 threads, the intent is to investigate whether there is a performance decrease by having more running threads than CPUs.

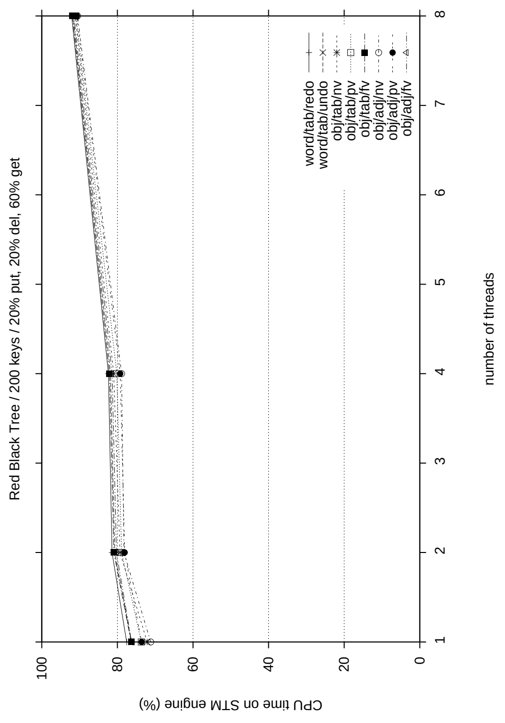
5.3 Test Harness Overhead

Before evaluating the test results, we start with the evaluation of the test harness quality in terms of overhead and impact on the test results.

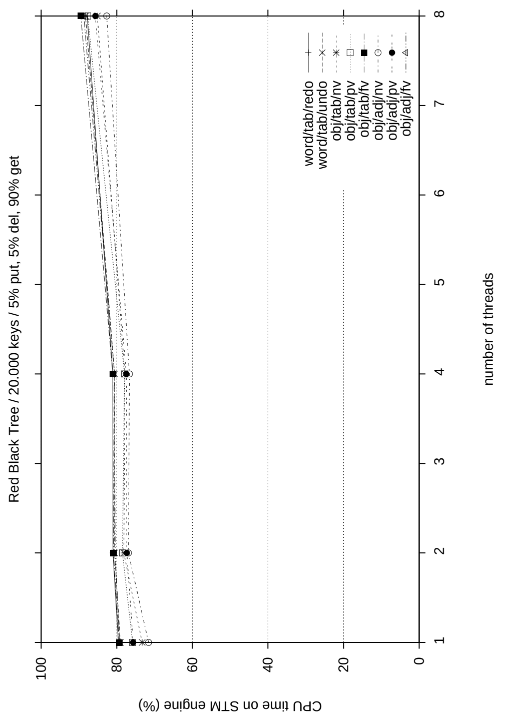
In the test harness there are three layers running: the harness launcher, the set implementation and the STM engine. Since we intend to verify the performance of the set implementation using the STM engine, one requirement on the test harness launcher, is to be as light as possible to avoid it from hiding the true subject of the test. Therefore it is desirable that the CPU time spent in the harness launcher is significatively less than the running time of the set implementation plus the STM engine. Figure 5.1 shows the percentage of time spent in the set implementation plus the STM engine, the remaining time is spent on the harness. As it can be seen the time spent on the harness is less than 25-30%, which leaves a solid margin for testing the set implementation with the STM engine.



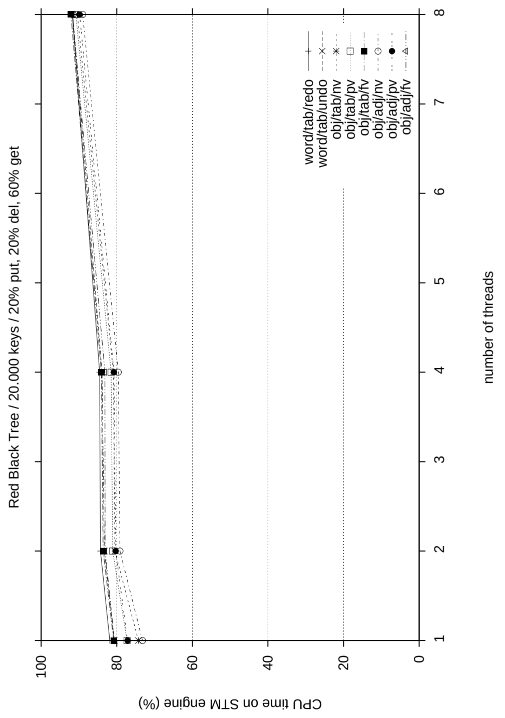
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get

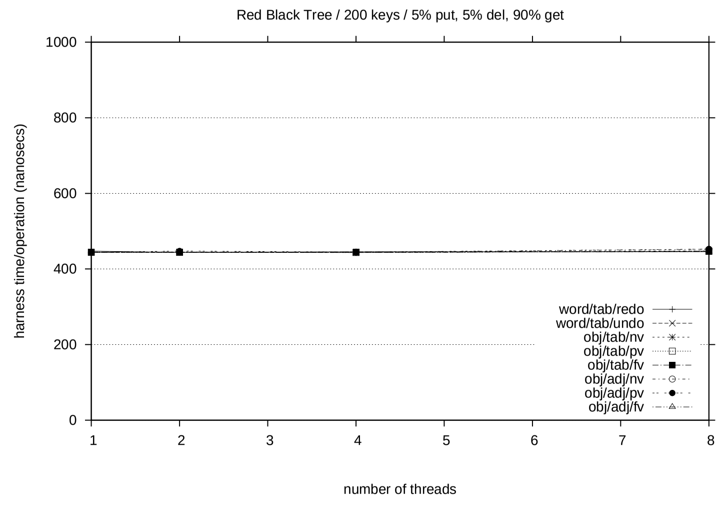


(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

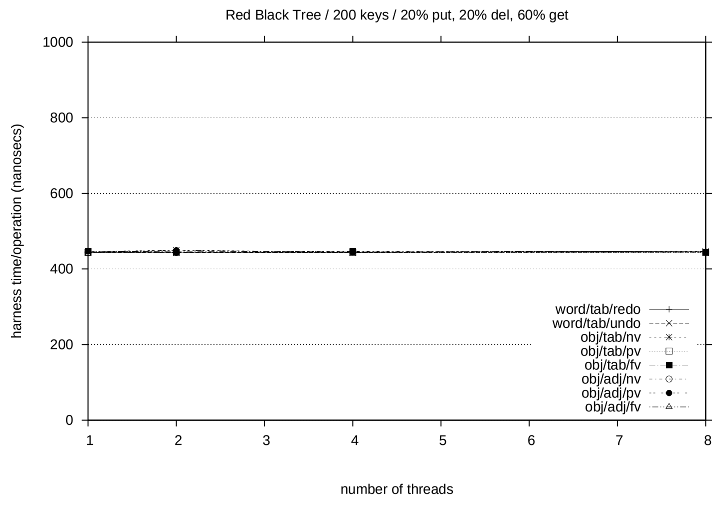
Figure 5.1: Percentage of time spent in the harness

Another useful requirement on the test harness is for the time per iteration spent in the harness launcher to be constant. This eases the speedup analysis and shows the real performance difference of the test subject between the various test runs. If, for instance, there was a consistent cache collision effect (e.g., false sharing) on the test harness, the time per iteration spent on the harness would increase with the number of CPUs and it would hide the real performance increase/decrease of the overall test. Therefore, a good quality harness should have a constant time per iteration. When this test harness was built, it was ensured that there were no read-write variables shared among threads (to avoid cache coherency traffic on the shared BUS) and the local variable were properly padded to avoid false sharing.

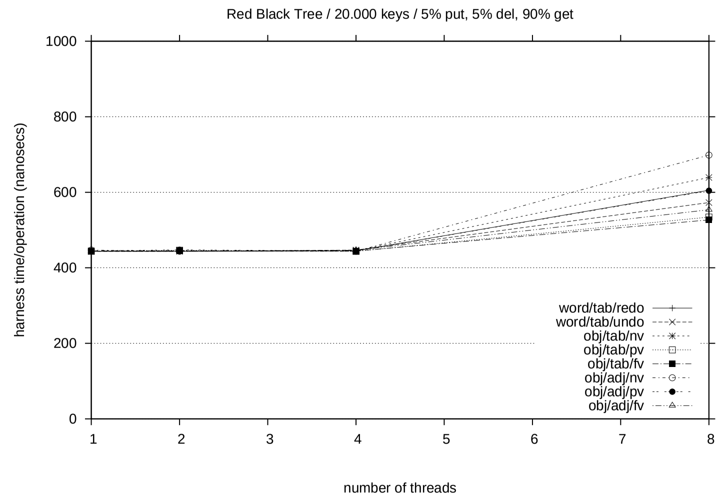
Figure 5.2 shows the time spent on the harness launcher per operation. As can be seen, the time is stable at 450 nanoseconds with up to four threads. With more than four threads the results are not significative because the time measurement is made between the start and the end of the harness operation. If some other thread preempts the CPU while the harness is running, the whole scheduler quantum of the other thread is counted as harness time. When there are more threads running than available CPUs this situation become a lot more frequent, specially when transactions are longer, which is the case on Figures 5.2c and 5.2d.



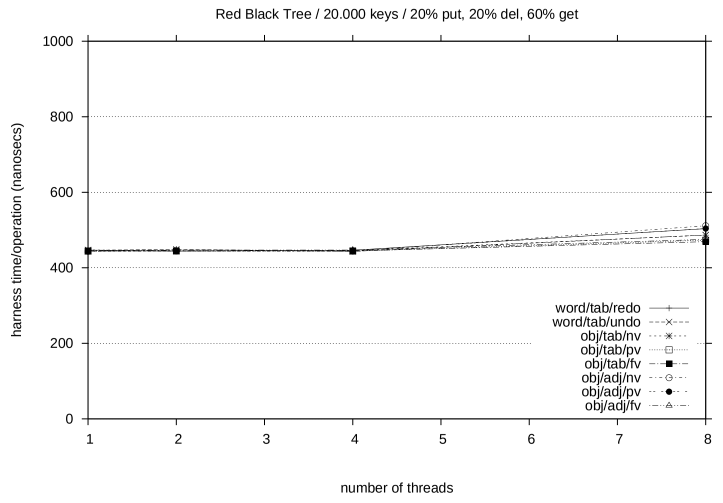
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

Figure 5.2: Test Harness Launcher Overhead

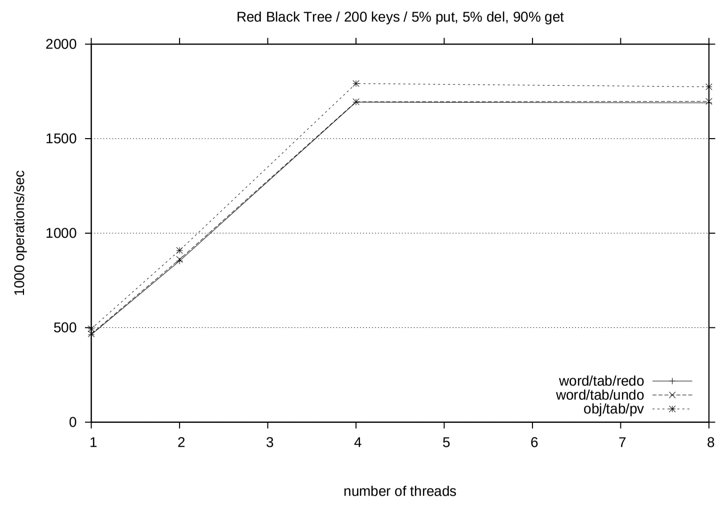
5.4 Test Execution

We now describe the executed tests and evaluate the results. Our benchmarks allow us to evaluate the performance of the combinations of our prototype, compare the performance against another STM engine and evaluate the gains achieved by the changes we made. Regarding the prototype combinations we compare the undo against the redo log strategies; we evaluate the cost of doing consistent state validation in all three modes—*full validation*, *partial validation* and *no validation*; we evaluate the difference between having a lock table and the lock adjacent to the object; and we evaluate the performance when working in the different block sizes.

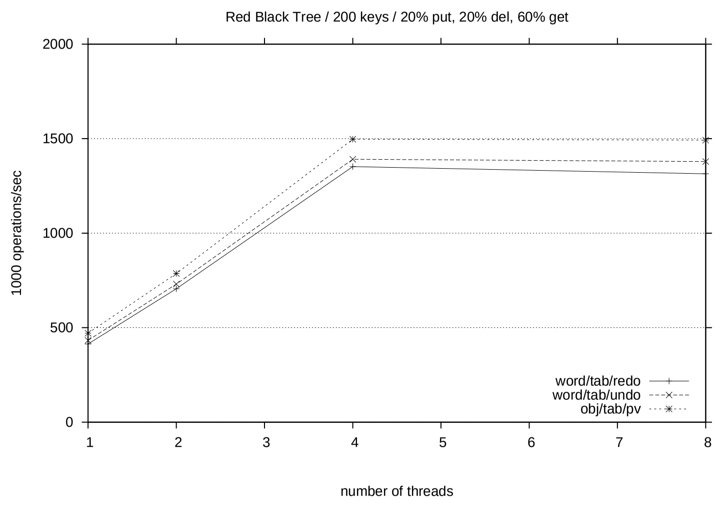
We did not execute benchmarks of our prototype against an implementation using locks. Several benchmarks comparing the performance of STM engines against several types of locks (coarse grained, fine grained, spin locks, Mellor Crummey and Scott locks [MCS91], etc) can be found on [DS06, DON06, DS07, SATH+06, HF03].

5.4.1 Comparing Undo/Redo, Word/Object modes

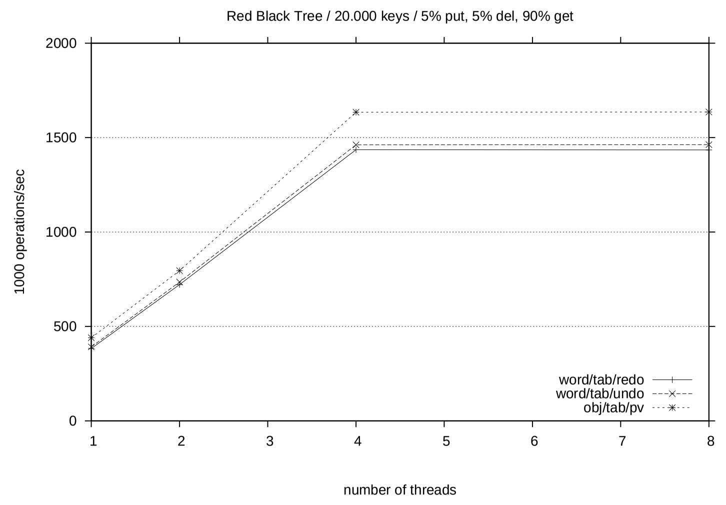
The first set of tests (Figure 5.3) compares the performance of undo/redo log strategies and word/object based modes.



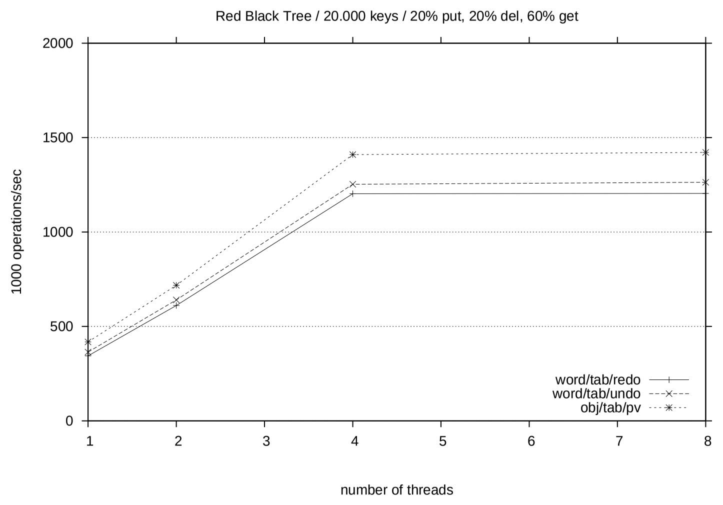
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

Figure 5.3: Evaluation of implementation alternatives - Red Black Tree

These tests were performed using the three combinations of our prototype. The Object based STM tests were performed with *partial validation*.

All tests show that word based/undo log and word based/redo log strategies have similar performance under all scenarios, although the undo log scheme generally has a short advantage.

Tests also show that the object mode with *partial validation* out-performs the others. This is due to the lower overhead of object mode, where there is only one transactional operation per node instead of one per field and the reduced number of state validations.

When running with more threads (8) than CPUs (4), the performance is not affected.

With the list implementation the results (Figure 5.4) also show an advantage of the object mode with partial state validation.

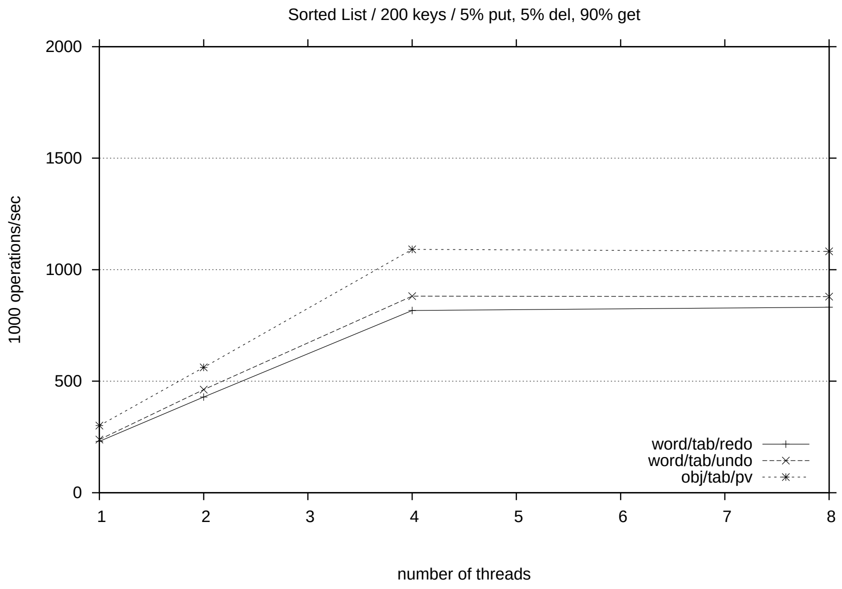
5.4.2 The cost of consistent state validation

The second set of tests evaluates the cost of consistent state validation in undo log/object mode. Figure 5.5 compares the performance of the three alternatives—*full validation*, *partial validation* and *no validation*. The graphics show that *full validation* has 5% to 20% less performance than the other combinations.

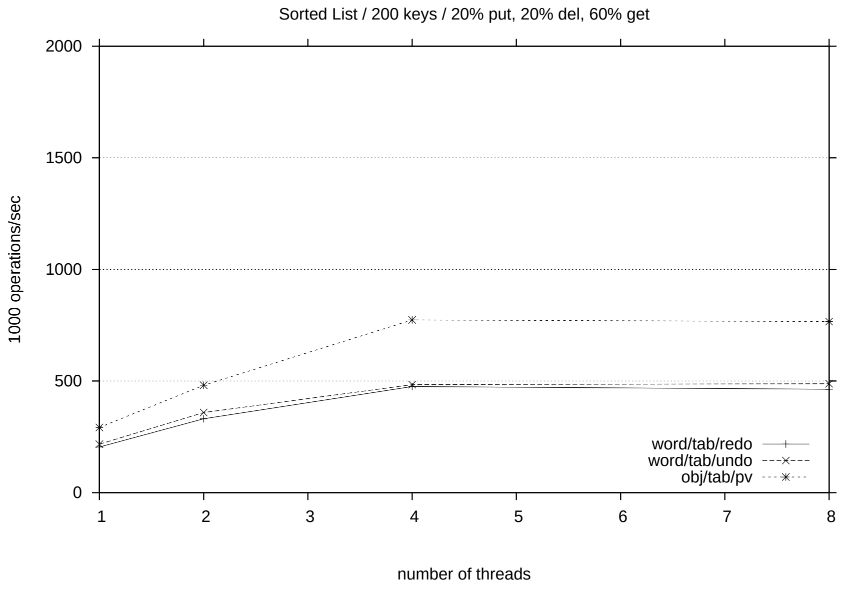
The tests also indicate that the relative cost of *full validation* decreases with the number of threads. Although the performance difference between *full validation* and *no validation* is higher with four threads in absolute terms, the relative difference is smaller. The reason is that with consistent state validation, transactions detect inconsistent states sooner, thus they do less useless work. This is confirmed by the abort time graphics on Figure 5.6.

Aborts may occur during a transactional load, store, during a verification, during a commit or while handling a fault (e.g., segmentation fault). It is preferable that a transaction aborts early when the transaction starts running rather than later, when the transaction has already made a lot of work. In terms of performance, the worst possible time for a transaction to abort is at commit time because it has consumed all resources it needed, but unfortunately it finished in a state where it can't commit. The graphics on Figure 5.6 shows the percentage of aborts at commit time. In the *no validation* combination, the percentage of commit time aborts is always higher than 80%, whereas on *partial validation* and *full validation* the number of commit time aborts is always lower than 40%.

Having much more commit time aborts explains why the *no validation* combination is overtaken by the *partial validation* combination on the small tree from 2 threads onwards. When there is only one thread, the *no validation* combination has the best



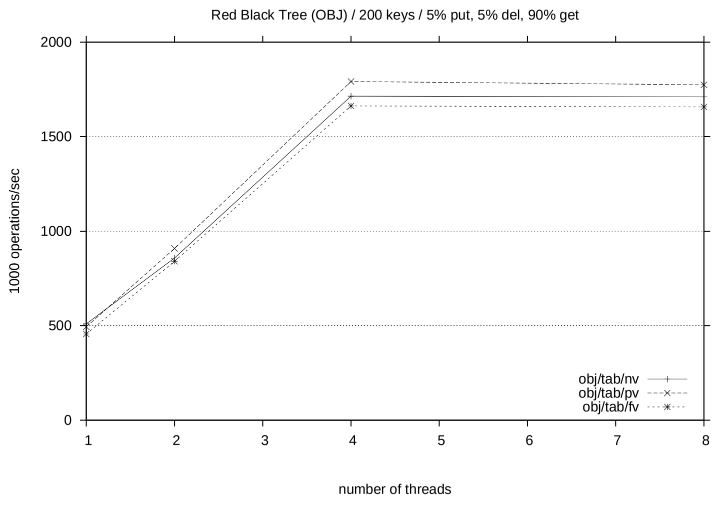
(a) Sorted lists / 200 keys / 5%put 5%del 90%get



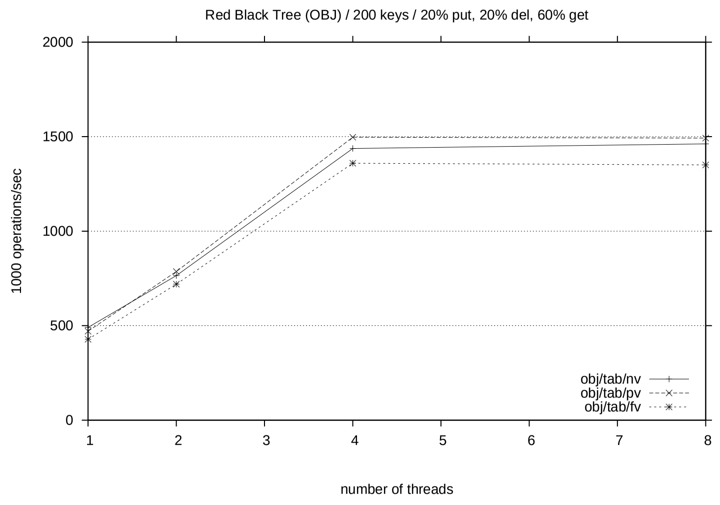
(b) Sorted lists / 200 keys / 20%put 20%del 60%get

Figure 5.4: Evaluation of implementation alternatives - Sorted List

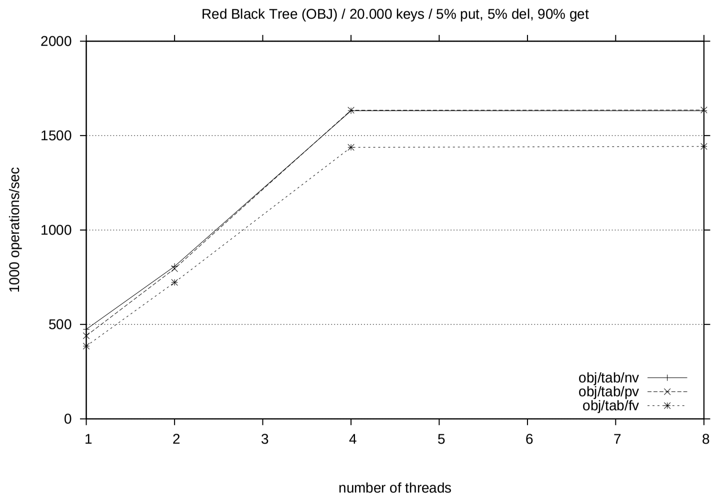
performance, but despite having less overhead, it does more useless work when there is more than one thread, leading to a poorer performance.



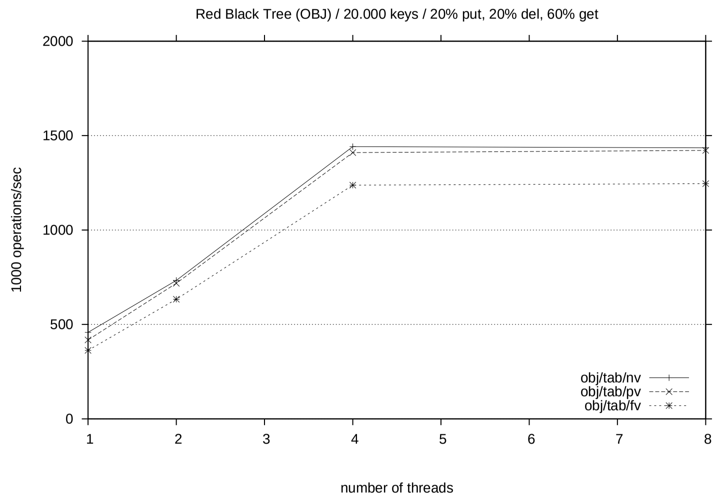
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get

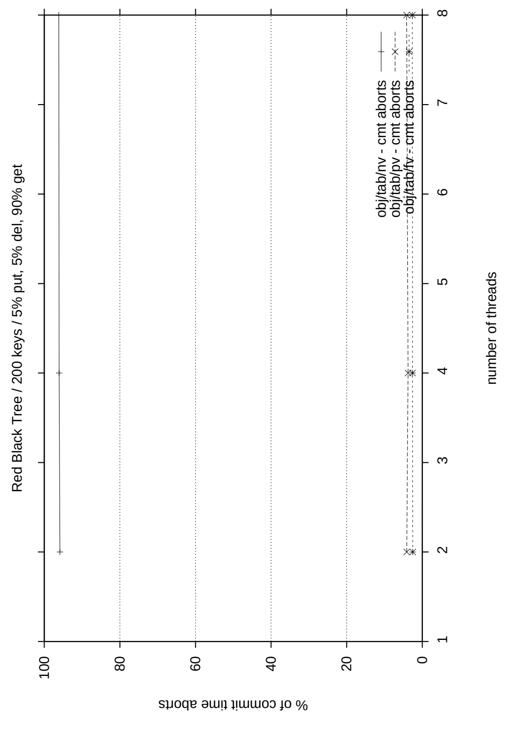


(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get

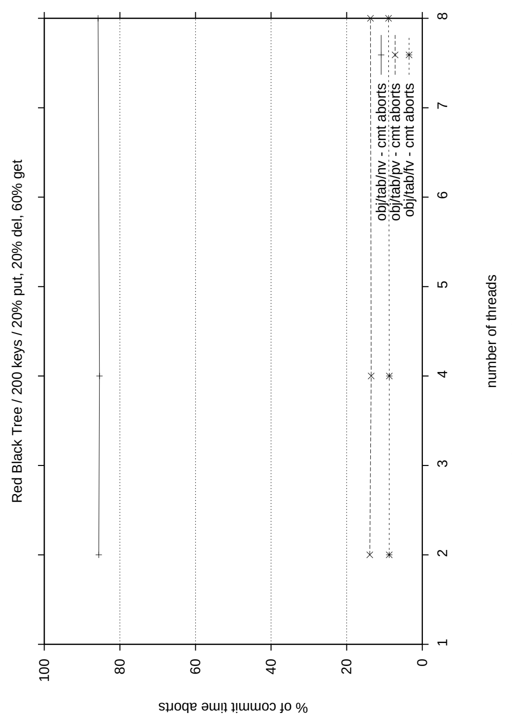


(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

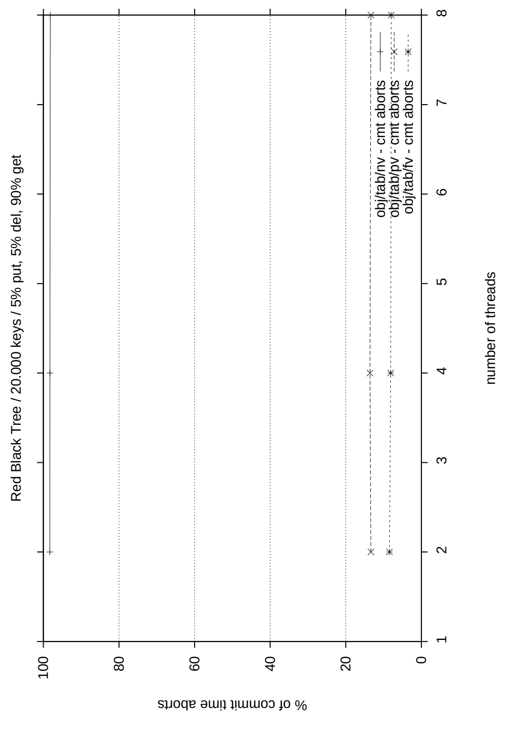
Figure 5.5: Consistent state validation alternatives - Red Black Tree



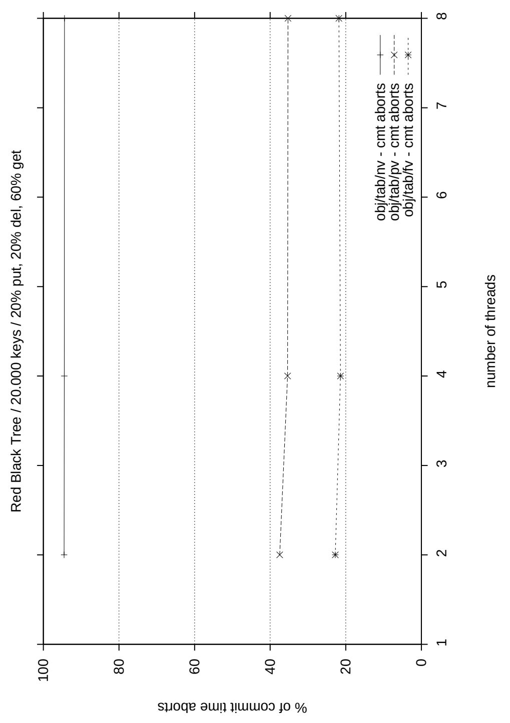
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



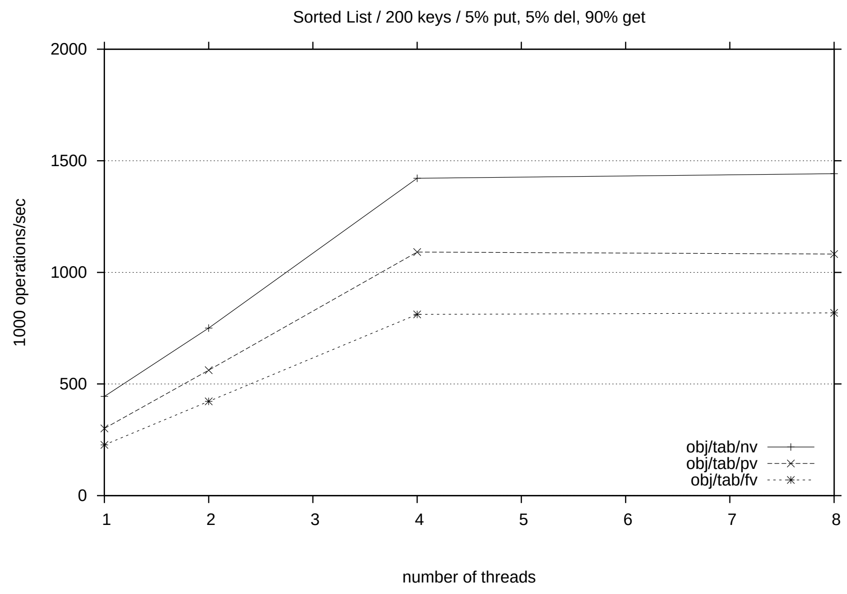
(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



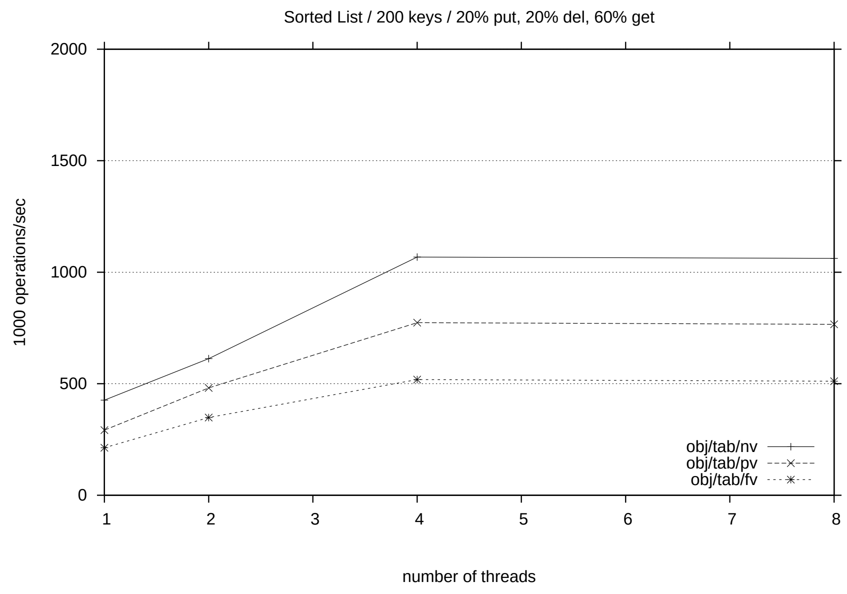
(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

Figure 5.6: Abort Time

The results with sorted lists are clearly different. With a sorted list, the *no validation* scheme is always faster as shown on Figure 5.7. Here the validation overhead drives the result of the test, as the number of operations per transaction is much larger than on the red black tree. The number of operations is $O(k)$, where k is the key range, whereas on the red black tree the number of operations is $O(log(n))$.



(a) Sorted lists / 200 keys / 5%put 5%del 90%get



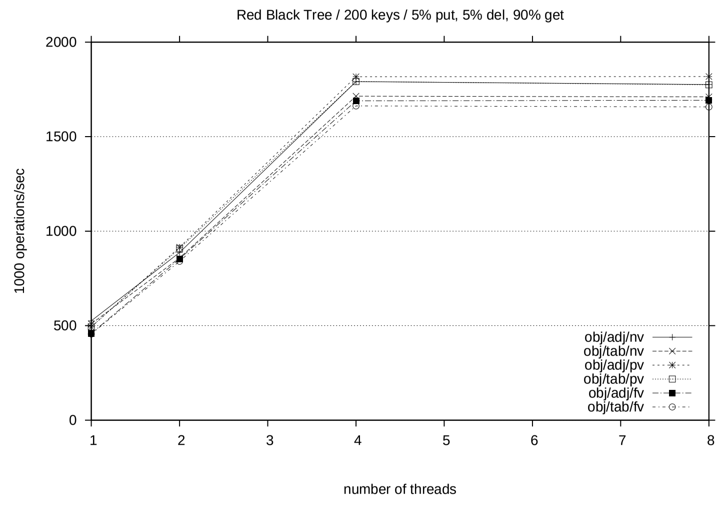
(b) Sorted lists / 200 keys / 20%put 20%del 60%get

Figure 5.7: Cost of validation - Sorted List

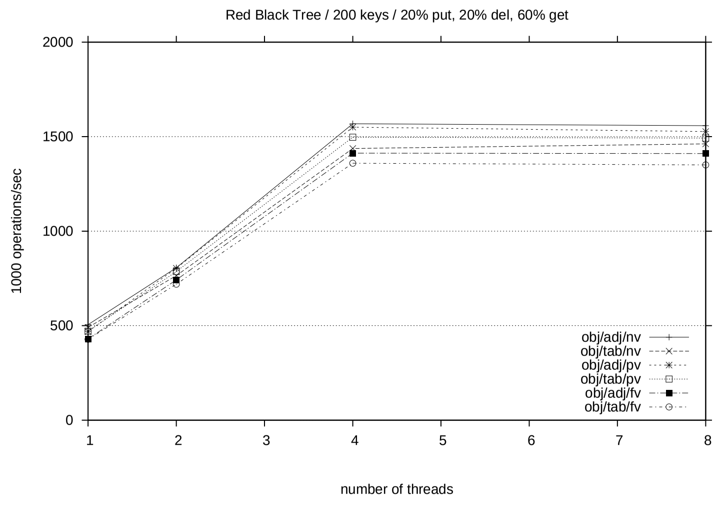
5.4.3 Lock adjacent to the data

This test was made on the object based/undo log mode and it evaluates the performance gain of having the lock adjacent to the object, versus having a lock table.

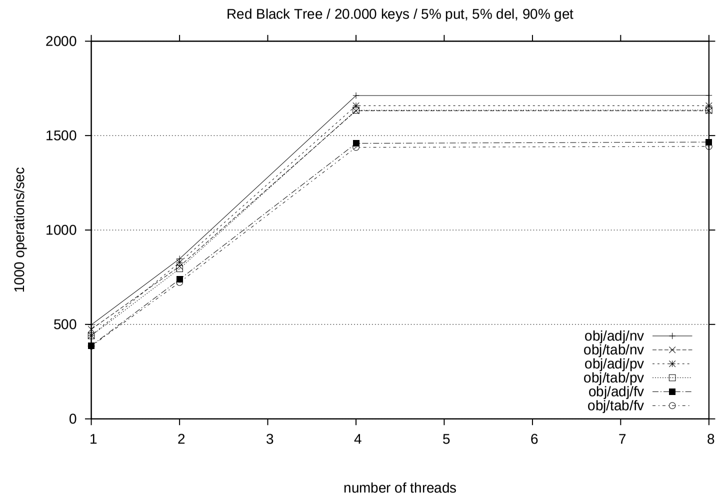
The benchmarks with the lock adjacent to the data (Figure 5.8) show a small performance improvement of less than 10% over the lock table. This result was confirmed with the tests made on the sorted list implementation shown on Figure 5.9.



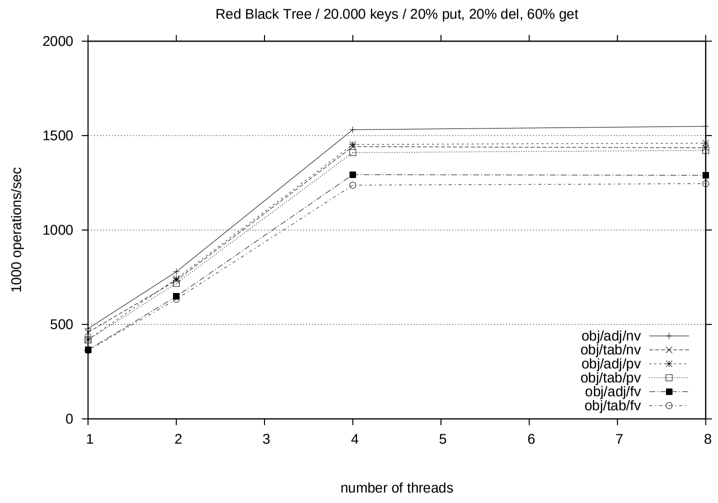
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get

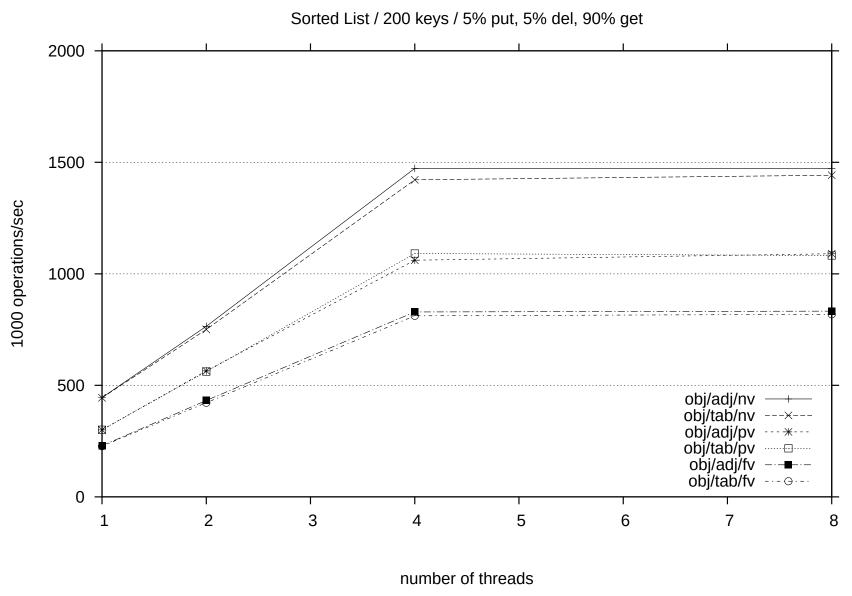


(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get

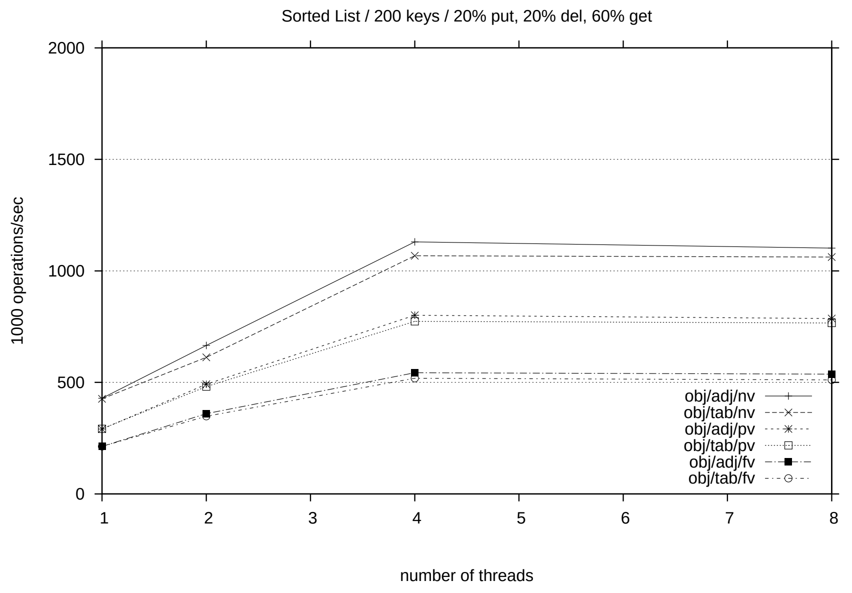


(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

Figure 5.8: Adjacent Lock vs Lock Table - Red Black Tree



(a) Sorted lists / 200 keys / 5%put 5%del 90%get



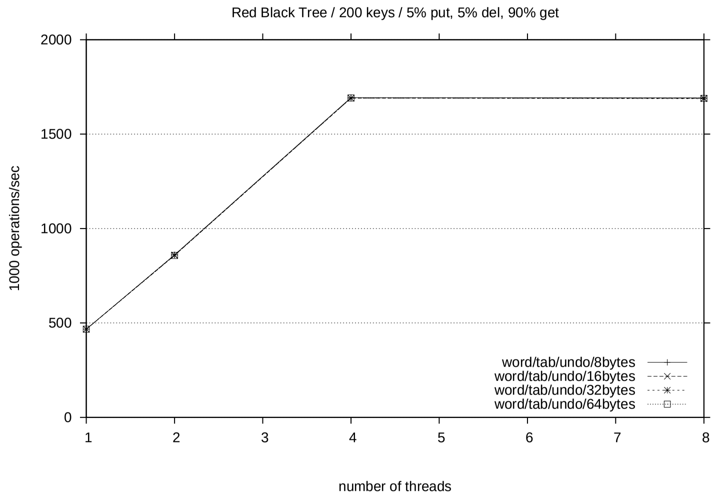
(b) Sorted lists / 200 keys / 20%put 20%del 60%get

Figure 5.9: Adjacent Lock vs Lock Table - Sorted List

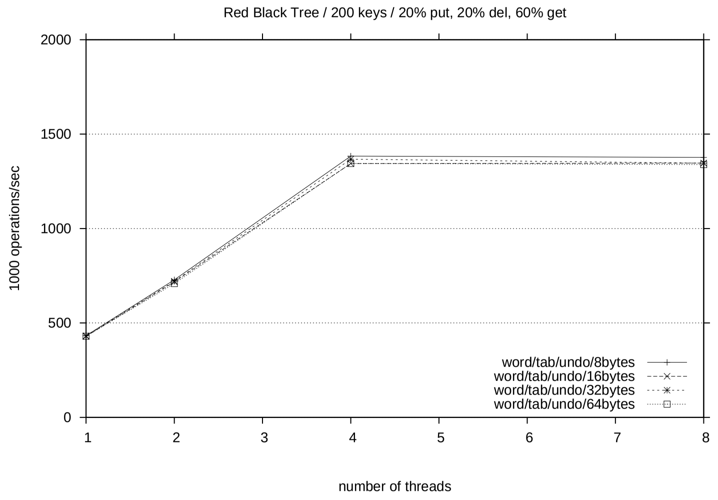
5.4.4 Different Block Sizes

Some STM engines [SATH+06] use this strategy to reduce the number of locks in the system and reduce the overhead of locking several variables. We have implemented this strategy in our prototype, but the results are disappointing. As shown on Figure 5.10 the results are nearly the same, there is no performance difference between them. If, for instance, an object fits inside a block, a first write to that object will lock the entire block, a second write will find the block already locked and doesn't need to lock it again. However, the STM engine still has to verify if the lock is held, therefore the performance advantage is not so big to be noticeable.

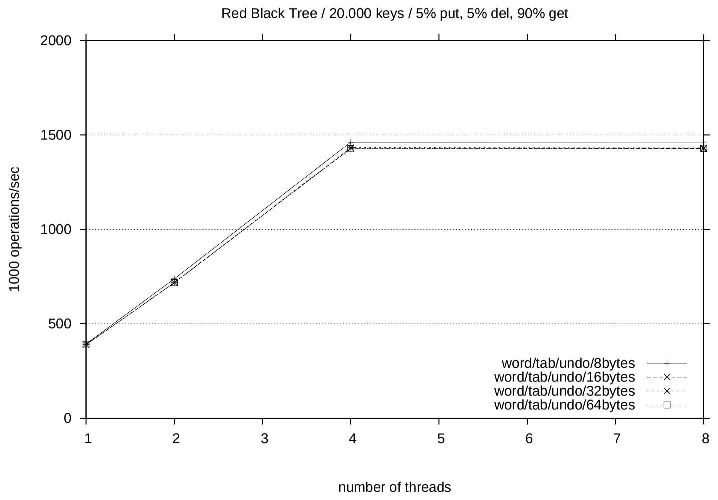
One change that could take advantage of the bigger block size would be to have a runtime log filter as described in [HPST06]. The optimization described there, reduces the number of entries in the read and write set by detecting duplicate entries if, for instance, a variable is loaded twice. With this filtering optimization and a bigger block size, the size of the read and write sets could be further reduced, and it could eventually reduce the read-set validation overhead, speeding up the transactions. This may be an interesting line of investigation.



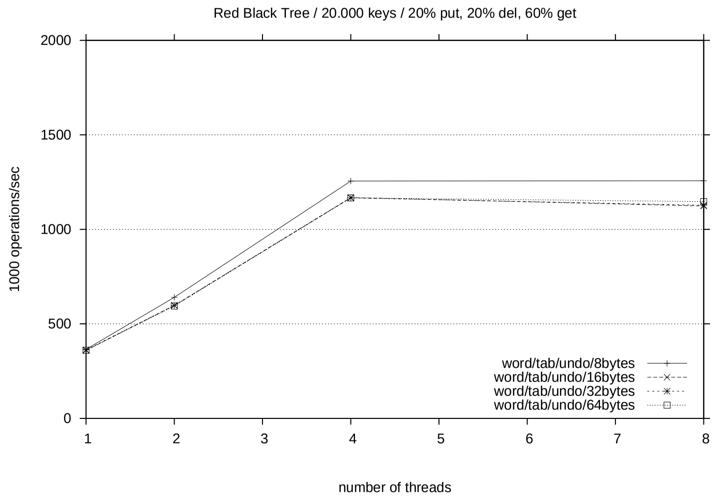
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

Figure 5.10: Comparing different block sizes.

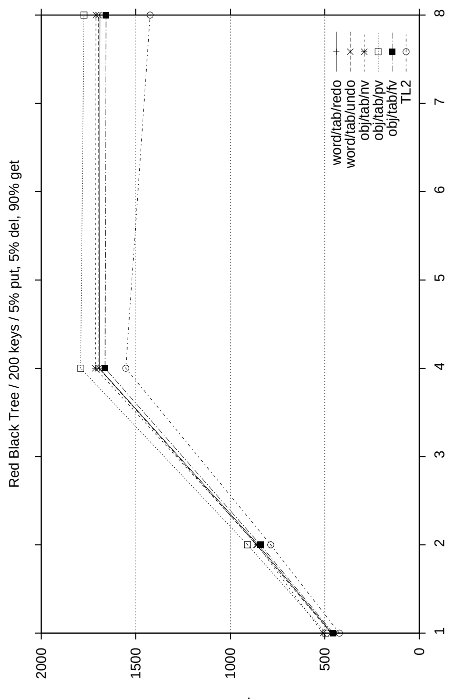
5.4.5 Comparing the performance against the ported TL2

As said before, the original TL2 was developed for Sun Solaris with SPARC architecture and using the SUN Pro C compiler. During this thesis we had no chance of testing the prototype with this platform, this is why we have ported TL2 to Linux/X86/GCC. This thesis was born after the port to our platform, however we have kept the ported version without additional changes to serve for baseline performance analysis.

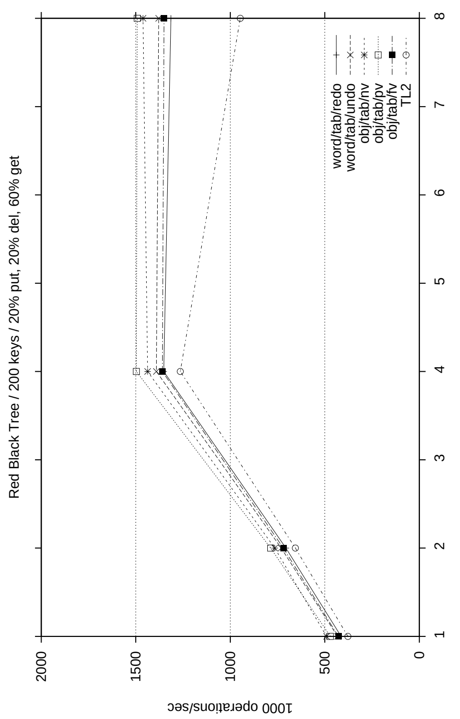
In this set of tests we have compared the performance of the ported TL2 (the version with the minimal set of changes necessary to run on the available configuration) against the fully modified prototype of this thesis. It is therefore, unfair to say that the comparison is against the original TL2 because of the necessary changes to make it work on our platform. The changes made to the ported TL2 were described in the Sections 3.3.1 and 3.3.2—X86 assembly instructions; removal of the *schedctl* mechanism; and replacement of the non-faulting load instructions with fault handlers.

Figure 5.11 shows the test against the ported TL2. There is a performance improvement seen on all the combinations.

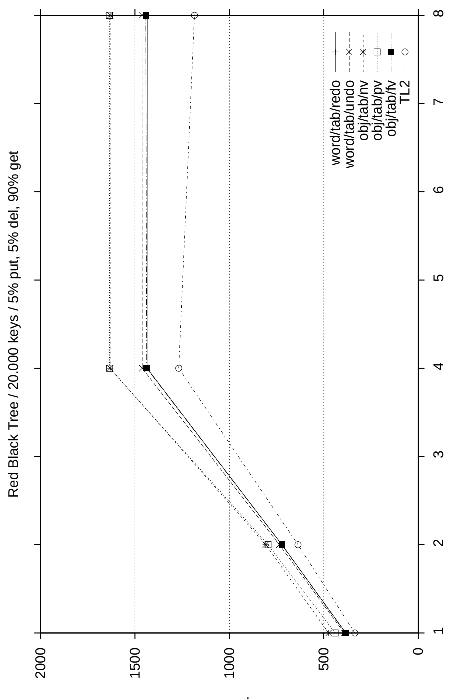
The test also shows that the ported TL2 is less resilient to overload—when running the prototype with more threads than available CPUs. On all tests, the performance of the ported TL2 drops 10-20%, whereas on our prototype, the performance drop is negligible on any of the tested combinations. The reason is related to the reduced cache coherency traffic on the bus due to the improved *TxLoad* algorithm, in which reads the shared lock only once.



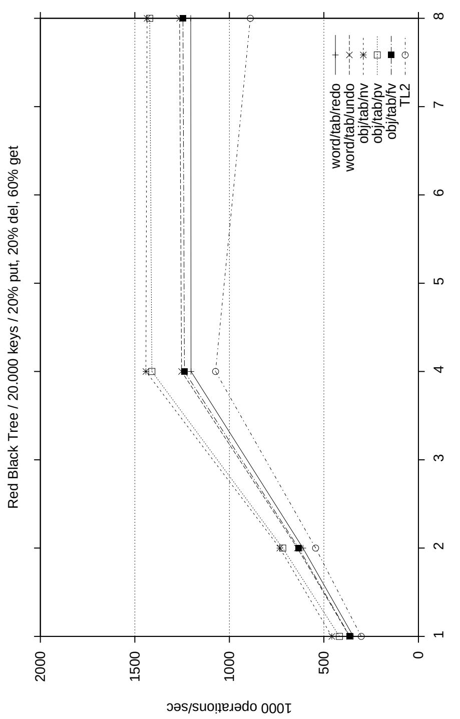
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

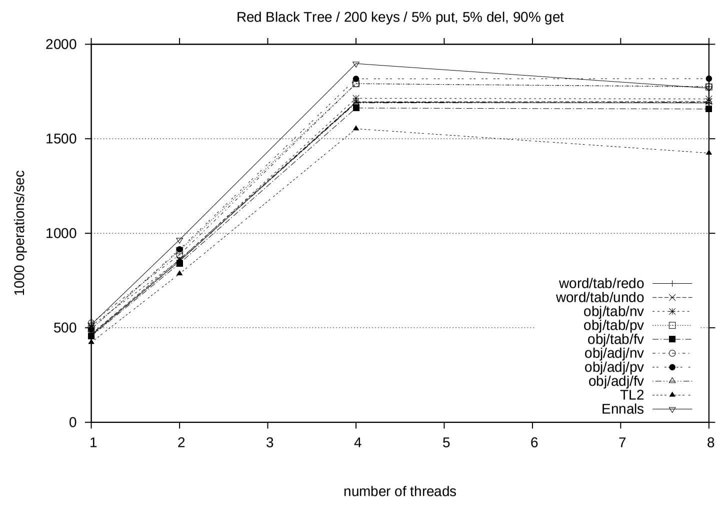
Figure 5.11: Ported TL2 vs improved prototype

5.4.6 Comparing the performance against Ennals STM

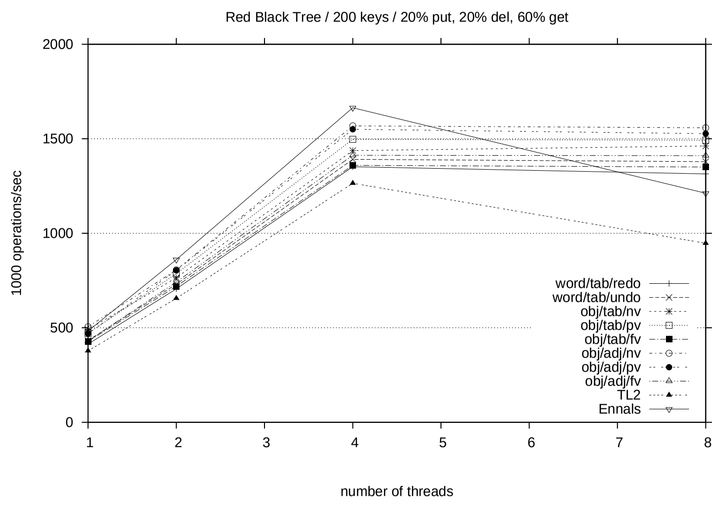
On this test we verified the performance of our prototype against one of the top performing STM engines to date. To prepare these tests we had to adapt our harness to be used by Ennals implementation, therefore these tests are significantly different from the ones used by Ennals on [Enn06] and TL2 on [DON06]. Ennals STM code had to do some hard coded checks in the engine to be able to use the original red black tree implementation from [HF03]. We decided to remove those hard coded checks since they were useless with our test harness.

This test results are shown in Figure 5.12. We can see Ennals STM having a close but higher performance than our prototype. The closest combination in terms of performance is the *object mode/lock adjacent to the data/no validation/undo log mode*. In fact this combination is the closest to the Ennals STM implementation—still, with this combination, some algorithmic differences subsist between Ennals STM and our prototype, namely Ennals uses a specialized malloc/free implementation whereas our prototype uses standard malloc/free; and Ennals uses a standard version write lock, whereas our prototype uses the global version clock algorithm (although it is not used for consistent state validation, it is required to mark deleted/free'd objects). We consider the performance difference to be acceptable (less than 5%), considering that our prototype does not need a special malloc/free implementation.

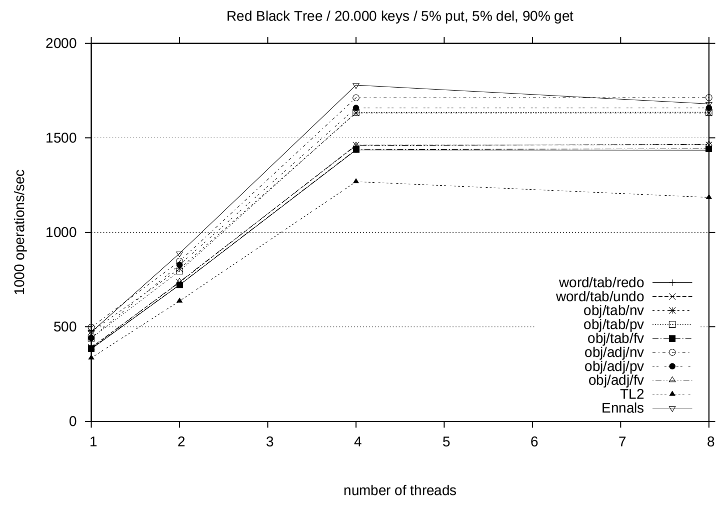
When running with more threads than CPUs, the performance of Ennals STM drops significantly and is overtaken by our prototype, which maintains the same performance level. This effect on Ennals STM may be due to the inexistence of a backoff mechanism. This mechanism, which is available in TL2, reduces contention by throttling down transactions when they try to access the same variable. If using the backoff mechanism, when a transaction tries to access a variable that is locked by another transaction, it rolls back the changes it made and then waits a certain period of time before retrying.



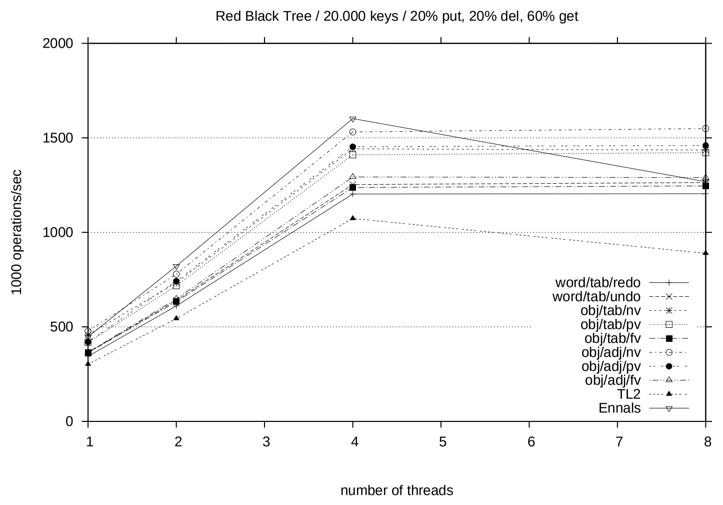
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

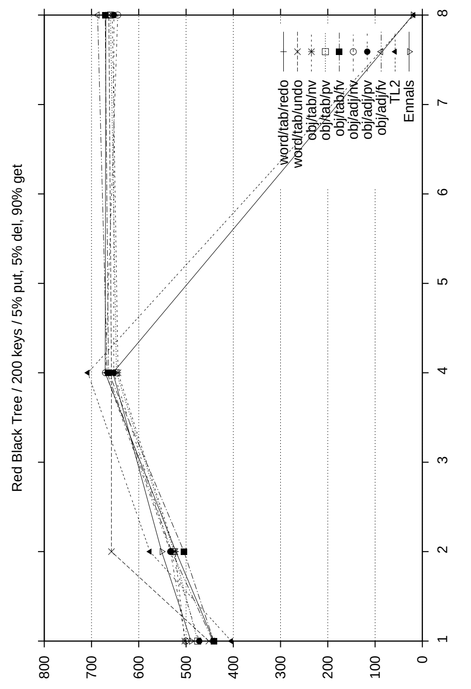
Figure 5.12: Ennals vs improved prototype

5.4.7 Cache coherency problems

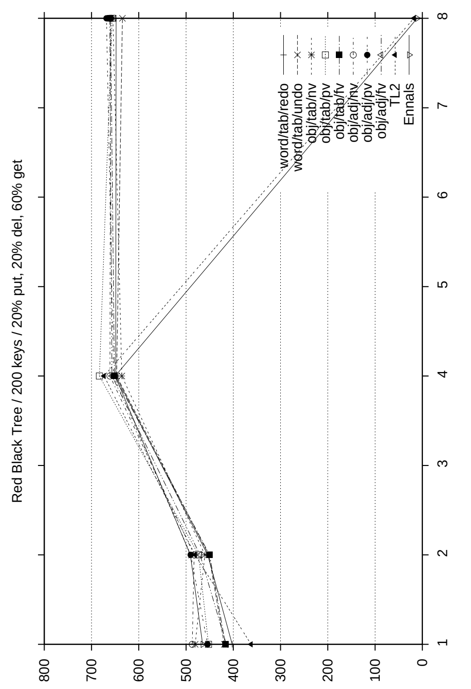
One important topic when trying to create a good performance multi-threaded program is the cache coherency traffic. Most common shared memory hardware architectures use a shared bus among all CPUs. The bus has several uses, two of which are: reading and writing data to the main memory; and for the CPUs to execute the cache coherency protocol. This protocol maintains a consistent view of the main memory among all memory caches in situations where two or more CPU are handling the same memory address.

If, for example, the value of a memory address is stored on the cache of two or more CPUs and one of them writes to that address, the CPU must inform the others that the value on their caches is no longer up-to-date (invalid) [HP96]. Therefore, when the other CPUs need to access the same address a cache miss will occur (even tough the address is in the cache, it is no longer up-to-date), causing extra traffic on the shared bus and delaying the execution.

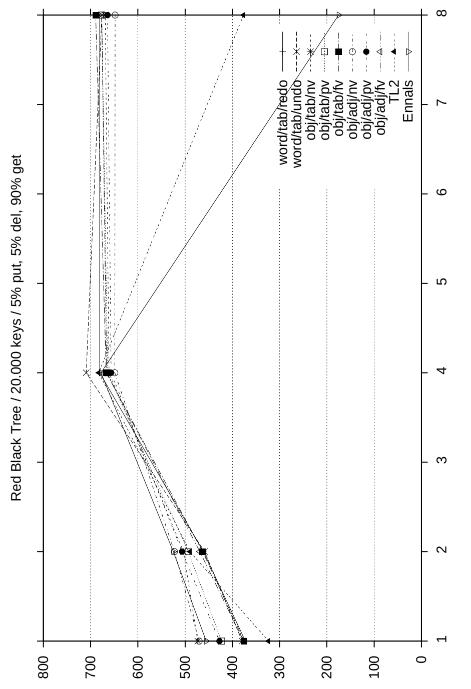
In this test we have created a shared read/write variable that is read and written for every iteration of the harness and it occupies the size of one cache line, the shared variable was the random number generator seed. The test is the same as the one on Section 5.4.6 and the results are shown on Figure 5.13.



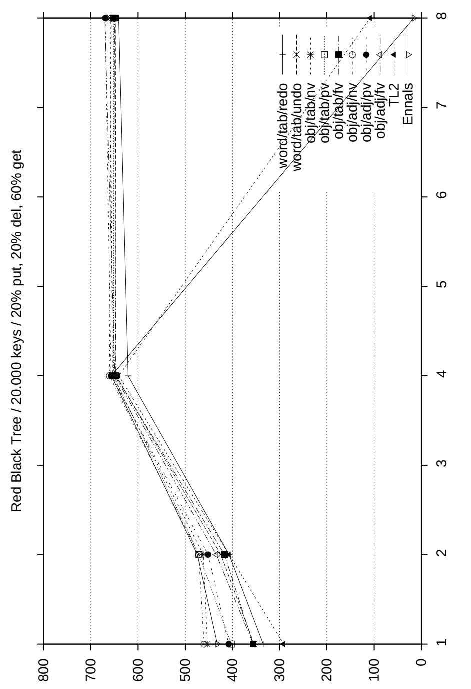
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

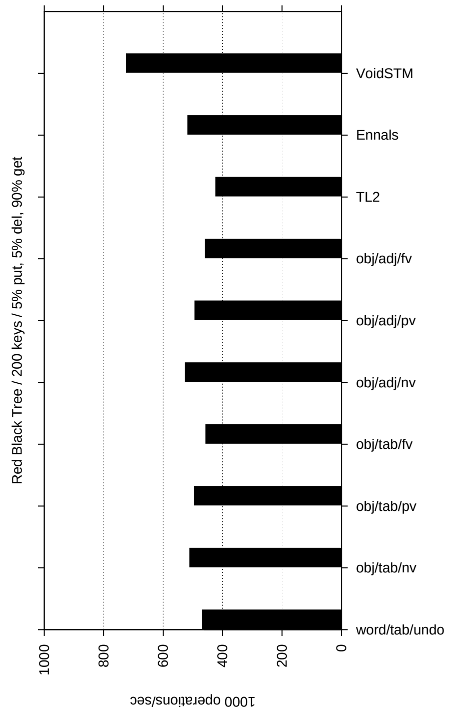
Figure 5.13: High cache coherency traffic on the bus

In this test, the performance increase with the number of threads is much lower (on all test combinations) than it was on previous tests. With eight threads, the performance of our prototype is maintained, but the performance of the ported TL2 and Ennals STM drops acutely. The difference is related to the improved *TxLoad* algorithm and to the backoff contention reduction algorithm. Ennals STM does not have a contention reduction algorithm—when a transaction aborts it immediately retries. TL2 used an algorithm which spined for an exponential amount of time on abort, whereas our prototype uses an algorithm similar to TL2 but it yields the CPU to other threads instead of doing a busy wait like TL2 does. Further study must be made to investigate the impact of the contention reduction algorithm.

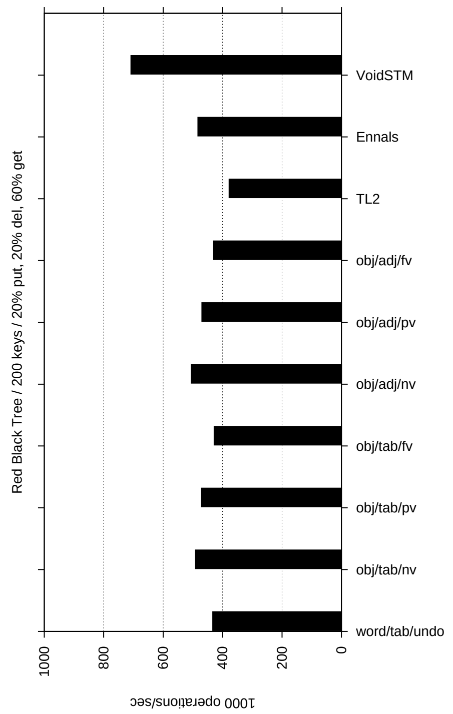
5.4.8 STM engine overhead

One way to evaluate the overhead of the STM engine is to evaluate its performance against a non synchronized version of the algorithm. In this test, all the transactional primitives were removed from the test harness, therefore there is no logging, locking, nor validation. Naturally, the non synchronized version can only run with one thread.

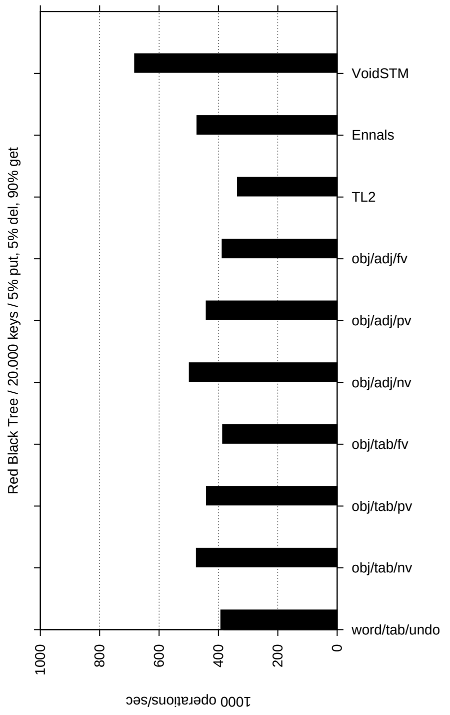
In Figure 5.14, it is shown the performance of all combinations and the non synchronized red black tree version, which is identified as *VoidSTM*. All tests where run with one thread only and as expected, the non synchronized version outperforms all others, where it achieves a performance that is between 1.5 and 3 times higher than the others. The performance of the non synchronized version is only beaten by the STM version running with 2 threads. There is still a big overhead and with a small number of CPUs and the purely sequential version is a respectable adversary.



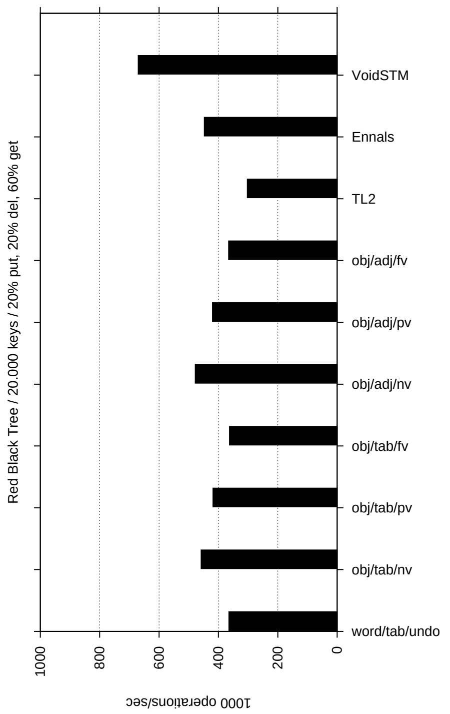
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get

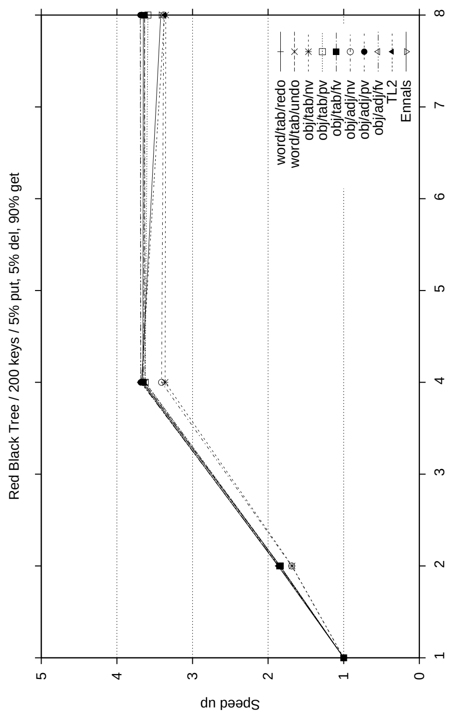


(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

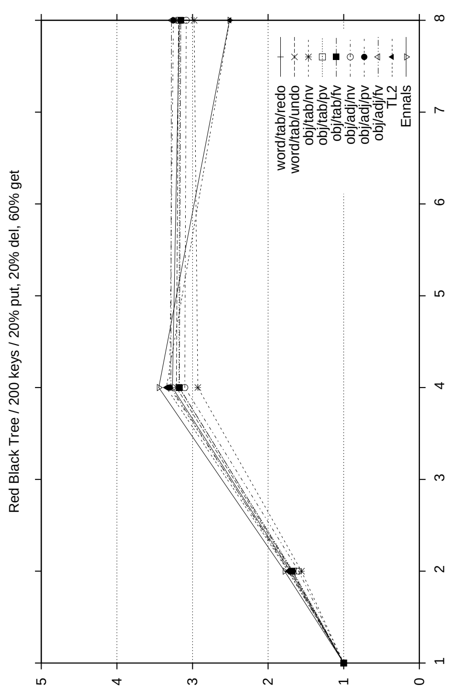
Figure 5.14: STM engine overhead

5.4.9 Speedup Analysis

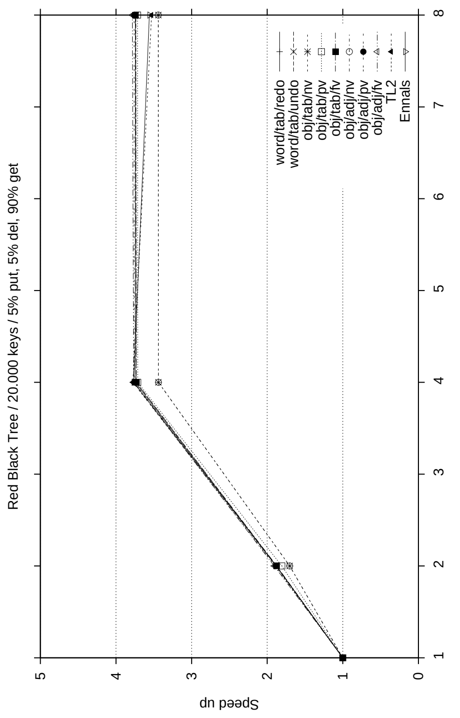
Another important measure of the prototype's performance is the speedup. Figure 5.15 shows that the speedup is almost linear on the read patterns but still very good on the write patterns. The combinations that don't do consistent state validation have the lowest speedup as they waist more time running in inconsistent states. Also the speedup is negative with more threads than CPUs for Ennals and the ported TL2.



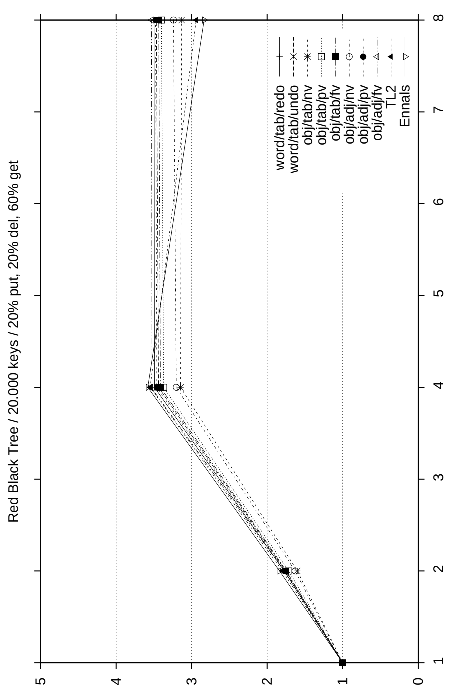
(a) Red Black Tree / 200 keys / 5%put 5%del 90%get



(b) Red Black Tree / 200 keys / 20%put 20%del 60%get



(c) Red Black Tree / 20.000 keys / 5%put 5%del 90%get



(d) Red Black Tree / 20.000 keys / 20%put 20%del 60%get

Figure 5.15: STM engine speedup

5.5 Conclusions

The above tests show a small but consistent advantage of undo-log strategy over redo-log. They also show that object mode has a better performance than word based mode although the advantage fades when contention is higher.

In terms of performance, consistent state validation has a prize and a penalty, the prize is the early detection of inconsistent states and therefore the early abort of the transaction; the penalty is related to the additional instructions involved in validation. The tests with *partial validation* show significantly better performance than *full validation* and in some cases it overtakes the *no validation* option.

The tests made with the adjacent lock show a small advantage over the tests with lock table.

Finally, the test with the show that all prototype combinations have a close to linear speedup.

Chapter 6

Conclusions

This Chapter summarizes the results of this investigation and brings out some pointers for future directions.

6.1 Conclusions

This work has presented several implementation options for STM engines. Those options were evaluated in terms of performance and safety, always with a focus on running transactions in consistent states.

We have based our work on the TL2 implementation described on [DON06]. We started by porting TL2 implementation from Solaris SPARC 64bit to Linux X86. In the X86 version we implemented a set of new features, performance enhancements and a new tracing engine.

The new added features were:

- We have implemented user called transaction aborts.
- We have included an automatic transaction retry mechanism which restarts the transaction when an inconsistent state is detected, changing the commit semantic from "at most one" to "exactly one".
- We have implemented transaction nesting on TL2 with partial rollbacks in undo and redo log schemes and with object and word based modes.

We have made experiments with different design options:

- We have improved the global version clock algorithm to increase its safety features. The original algorithm did not guarantee running in consistent states when transactional memory was released (free'd).
- We have adapted the TL2's global version clock algorithm to be used with the undo logging recovery strategy.
- We have adapted TL2 to work on object based mode. This is the first STM implementation, known to us, to do consistent state validation with undo logging strategy (in either object or word based mode). We have concluded that the object based mode with undo logging strategy may have significantly better performance than the others. The benchmarks show that on word based mode the undo and redo logging strategy yields similar results and that the undo log/object mode has the best performance at the cost of a more complex validation scheme.
- We have improved the global version clock algorithm in terms of performance, achieving a significantly lower overhead on the most used transactional method—the transactional load; and a lower cache coherency traffic on the shared bus. The test results show that it yields a better performance than the original algorithm,

both on regular load and on overload (more running transactions than CPUs)—with a close to zero performance degradation when there are more threads running than CPUs.

- We have studied the performance cost of validating the consistent state, showing that the impact can be high, specially on low contention. Therefore we propose a partial state validation scheme, which proves to be a good option in terms of performance and safety.
- We have studied the effect of lock placement: locks in a separate table versus locks stored adjacent to data and we have compared their performance. Our test results show a small performance improvement for the later over the former.
- We have implemented and made experiments with different word sizes to use the whole cache line instead of just one word. The results don't show any significant performance difference.

Other Contributions were:

- We have built a very lightweight tracing engine, which was indispensable to debug the STM implementation. We avoided using standard locks on the tracer synchronization because most concurrency problems would be hidden due to the very fine grained lock granularity of the STM engine. This tracing engine records events in order of occurrence within an exclusive section of a single CPU instruction. We have been able to reproduce all observed bugs using this engine.
- All these changes have been tested with two test harnesses: a red black tree and a sorted list implementation, which have been exposed to severe test conditions like: a list or tree with less nodes than transactions operating on it, causing a huge number of aborts; and originating a high number of lock collisions, by using very small lock tables (even with a single lock).
- In addition to the changes made on the prototype we have proposed a novel classification scheme for transaction states. We classified them as: *updated consistent* when a transaction has seen a fully updated memory snapshot; *obsolete consistent* when a transaction has seen a past memory snapshot; and *inconsistent* when a transaction has seen a dirty snapshot.
- We have shown in detail several hard to find concurrency bugs in the STM implementation and we have designed a few testing patterns which aided at finding and reproducing those bugs.

- Finally, we are also the first to present a type of hazard that may occur on existing lock based STMs that use the undo log strategy. This hazard may occur because transactional writes may take place when a transaction is in an invalid state and therefore the write may happen on a non-transactional variable.

6.2 Future Work

Still a lot of work can be done to improve STM engines in terms of features, performance and integration with other applications.

An interesting area is the integration of the STM engine with a debugger. A debugger could attach to a transaction and show the memory snapshot the transaction has by having a look at the transaction log. It could hide the transactional object headers from the user (unless requested) to reduce the debugging complexity. It could also abort a transaction by user demand.

Another interesting work would be to create a trace visualization tool that analyzes and displays the interleavings of the transactions. It could also replay the interleavings recorded on a specific run.

Regarding our prototype some interesting next steps are:

- Implementation and evaluation of a redo log/object based combination.
- Benchmarks with higher capacity machines, and with different (possibly non synthetic) test harnesses.
- Integration of the STM engine with a compiler, to avoid the overhead of the function call and enable further optimizations.
- Implementation of the *retry* and *orElse* primitives.
- Improving the transaction nesting functionality to have full support for closed nesting.

Appendix A

Raw test data

This Appendix shows the raw numbers of the executed tests.

A.1 Keywords

The following tables show the keywords used on the test results table.

Keyword	**Meaning**
cmd	STM engine options (see bellow)
duration	Test duration in 1/10 second
nthr	Number of threads
pput	Frequency of puts (%)
pdel	Frequency of deletes (%)
pget	Frequency of gets (%)
krange	Key Range
total	Number of operations performed
ld_aborts	Number of aborts detected while running *TxLoad*
vfy_aborts	Number of aborts detected while running *TxVerifyAddr*
st_aborts	Number of aborts detected while running *TxStore*
segf_aborts	Number of aborts detected due to dereferencing an invalid pointer
cmt_aborts	Number of aborts detected while running *TxCommit*
total_aborts	Total number of aborts
total_time	Total CPU time of the test run ($\approx$ test duration x number of threads)
stm_time	CPU time spent running the STM engine
harn_time	CPU time spent running the test harness

Figure A.1: Test results keywords

Keyword	**Meaning**
wtr	Prototype running with: word based mode; lock on separate table; and redo log mode
wtu	Prototype running with: word based mode; lock on separate table; and undo log mode
otn	Prototype running with: object based mode; lock on separate table; undo log mode; and no consistent state validation
otp	Prototype running with: object based mode; lock on separate table; undo log mode; and partial consistent state validation
otf	Prototype running with: object based mode; lock on separate table; undo log mode; and full consistent state validation
oan	Prototype running with: object based mode; lock adjacent to object; undo log mode; and no consistent state validation
oap	Prototype running with: object based mode; lock adjacent to object; undo log mode; and partial consistent state validation
oaf	Prototype running with: object based mode; lock adjacent to object; undo log mode; and full consistent state validation
TL2	Ported version of TL2
Ennals	Ennals STM
Void	Test harness running without synchronization primitives

Figure A.2: STM engine running modes

A.2 Raw Data of the Red Black Tree Tests

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
wtr	600	1	5	5	90	200	27770799	0	0	0	0	0	0	60005256	35395311	12421432
wtr	600	2	5	5	90	200	51180027	159466	0	0	0	4440	163906	120000383	74849376	22722799
wtr	600	4	5	5	90	200	101578160	869547	0	0	0	27564	897111	240008478	150263828	45283256
wtr	600	8	5	5	90	200	101367586	856565	0	0	0	26976	883541	479891088	390065200	45180438
wtr	600	1	20	20	60	200	24843451	0	0	0	0	0	0	60002760	38064489	11029156
wtr	600	2	20	20	60	200	42356240	481029	0	0	0	51238	532267	120001881	82563127	18830283
wtr	600	4	20	20	60	200	81143132	2712433	0	0	0	305448	3017881	239998080	168297488	36032943
wtr	600	8	20	20	60	200	78843529	4324297	0	0	0	702366	5026663	479982597	410051479	35238426
wtr	600	1	5	5	90	20000	22999185	0	0	0	0	0	0	60005909	39732384	10200177
wtr	600	2	5	5	90	20000	43303808	2063	0	0	0	72	2135	119995170	81807658	19202587
wtr	600	4	5	5	90	20000	86148801	23088	0	0	0	1171	24259	240002092	163870104	38319349
wtr	600	8	5	5	90	20000	86102931	19440	0	0	0	480	19920	479906605	375675211	52138840
wtr	600	1	20	20	60	20000	20665610	0	0	0	0	0	0	60001891	41641442	9237979
wtr	600	2	20	20	60	20000	36663977	6609	0	0	0	883	7492	120004604	87553786	16287041
wtr	600	4	20	20	60	20000	72179084	38883	0	0	0	4888	43771	239997728	175952775	32223392
wtr	600	8	20	20	60	20000	72286432	70978	0	0	0	10635	81613	479972789	406037670	36410904
wtu	600	1	5	5	90	200	28032554	0	0	0	0	0	0	60006032	35305746	12445230
wtu	600	2	5	5	90	200	51683985	176191	0	612	0	3489	180292	120004308	74447612	22939546
wtu	600	4	5	5	90	200	101710398	1045081	0	3940	0	21778	1070799	239989223	150431856	45094249
wtu	600	8	5	5	90	200	101848732	1056648	0	4130	0	22100	1082878	479932622	389746793	45561214
wtu	600	1	20	20	60	200	25977716	0	0	0	0	0	0	60001497	37101712	11534264
wtu	600	2	20	20	60	200	43842195	580042	0	7644	0	40153	627839	119999903	81357436	19453233
wtu	600	4	20	20	60	200	83459571	3320332	0	46283	0	257024	3623639	240002790	166475042	37029913
wtu	600	8	20	20	60	200	82737501	3836973	0	78217	0	350612	4265802	479990327	407021399	36763666
wtu	600	1	5	5	90	20000	23494051	0	0	0	0	0	0	60005215	39319004	10415566
wtu	600	2	5	5	90	20000	44121925	2272	0	6	0	50	2328	120001140	81136306	19568314
wtu	600	4	5	5	90	20000	87691388	13899	0	35	0	376	14310	240008467	162485068	39191180
wtu	600	8	5	5	90	20000	87778410	20569	0	43	0	475	21087	479885180	378254592	50266402
wtu	600	1	20	20	60	20000	21868235	0	0	0	0	0	0	60005497	40687016	9756332
wtu	600	2	20	20	60	20000	38428500	7700	0	56	0	683	8439	119999672	85944519	17249496
wtu	600	4	20	20	60	20000	75194827	89982	0	2245	0	12671	104898	239995054	173777145	33340963
wtu	600	8	20	20	60	20000	75828184	82714	0	3141	0	11792	97647	480311782	406482535	36918218

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
otn	600	1	5	5	90	200	30594083	0	0	0	0	0	0	60001329	32888883	13611637
otn	600	2	5	5	90	200	51488589	0	0	10942	0	254094	265036	120001897	74359931	22927010
otn	600	4	5	5	90	200	102860344	0	0	64416	25	1575091	1639532	240000615	148574738	45720116
otn	600	8	5	5	90	200	102627200	0	0	62764	10	1565464	1628238	479962248	388269108	45995894
otn	600	1	20	20	60	200	29448144	0	0	0	0	0	0	60005374	33877886	13102412
otn	600	2	20	20	60	200	45900067	0	0	143406	12	850558	993976	119999291	79088251	20639580
otn	600	4	20	20	60	200	86255751	2	0	1097069	217	6418389	7515677	240004343	163472823	38399616
otn	600	8	20	20	60	200	87708062	1	0	876467	213	5278339	6155020	479983305	401847316	39139504
otn	600	1	5	5	90	20000	28460472	0	0	0	0	0	0	60002796	34691724	12659866
otn	600	2	5	5	90	20000	48469284	0	0	1543	0	89000	90543	120001019	76913629	21550714
otn	600	4	5	5	90	20000	97891433	0	0	9235	0	524203	533438	240000150	152908352	43567632
otn	600	8	5	5	90	20000	97898044	0	0	9659	0	522824	532483	479993752	358332979	62580997
otn	600	1	20	20	60	20000	27485237	0	0	0	0	0	0	60001540	35513163	12296030
otn	600	2	20	20	60	20000	43996572	0	0	21029	0	363341	384370	120007518	80911070	19602067
otn	600	4	20	20	60	20000	86495636	0	0	125806	1	2144293	2270100	239994872	163162440	38517440
otn	600	8	20	20	60	20000	86114382	0	0	125903	2	2131478	2257383	479935797	395339281	41888548
otp	600	1	5	5	90	200	29647554	0	0	0	0	0	0	60005483	33879956	13159120
otp	600	2	5	5	90	200	54524261	0	280894	3472	0	12155	296521	119995524	71924680	24207249
otp	600	4	5	5	90	200	107502028	0	2155823	28585	0	85651	2270059	239970272	145266367	47686291
otp	600	8	5	5	90	200	106448876	0	1645253	21671	0	72069	1738993	479999967	385573168	47566570
otp	600	1	20	20	60	200	28250796	0	0	0	0	0	0	60004269	35096021	12547599
otp	600	2	20	20	60	200	47191176	0	899642	43699	0	152163	1095504	119999873	78363202	20969229
otp	600	4	20	20	60	200	89821587	0	5478588	282503	0	901475	6662566	239998454	160800615	39847400
otp	600	8	20	20	60	200	89545709	0	5436856	282495	0	908046	6627397	479950282	400898458	39778411
otp	600	1	5	5	90	20000	26413730	0	0	0	0	0	0	60000262	36723138	11723697
otp	600	2	5	5	90	20000	47717158	0	57357	1314	0	9049	67720	120008115	77778641	21304716
otp	600	4	5	5	90	20000	98071455	0	348993	7942	0	56185	413120	239987750	153584176	43499070
otp	600	8	5	5	90	20000	98128871	0	355178	8041	0	56085	419304	479878854	373436710	52434200
otp	600	1	20	20	60	20000	25094758	0	0	0	0	0	0	60006179	37874400	11135828
otp	600	2	20	20	60	20000	43066005	0	186565	17203	0	122131	325899	120012672	81965389	19139621
otp	600	4	20	20	60	20000	84610761	0	1160090	100240	0	690561	1950891	239995128	165360149	37567538
otp	600	8	20	20	60	20000	85310195	0	1177235	99832	0	696301	1973368	479850719	399312123	40350950

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
otf	600	1	5	5	90	200	27360720	0	0	0	0	0	0	60001701	35807311	12172364
otf	600	2	5	5	90	200	50451830	0	333827	3542	0	8718	346087	120000437	75356827	22449211
otf	600	4	5	5	90	200	99758846	0	1995046	22900	0	53510	2071456	239999002	151815074	44347026
otf	600	8	5	5	90	200	99453191	0	1926290	22223	0	52974	2001487	480000348	391775796	44385616
otf	600	1	20	20	60	200	25694091	0	0	0	0	0	0	60003004	37142505	11493192
otf	600	2	20	20	60	200	43178644	0	1098179	43905	0	109455	1251539	120000556	81653918	19235689
otf	600	4	20	20	60	200	81555248	0	6540437	285189	0	652087	7477713	240000333	167454956	36477379
otf	600	8	20	20	60	200	81048273	0	6366333	280675	0	651726	7298734	479987259	408025538	36078283
otf	600	1	5	5	90	20000	23134932	0	0	0	0	0	0	60002731	39561090	10279719
otf	600	2	5	5	90	20000	43353790	0	59668	1416	0	5631	66715	119999993	81653515	19287147
otf	600	4	5	5	90	20000	86283134	0	361731	8593	0	32735	403059	239995820	163739569	38346744
otf	600	8	5	5	90	20000	86570810	0	373579	9314	0	33184	416077	479912448	389287534	45614813
otf	600	1	20	20	60	20000	21756906	0	0	0	0	0	0	60002035	40711850	9671624
otf	600	2	20	20	60	20000	38016082	0	228681	18569	0	72903	320153	120007608	86296041	16916453
otf	600	4	20	20	60	20000	74251379	0	1419052	110646	0	416519	1946217	239988670	174165160	33006592
otf	600	8	20	20	60	20000	74742195	0	1375190	109155	0	414687	1899032	479942476	410324976	35063876
oan	600	1	5	5	90	200	31527072	0	0	0	0	0	0	60002427	32214419	13997669
oan	600	2	5	5	90	200	53243352	0	0	8974	0	222955	231929	119995821	73040602	23653721
oan	600	4	5	5	90	200	107480395	0	0	53658	11	1372472	1426141	240008164	145279607	47708370
oan	600	8	5	5	90	200	106551617	0	0	52936	7	1347975	1400918	479990227	384549675	48195213
oan	600	1	20	20	60	200	30330816	0	0	0	0	0	0	60003087	33270584	13459358
oan	600	2	20	20	60	200	48397736	0	0	123178	5	756618	879801	120002327	77320838	21486923
oan	600	4	20	20	60	200	94093308	0	0	760665	157	4809643	5570465	240001081	156841271	41986147
oan	600	8	20	20	60	200	93506448	0	0	743093	176	4751042	5494311	480035277	397442010	41598755
oan	600	1	5	5	90	20000	29875607	0	0	0	0	0	0	60004990	33623095	13352052
oan	600	2	5	5	90	20000	50864486	0	0	1477	0	87037	88514	119993488	75164028	22573315
oan	600	4	5	5	90	20000	102722985	0	0	8365	0	481990	490355	240018013	149571657	45588764
oan	600	8	5	5	90	20000	102796128	0	0	8501	0	490920	499421	479870277	341989627	71787419
oan	600	1	20	20	60	20000	28677186	0	0	0	0	0	0	60001584	34727166	12718123
oan	600	2	20	20	60	20000	46737624	0	0	20194	0	344395	364589	120001830	78780651	20760581
oan	600	4	20	20	60	20000	91880601	0	0	113180	0	1984196	2097376	240009110	159039993	40786886
oan	600	8	20	20	60	20000	92950550	0	0	116117	0	1994940	2111057	479619231	384773793	47519616

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
oap	600	1	5	5	90	200	29586622	0	0	0	0	0	0	60001372	33884369	13134939
oap	600	2	5	5	90	200	54899786	0	224832	2951	0	10361	238144	120001553	71353074	24574274
oap	600	4	5	5	90	200	109032504	0	1298338	18824	0	61012	1378174	240006016	143775840	48469700
oap	600	8	5	5	90	200	109092444	0	1321900	18867	0	62325	1403092	479910490	382403230	49362434
oap	600	1	20	20	60	200	28164182	0	0	0	0	0	0	60002467	35056823	12586029
oap	600	2	20	20	60	200	48226210	0	747967	38994	0	135545	922506	119995241	77179505	21672166
oap	600	4	20	20	60	200	93029877	0	4567540	258557	0	801381	5627478	240007501	157896213	41354869
oap	600	8	20	20	60	200	91619941	0	4743343	253528	0	813452	5810323	480014925	399022760	40806897
oap	600	1	5	5	90	20000	26464313	0	0	0	0	0	0	60000194	36631762	11777543
oap	600	2	5	5	90	20000	49717696	0	50402	1146	0	8389	59937	120007914	75948260	22250548
oap	600	4	5	5	90	20000	99516265	0	283135	7191	0	48952	339278	240022596	152184224	44239784
oap	600	8	5	5	90	20000	99500357	0	290860	6964	0	49218	347042	479898971	358759177	60099974
oap	600	1	20	20	60	20000	25192743	0	0	0	0	0	0	60001746	37772095	11195341
oap	600	2	20	20	60	20000	44522668	0	167818	15923	0	114924	298665	119999098	80680699	19804006
oap	600	4	20	20	60	20000	87192742	0	1008067	91573	0	627140	1726780	240033004	162951564	38869850
oap	600	8	20	20	60	20000	87622780	0	997565	93874	0	637860	1729299	479763543	389636223	44156038
oaf	600	1	5	5	90	200	27510585	0	0	0	0	0	0	60005917	35713344	12218835
oaf	600	2	5	5	90	200	51199718	0	278832	3252	0	7930	290014	119998860	74775102	22753206
oaf	600	4	5	5	90	200	101375701	0	1634857	20245	0	47082	1702184	239986892	150529689	44978428
oaf	600	8	5	5	90	200	101550307	0	1652774	20221	0	47136	1720131	479980194	389962935	45310473
oaf	600	1	20	20	60	200	25809312	0	0	0	0	0	0	60003994	37044237	11542933
oaf	600	2	20	20	60	200	44548544	0	967341	40994	0	101706	1110041	119999212	80484476	19812047
oaf	600	4	20	20	60	200	84779472	0	5749469	264566	0	598396	6612431	239997378	164821871	37681929
oaf	600	8	20	20	60	200	84638035	0	5741557	262842	0	598606	6603005	479996201	404877236	37643197
oaf	600	1	5	5	90	20000	23242525	0	0	0	0	0	0	60004003	39488492	10319107
oaf	600	2	5	5	90	20000	44316350	0	52644	1386	0	5381	59411	120008541	80866873	19690568
oaf	600	4	5	5	90	20000	87551921	0	303010	7713	0	29812	340535	239998067	162485126	39100866
oaf	600	8	5	5	90	20000	87948449	0	311170	7770	0	30089	349029	479858942	382666537	48710828
oaf	600	1	20	20	60	20000	21927175	0	0	0	0	0	0	60001981	40567015	9740824
oaf	600	2	20	20	60	20000	38915085	0	212462	17371	0	68761	298594	119999024	85414269	17392838
oaf	600	4	20	20	60	20000	77582700	0	1235708	104794	0	390514	1731016	239993659	171041807	34680403
oaf	600	8	20	20	60	20000	77403976	0	1239388	103498	0	389152	1732038	480012261	405691697	36780245

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
TL2	600	1	5	5	90	200	25359092	0	0	0	0	0	0	60003603	37716995	11213285
TL2	600	2	5	5	90	200	47164465	0	0	0	0	0	0	120003086	78550168	20877901
TL2	600	4	5	5	90	200	93194306	0	0	0	0	0	0	239989569	157837859	41518250
TL2	600	8	5	5	90	200	85463351	0	0	0	0	0	0	479932512	331949552	71641684
TL2	600	1	20	20	60	200	22671725	0	0	0	0	0	0	60000280	40096105	10031603
TL2	600	2	20	20	60	200	39342457	0	0	0	0	0	0	120008486	85399794	17417307
TL2	600	4	20	20	60	200	75872161	0	0	0	0	0	0	240000774	173183315	33614453
TL2	600	8	20	20	60	200	56848863	0	0	0	0	0	0	479882753	375897236	50907048
TL2	600	1	5	5	90	20000	20147984	0	0	0	0	0	0	60004266	42288251	8918220
TL2	600	2	5	5	90	20000	38193302	0	0	0	0	0	0	120007634	86417684	16941922
TL2	600	4	5	5	90	20000	76088290	0	0	0	0	0	0	239984289	173175517	33654747
TL2	600	8	5	5	90	20000	71099625	0	0	0	0	0	0	479832114	354715320	63226053
TL2	600	1	20	20	60	20000	18141997	0	0	0	0	0	0	60006470	44055408	8022363
TL2	600	2	20	20	60	20000	32640660	0	0	0	0	0	0	120005334	91305214	14458453
TL2	600	4	20	20	60	20000	64396603	0	0	0	0	0	0	240015277	183236623	28711959
TL2	600	8	20	20	60	20000	53347455	0	0	0	0	0	0	480620204	388924968	48453630
Ennals	600	1	5	5	90	200	31008235	0	0	0	0	0	0	60003500	32586228	13827009
Ennals	600	2	5	5	90	200	57948473	0	0	0	0	0	0	120004877	68670332	25933676
Ennals	600	4	5	5	90	200	113866795	0	0	0	0	0	0	239986541	139017285	51015516
Ennals	600	8	5	5	90	200	106058556	0	0	0	0	0	0	479907810	293451603	95416648
Ennals	600	1	20	20	60	200	28975153	0	0	0	0	0	0	60002822	34282863	13000602
Ennals	600	2	20	20	60	200	51636438	0	0	0	0	0	0	119999027	74269141	23068011
Ennals	600	4	20	20	60	200	99847006	0	0	0	0	0	0	239987979	151284522	44789478
Ennals	600	8	20	20	60	200	72726881	0	0	0	0	0	0	479516492	353760727	64103439
Ennals	600	1	5	5	90	20000	28337332	0	0	0	0	0	0	60006615	34953922	12643192
Ennals	600	2	5	5	90	20000	53319991	0	0	0	0	0	0	120010041	72827197	23824109
Ennals	600	4	5	5	90	20000	106750426	0	0	0	0	0	0	239999475	145564861	47678344
Ennals	600	8	5	5	90	20000	100814261	0	0	0	0	0	0	480031716	305503277	88296304
Ennals	600	1	20	20	60	20000	26843927	0	0	0	0	0	0	60003685	36257892	11983916
Ennals	600	2	20	20	60	20000	49312823	0	0	0	0	0	0	119999804	76352967	22013803
Ennals	600	4	20	20	60	20000	96110399	0	0	0	0	0	0	240002013	154839024	42972613
Ennals	600	8	20	20	60	20000	76087596	0	0	0	0	0	0	479910403	346962989	70685694

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
Void	600	1	5	5	90	200	43378565	0	0	0	0	0	0	60000996	21742446	19289642
Void	600	1	20	20	60	200	0	0	0	0	0	0	0	60000682	22524814	18887228
Void	600	1	5	5	90	20000	0	0	0	0	0	0	0	60001358	23865634	18222003
Void	600	1	20	20	60	20000	40202564	0	0	0	0	0	0	60000558	24529991	17880562

A.3 Raw Data of the Sorted List Tests

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
wtr	600	1	5	5	90	200	13810886	0	0	0	0	0	0	60002659	47782939	6166247
wtr	600	2	5	5	90	200	25748860	618161	0	0	0	64383	682544	120007867	97194426	11446165
wtr	600	4	5	5	90	200	49031435	3930508	0	0	0	411619	4342127	239993210	196400069	21795163
wtr	600	8	5	5	90	200	49904779	3519957	0	0	0	420109	3940066	479914367	435389791	22363957
wtr	600	1	20	20	60	200	12283737	0	0	0	0	0	0	60005026	49177935	5449541
wtr	600	2	20	20	60	200	19874149	1922199	0	0	0	864580	2786779	120006199	102264814	8849850
wtr	600	4	20	20	60	200	28531294	8841395	0	0	0	4202789	13044184	240001150	214463013	12718022
wtr	600	8	20	20	60	200	27815591	8340645	0	0	0	3922095	12262740	480006065	455109559	12396882
wtu	600	1	5	5	90	200	14294117	0	0	0	0	0	0	60001881	47393485	6347299
wtu	600	2	5	5	90	200	27728651	584654	0	65	0	58854	643573	120007465	95466102	12329539
wtu	600	4	5	5	90	200	52872657	3478558	0	506	0	355893	3834957	239999100	193058352	23540465
wtu	600	8	5	5	90	200	52769253	3466576	0	535	0	357943	3825054	479957643	432708421	23642308
wtu	600	1	20	20	60	200	13032169	0	0	0	0	0	0	60005204	48500118	5785051
wtu	600	2	20	20	60	200	21542610	1954409	0	1018	0	778338	2733765	119999651	100762719	9606922
wtu	600	4	20	20	60	200	29022253	11446449	0	25230	0	3232825	14704504	240002354	213981072	12957682
wtu	600	8	20	20	60	200	29286073	10528199	0	17185	0	3060749	13606133	479914829	453616076	13111325
otn	600	1	5	5	90	200	26670253	0	0	0	0	0	0	60000590	36491591	11846630
otn	600	2	5	5	90	200	45024273	0	0	73946	6	829510	903462	120004951	79987994	20154992
otn	600	4	5	5	90	200	85308910	0	0	485064	38	5281140	5766242	239996683	164339103	37909561
otn	600	8	5	5	90	200	86533271	0	0	486605	13	5277908	5764526	479964710	402095930	39111010
otn	600	1	20	20	60	200	25573609	0	0	0	0	0	0	60005855	37410923	11414448
otn	600	2	20	20	60	200	36763707	0	0	1050246	21	2436827	3487094	120009825	86921275	16547934
otn	600	4	20	20	60	200	64086868	0	0	6857094	308	13535579	20392981	239998409	182359703	28788142
otn	600	8	20	20	60	200	63736235	0	0	6879211	298	13473371	20352880	480025632	422633643	28619099
otp	600	1	5	5	90	200	18071565	0	0	0	0	0	0	60001324	44079895	8019887
otp	600	2	5	5	90	200	33727703	0	650557	572	0	42053	693182	120001786	90108928	15039930
otp	600	4	5	5	90	200	65469173	0	3916718	3978	0	264843	4185539	240001078	181762335	29281185
otp	600	8	5	5	90	200	64944439	0	3885725	3943	0	261943	4151611	479968211	420688968	29709064
otp	600	1	20	20	60	200	17529044	0	0	0	0	0	0	60000628	44556384	7777062
otp	600	2	20	20	60	200	28869976	0	2250311	8649	0	657323	2916283	120003135	94218440	12893523
otp	600	4	20	20	60	200	46443691	0	13456278	65637	0	3455905	16977820	240018298	198236930	20813163
otp	600	8	20	20	60	200	45974182	0	13081137	60524	0	3385059	16526720	480011850	438644298	20593690

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
otf	600	1	5	5	90	200	13629838	0	0	0	0	0	0	60003774	47970585	6050798
otf	600	2	5	5	90	200	25338497	0	543550	464	0	45334	589348	120007327	97549041	11266776
otf	600	4	5	5	90	200	48722470	0	3290272	3048	0	270087	3563407	239988245	196734633	21663904
otf	600	8	5	5	90	200	49105780	0	3318712	3140	0	271500	3593352	479944073	435925724	22027238
otf	600	1	20	20	60	200	12763247	0	0	0	0	0	0	60003918	48681836	5664183
otf	600	2	20	20	60	200	20915274	0	1915708	6608	0	644994	2567310	119996330	101243347	9315915
otf	600	4	20	20	60	200	31118194	0	10528717	37741	0	2797421	13363879	240000388	211938964	13900120
otf	600	8	20	20	60	200	30686122	0	10208943	36052	0	2735707	12980702	480004796	452347886	13708508
oan	600	1	5	5	90	200	26715884	0	0	0	0	0	0	60002477	36393865	11922833
oan	600	2	5	5	90	200	45886476	0	0	63911	727	723321	787959	120008955	79394426	20418075
oan	600	4	5	5	90	200	88372153	0	0	435566	3410	4749083	5188059	240015152	161377348	39611163
oan	600	8	5	5	90	200	88377971	0	0	428698	3341	4719020	5151059	479947158	399039849	40850613
oan	600	1	20	20	60	200	25751366	0	0	0	0	0	0	60002170	37310035	11430759
oan	600	2	20	20	60	200	39940825	0	0	844776	1879	1970141	2816796	119999471	84334952	17852313
oan	600	4	20	20	60	200	67821181	0	0	6323543	11447	12838303	19173293	239998093	178864428	30584331
oan	600	8	20	20	60	200	66146450	0	0	6180378	12102	12534310	18726790	479973935	420380263	29740826
oap	600	1	5	5	90	200	18033475	0	0	0	0	0	0	59999939	44009050	8041850
oap	600	2	5	5	90	200	33897622	0	595006	618	0	39367	634991	120006567	89816335	15156705
oap	600	4	5	5	90	200	63666720	0	3548704	3850	0	241440	3793994	239991553	183103416	28509333
oap	600	8	5	5	90	200	65397090	0	3598651	3869	0	246737	3849257	479943422	420020545	30181002
oap	600	1	20	20	60	200	17455461	0	0	0	0	0	0	60003308	44447899	7786350
oap	600	2	20	20	60	200	29509231	0	2124585	8595	0	634096	2767276	120007970	93358943	13248519
oap	600	4	20	20	60	200	48074349	0	12985913	65222	0	3381326	16432461	239979812	196102554	21806558
oap	600	8	20	20	60	200	47185168	0	12833747	64013	0	3374974	16272734	479999058	437069839	21266684
oaf	600	1	5	5	90	200	13768663	0	0	0	0	0	0	60006251	47869276	6110934
oaf	600	2	5	5	90	200	25933939	0	525382	524	0	43813	569719	120004503	97063159	11527263
oaf	600	4	5	5	90	200	49767630	0	3183349	3290	0	268937	3455576	239996176	195876114	22137021
oaf	600	8	5	5	90	200	49969244	0	3221172	3223	0	266908	3491303	479817569	434954288	22473860
oaf	600	1	20	20	60	200	12787570	0	0	0	0	0	0	60001562	48744689	5660192
oaf	600	2	20	20	60	200	21609673	0	1889559	6962	0	639355	2535876	120000540	100749215	9620671
oaf	600	4	20	20	60	200	32617433	0	10603270	40354	0	2854341	13497965	240002947	210721523	14606071
oaf	600	8	20	20	60	200	32219843	0	10233040	38979	0	2786966	13058985	479952200	450998171	14463122

cmd	duration	nthr	pput	pdel	pget	krange	total	ld_aborts	vfy_aborts	st_aborts	segf_aborts	cmt_aborts	total_aborts	total_time	stm_time	harn_time
Void	600	1	5	5	90	200	39888311	0	0	0	0	0	0	60002543	24757846	17802303
Void	600	1	20	20	60	200	39274849	0	0	0	0	0	0	60004842	25315241	17471426

Appendix B

STM Engine API

This Appendix shows the API of the prototype.

B.1 STM engine data structures

Thread	Contains the thread and transaction state information.

B.2 API for transaction initiation and control

Thread * *TxNewThread* ();
Creates, initializes and returns the **Thread** data structure. Must be called by every thread before starting any transaction.

void *TxStart* (**Thread** * const `self`, **int** `roflag`);
Starts a transaction. If `roflag` is 0 the transaction is read-only and it will have a smaller overhead. Otherwise the transaction is read-write.

int *TxValid* (**Thread** * const `self`);
Scans the read set and returns 1 if the read-set is in updated consistent state; returns 0 otherwise.

int *TxValidateAndAbort* (**Thread** * const `self`);
Scans the read set and returns 1 when the read set is updated consistent; aborts and retries the transaction otherwise.

int *TxCommit* (**Thread** * const `self`) ;
Commits the transaction. Always return 1. If the transaction cannot commit, it is restarted.

void *TxAbort* (**Thread** * const `self`, **int** `retry`) ;
Aborts the transaction. If `retry` is set to 1 the transactions is retried.

void *TxSterilize* (**Thread** * const `self`, **void** volatile * `base`, **size_t** const `length`);
Quiesces the variable with address `base` and size `length`. The size is in number of memory words. This function cannot be used inside a transaction.

B.3 API for loading and storing data in word based mode

intptr_t *TxLoad* (**Thread** * const `self`, **intptr_t** volatile * `addr`);
Returns the value of the transactional variable with address `addr`. The transaction aborts and retries if the variable has changed since the beginning of this transaction.

void *TxStore* (**Thread** * const `self`, **intptr_t** volatile * `addr`, **intptr_t** `value`);
Stores the value `value` on the transactional variable with address `addr`. The transaction aborts and retries if the variable has changed since the beginning of this transaction.

B.4 API for loading and storing data in object based mode

int *TxOpenRead* (**Thread** * const `self`, **void** volatile * `addr`);
Opens the transactional object/structure with address `addr` for read. Transactions must call this function before reading its contents. The transaction aborts and retries if the object/structure has changed since the beginning of this transaction.

int *TxOpenWrite* (**Thread** * const `self`, **void** volatile * `addr`, **unsigned int** `size`);
Opens the transactional object/structure with address `addr` and size `size` for write. Transactions must call this function before writing. The transaction aborts and retries if the object/structure has changed since the beginning of this transaction.

int *TxVerifyAddr* (**Thread** * const `self`, **void** volatile * `addr`);
Verifies if the object/structure with address `addr` has changed by another transaction since the beginning of this transaction and aborts/retries if that was the case. Otherwise it returns 1.

[This page was intentionally left blank]

Bibliography

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

[ATLM^{+}06] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM Press.

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

[Dat94] C. J. Date. *Introduction to Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1994.

[DON06] D. Dice, Shalev O., and Shavit N. Transactional Locking II. In *Proc. of the 20th International Symposium on Distributed Computing (DISC 2006)*, pages 194–208, Stockholm, Sweden, 2006.

[DS06] David Dice and Nir Shavit. What really makes transactions faster? In *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Jun 2006.

[DS07] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *Proc. of the 2007 International Symposium on Code Generation and Optimization*, 2007.

[Enn06] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report, Jan 2006.

[HF] T. Harris and K. Fraser. Concurrent programming without locks. http://research.microsoft.com/~tharris/drafts/cpwl-submission.pdf.

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

[HLM03] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 522, Washington, DC, USA, 2003. IEEE Computer Society.

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

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

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

[HP96] John L. Hennessy and David A. Patterson. *Computer architecture (2nd ed.): a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[HPST06] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 14–25, New York, NY, USA, 2006. ACM Press.

[int] *Intel 64 and IA-32 Architectures Software Developer's Manual*. http://www.intel.com/products/processor/manuals/index.htm.

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

[McK05] Paul E. McKenney. Memory ordering in modern microprocessors, Part I. *Linux J.*, 2005(136):2, 2005.

[MCS91] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[NSS] Dave Dice Nir Shavit and Ori Shalev. Transactional Locking II—Slides from TRANSACT06.

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

[sch] *Standard C Library Functions - schedctl_init Solaris man page.*

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

[Sut05] Herb Sutter. The free lunch is over. *Dr. Dobb's Journal*, 03 2005.

[WG] D. Weaver and T. Germond. *The SPARC Architecture Manual, Version 9.* PTR Prentice Hall, Englewood Cliffs, New Jersey 07632.

[WNSS05] III William N. Scherer and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM Press.