Transactional Java Futures

José Carlos Marante Pereira

Thesis to obtain the Master of Science Degree in
Information Systems and Computer Engineering

Supervisors: Prof. João Pedro Faria Mendonça Barreto
Prof. Paolo Romano

Examination Committee
Chairperson: Prof. Pedro Manuel Moreira Vaz Antunes de Sousa
Supervisor: Prof. João Pedro Faria Mendonça Barreto
Member of the Committee: Prof. Hérve Miguel Cordeiro Paulino

November 2014
Agradecimentos

There are many people to thank for the development of this work. I first would like to thank my coordinators, Professor João Barreto and Professor Paolo Romano. Their knowledge, experience and support have always provided me with the right guidance. Without them, i would not be able to develop this work, and its quality is greatly due to them.

I would like to thank everyone else at the SD Group at INESC-ID, specially Nuno Diegues and Ricardo Felipe, who were always available to help me out clarify any doubts during the development of this work.

I would also like to thank Ivo Anjos, for providing me his JVM implementation with support for first-class continuations. His system plays a very important role in the technical work of this dissertation.

Last but not least, i would like to thank my family and friends for all the strength and support they gave me in times of need. Two people which deserve to be highlighted are my mother and Tiago Rafael. A special thanks to them, for being everyday right by my side.

Lisboa, November 2014
José Carlos Marante Pereira
Resumo

Devido à sua importância na tecnologia actual, a programação paralela tem sido alvo de intensa investigação e desenvolvimento nos últimos anos com o objectivo de simplificar a programação de programas altamente paralelos. Memória Transacional em Software e Futures são dois exemplos proeminentes que resultaram de tal investigação. Ao providenciar abstracções importantes sobre aspectos complexos de concorrência, estes modelos permitem aos programadores construírem os seus programas paralelos com maior simplicidade que aquela que é fornecida por outros modelos de programação paralela. Contudo, mesmo estes dois exemplos estão longe de ser uma panaceia para a programação paralela. Pois ambos demonstram limitações cruciais que limitam as suas capacidades de extrair altos níveis de paralelismo das aplicações. Esta dissertação propõe um sistema unificado que suporta ambos os modelos, STM e Futures. Nesta dissertação mostramos que a nossa solução preserva as abstrações providenciadas por ambos e obtém uma maior eficácia em extrair paralelismo do que sistemas que se concentram em explorar cada um dos modelos individualmente.
Abstract

Because of its importance in nowadays technology, parallel programming has been subject of numerous efforts over the years to ease the task of building highly parallel programs. Software transactional memory and Futures are two prominent examples that arise from such efforts. By providing important abstractions over complex concurrency issues, they allow programmers to build parallel programs easier than other parallel programming models. However, they are not a panacea for parallel programming, as they often demonstrate crucial limitations that hinder one’s ability to extract higher levels of parallelism from applications. This dissertation proposes an unified system that supports the combination of both models, STM and Futures. In this dissertation we show that our solution preserves the abstractions provided by both systems, and achieves better effectiveness of extracting parallelism than systems that focus on exploring each one individually.
Palavras Chave

Keywords

Palavras Chave

Memória Transaccional em Software
Speculação de Threads
Java Futures
Java Software Transactional Memory

Keywords

Software Transactional Memory
Thread-Level Speculation
Java Futures
Java Software Transactional Memory
# Indice

## 1 Introducción

1.1 Context ........................................... 3
1.2 Motivación ........................................ 4
   1.2.1 Java Futures .................................. 4
   1.2.2 Software Transactional Memory .................. 6
   1.2.3 Software Transactional Memory + Futures ....... 7

## 2 Trabajo Relacionado

2.1 Transactional Memory .............................. 11
   2.1.1 Correctness criteria ........................... 12
   2.1.2 Weak and Strong Isolation ..................... 13
   2.1.3 Optimistic versus pessimistic concurrency control ..... 13
   2.1.4 Version management ............................ 14
   2.1.5 Conflict Detection ............................. 14
   2.1.6 Nesting ........................................ 15
   2.1.7 Progress Guarantees ............................ 16
   2.1.8 Providing Opacity .............................. 17
   2.1.9 STM vs HTM .................................... 18
   2.1.10 Programming Model ............................ 18
   2.1.11 JVSTM ........................................ 19
      2.1.11.1 API ........................................ 19
      2.1.11.2 Nesting transactions ...................... 21
2.1.11.3 Versioning and Conflict Detection .................................. 22
2.1.12 Other Java STM Examples .................................................. 24
2.1.13 Conclusion ................................................................. 24

2.2 Thread-Level Speculation ...................................................... 25
  2.2.1 Parallelizing Compilers .................................................. 25
  2.2.2 Thread-Level Speculation ................................................ 26
  2.2.3 Hardware Thread-Level Speculation ................................... 28
  2.2.4 Software Thread-Level Speculation ................................... 28
    2.2.4.1 S-TLS .............................................................. 28
    2.2.4.2 SableSpMT ....................................................... 29
    2.2.4.3 JaSPEx .......................................................... 30
  2.2.5 Hardware vs Software .................................................. 30
  2.2.6 Java Continuation ....................................................... 30
  2.2.7 Futures ................................................................. 31
  2.2.8 Conclusion ............................................................. 32

2.3 Parallel Nesting vs Speculation of Asynchronous Methods .......... 32

2.4 Combining TLS and STM ...................................................... 36

3 Java Transactional Futures runtime system ............................... 39
  3.1 Architecture ............................................................... 39
  3.2 API ......................................................................... 40
  3.3 Algorithm ................................................................. 42
    3.3.1 Metadata ............................................................. 45
      3.3.1.1 Transaction Metadata ....................................... 45
      3.3.1.2 Object Metadata ............................................. 46
    3.3.2 Transactional procedures ......................................... 47
      3.3.2.1 Read procedure .............................................. 47
      3.3.2.2 Write procedure ............................................. 49
    3.3.3 Committing transactions ........................................... 53

ii
3.3.4 Aborting transactions .................................................. 54
3.4 Optimizing read-only transactions ....................................... 56

4 Experimenting Results ....................................................... 57
  4.1 Experimental Settings .................................................... 57
    4.1.1 Vacation benchmark .................................................. 57
    4.1.2 Red-Black Tree benchmark .......................................... 58
    4.1.3 Platform .............................................................. 59
  4.2 Results ........................................................................... 59
    4.2.1 Vacation ................................................................. 59
    4.3 Red-Black Tree ............................................................ 62

5 Conclusions .......................................................................... 65

Bibliography ............................................................................ 70
List of Figures

1.1 Illustrative execution of STM+Futures utilization ........................................... 7
2.1 Difference between program’s original execution and its modification using STLS 27
2.2 Sequence diagram illustrating the invocation of parallel nested transactions . . . 35
3.1 JTF Architecture ................................................................................................. 40
3.2 Sequence diagram illustrating the invocation of Transactional Futures ........... 42
3.3 Transactional tree ............................................................................................... 43
3.4 Attribution of seqIDs .......................................................................................... 46
3.5 Metadata used by JTF .......................................................................................... 46
3.6 VBox structure ..................................................................................................... 46
3.7 Possible inefficiency in the read procedure ........................................................ 51
3.8 Example of an inter-tree conflict resolution. ....................................................... 52
3.9 Problem with overwrites ...................................................................................... 55
3.10 Removal of overwrites ....................................................................................... 55
4.1 Speed-ups and abort rate in high contention reservation operation. ............... 60
4.2 Speed-ups and abort-rate in low contention reservation operation.................. 61
4.3 Results in Red-Black tree benchmark with 0,1% write probability ................... 62
Acronyms

**JVSTM**  Java Versioned Transactional Memory
**TLS**  Thread-Level Speculation
**STM**  Software Transactional Memory
**JTF**  Java Transactional Futures
Chapter 1

Introduction

1.1 Context

Since their early ages, processors have been subject to a growing interest from the research community that focus their efforts in increasing processors computational power (Hennessy & Patterson 2007; Olukotun & Hammond 2005). Nowadays, many computer devices are powered by computational powerful parallel architectures. The vast computing power in modern hardware opens doors to new kinds of applications that require powerful processing which traditional single-core computers cannot offer. At the same time, computational requirements are ever increasing, both in the area of scientific and business computing. Software companies have applications in which fast runtime is a necessity or a competitive advantage (Buyya 2000; Spyrou 2009; Howard 2010).

However, it is hard for software developers to take advantage of such computational resources, as parallel programs are harder to design, implement and debug than their equivalent sequential versions (Quinn 2003; Harris, Larus, & Rajwar 2010). Concurrent programming can even be a double-edged sword, as badly designed parallel programs can perform worse than their equivalent sequential program (Harris, Larus, & Rajwar 2010).

Because parallel programming is such a hard task, researchers have been constantly trying to develop new models that ease the challenge of building non-trivial parallel programs (Quinn 2003; Romano, Carvalho, & Rodrigues 2008; Harris, Larus, & Rajwar 2010; Rundberg & Stenström 2001; Welc, Jagannathan, & Hosking 2005). Unfortunately, even state-of-the-art paradigms for parallel programming have crucial limitations that hinder their ability to extract the increasing parallelism in nowadays hardware.
1.2 Motivation

1.2.1 Java Futures

Nowadays, there are several models that ease the task of developing parallel programs, two of the most relevant examples are Transactional Memory (TM) and Futures. Futures are recently part of the Java 2 Platform Standard Edition 5. A Java future is a simple and elegant concurrency abstraction that allows the programmer to annotate method calls in a sequential program that can run in parallel with the corresponding continuation code.

Unlike traditional abstractions for explicit fork-join parallel programming in Java (Oracle 2013c; Oracle 2010), futures require substantially less effort from the programmer. With a simple interface that encapsulates many complex details such as thread creation, scheduling, as well as joining and return value synchronization, programmers can invoke asynchronous methods similar to the way they invoke synchronous methods (Oracle 2013b; Oracle 2013a; Welc, Jagannathan, & Hosking 2005).

Listing 1.1 shows an example of how the elegant Future’s interface allows such an abstraction. Objects that run asynchronous methods (call() methods) implement the Java Callable interface. The Callable object allows to return values after completion and uses generics to define the type of object which is returned. By submitting a Callable object to an Executor, an object of type Future is returned, and a new thread, which will execute the asynchronous method concurrently with the rest of the program (method continuation), is spawned.

Listing 1.1: Future Usage Example

```java
// Future_Usage_Example

public class MyCallable implements Callable<Integer> {
    public Integer call() {
        //do stuff
    }
}

public static void main(..){
    //do stuff
    ExecutorService executor = new Executor();
    Callable<Integer> asynchronousMethod = new MyCallable();
    Future<Integer> future = executor.submit(asynchronousMethod);
    //continuation
    Integer result = future.get();
    //do stuff
}
```
1.2. MOTIVATION

The Java `Executor` class automatically creates and executes threads (Oracle 2013a). Some extended implementations of the class even allow the programmer to easily define additional thread scheduling policies (Oracle 2013d).

The `Future` object can be used to check the status of a `Callable` and to retrieve the result from the `Callable` by calling the `get` method. The thread calling the `get` method will automatically block and wait if the asynchronous method has not yet been computed by the new thread.

Recently proposed extensions to the original Java futures promise further improvements to the abstraction: safe futures (Welc, Jagannathan, & Hosking 2005) alleviate the programmer from the burden of avoiding side-effects that might arise from concurrent accesses to shared objects between the asynchronous method and its continuation; Pratikakis et al. (Pratikakis, Spacco, & Hicks 2004) propose a framework that drastically simplifies programming with Futures by eliminating the need to satisfy type restrictions and by automatically inserting coercions that perform the claim/get operation on the Future at points where the value yielded by it is required.

The speed-up that one can attain with Futures is, however, limited. Since a program that is parallelized with Futures ensures that any data dependency stemming from the original sequential program order is respected in the parallelized execution. Such data dependencies severely restrict the effective parallelism that one can obtain by relying on Futures. Recent results show that even the optimized implementations of the Future abstraction rarely go beyond a relatively modest horizon of speed-up relatively to the original single-threaded program (Welc, Jagannathan, & Hosking 2005).

Hence, if the programmer wishes to harness the high parallelism of today’s and tomorrow’s multi-core computers, he often needs to resort to traditional explicit fork-join multi-threading programming.
1.2.2 Software Transactional Memory

Listing 1.2: STM Usage Example

```java
// STM_Usage_Example
public class MyThread implements Thread{
    public void start() {
        Begin();
        //do stuff
        //critical section
        //do stuff
        Commit();
    }

    public static void main(..){
        //do stuff
        ExecutorService executor = new Executor();
        Thread thread1 = new MyThread();
        Thread thread2 = new MyThread();
        Executor.submit(thread1);
        Executor.submit(thread2);
        //do stuff
    }
}
```

One of the most prominent example of fork-join multi-threaded programming paradigms is software transaction memory (STM). STM is a concurrency mechanism analogous to database transactions for controlling access to shared memory. With simple annotations, or begin and commit instructions, parts of the program’s computation are wrapped in a transaction, to which a runtime system grants atomicity and isolation properties (Harris, Larus, & Rajwar 2010) (Listing 1.2).

STM systems involve the programmer in the parallelization effort, by requiring him to reason about the semantics of the application and adapt it to the parallelization process. This allows STM systems to achieve considerable speed-ups, since some data dependencies, that could severely restrict the number of threads the system can parallelize effectively, can be removed.

Unfortunately, in order to avoid having to reason about complex program semantics, data dependencies and control-flows, programmers will typically choose a monolithic organization of coarse-grained threads when handparellelizing their applications (Zyulkyarov, Gajinov, Unsal, Cristal, Ayguadé, Harris, & Valero 2009; Barreto, Dragojevic, Ferreira, Filipe, & Guerraoui 2012). Thus, when using STM systems, programmers are dissuaded from exposing the full parallelism that the program effectively contains, as their STM transactions often have fine-grain parallelism that is left unexplored.
1.2. MOTIVATION

1.2.3 Software Transactional Memory + Futures

Listing 1.3: STM+Futures Usage Example

```java
// STM+Futures_Usage_Example

public class MyThread implements Thread{
    static public ExecutorService executor = new Executor();

    public void start() {
        Begin();
        //code block1
        MyCallable asyncMethod = new MyCallable();
        Future<Integer> future = executor.submit(asyncMethod);
        //continuation
        Commit();
    }

    public static void main(..){
        //do stuff
        Thread thread1 = new MyThread();
        Thread thread2 = new MyThread();
        Executor.submit(thread1);
        Executor.submit(thread2);
        //do stuff
    }
}
```

If programmers could use, in their transactions, the abstraction mechanisms that Futures provide, it would be easier and more appealing for them to explore the fine-grain parallelism often present in their transactions. By combining both components we would achieve the best of two worlds and overcome each one’s shortcomings. However, to the best of our knowledge, there is no solution that correctly combines Futures and STM.
Listing 1.3 and Figure 1.1 illustrate the execution we wish to support. One could consider a first naive solution of simply executing Futures inside a STM transaction, with no modifications on any of the two components. In such solution, the asynchronous method $(asyncMethod)$ invoked with Futures would run out of the control of the STM system. Concurrent accesses to shared data between that method and the $continuation$ would not be synchronized in any way. Furthermore, since both the asynchronous method and the $continuation$ run independently from each other, one could finish its execution without waiting for the other.

Combining both systems is far more complicated than that. Let’s analyse in more detail why such naive solution would not work by describing several problems that can emerge from it:

- **Atomicity breach** - The transaction commit procedure ensures that all operations wrapped in the transaction complete successfully, or that none of them appears to have been executed. However, the commit procedure cannot ensure this property to transactions running asynchronous methods, as this methods run outside of the STM system control. If the transaction commits successfully and the computation of the asynchronous method lasts beyond the transaction commit, only part of the transaction (code block1 and $continuation$) is guaranteed to complete successfully, the remaining part (inside $asyncMethod$) is entirely dependent of the computation of the asynchronous method ($asyncMethod$).

- Conflicts from concurrent shared data accesses, which break the program sequential semantics, can occur between the $continuation$ and the asynchronous method ($asyncMethod$) concurrent execution. More precisely, three kinds of conflicts can occur:
  
  - **Write-after-read (WAR)** - a conflict where an earlier (in the sequential order) asynchronous method ($asyncMethod$) writes to a location that the $continuation$ has already read from.
  
  - **Read-After-Write conflicts (RAW)** - where an earlier (in the sequential order) asynchronous method ($asyncMethod$) reads a location that the $continuation$ has already written to. This case happens with STMs that allow transactions to write directly to memory (direct update).
  
  - **Write-after-Write conflicts (WAW)** - where an earlier (in the sequential order) asynchronous method wants to write to a location already wrote by the $continuation$. While in STMs with direct update this conflict may happen after and before the transaction commits, in STMs with deferred update this conflict can only happen after the transaction commits, since the $continuation$ operations effects are still buffered before that.

Neither Futures nor STM systems ensure any safety guarantees that prevent/resolve these conflicts. Although it is important to mention that similar type of conflicts have already
been explored on proposed extensions to Java futures (Safe Futures) (Welc, Jagannathan, & Hosking 2005). They provide safety guarantees that ensure the results of executing concurrently Java futures and the *continuation* equals the results of executing them sequentially.

However, they do not take into account the presence of other program threads (in our case transactions) whose correctness criterium is different and depends on how the whole operation of transactions is executed and committed. Since the commit of safe futures is independent of the commit used in STM, transactions that execute concurrently with other transactions, which in turn run asynchronous methods, will observe the actions of those concurrent asynchronous methods and the actions of their respectively *continuation*. STM correctness criterium no longer holds in such situation.

We can even imagine more complex scenarios, where this naive solution would not work:

- **Intra-transaction asynchronous method conflicts** - We can have several concurrent asynchronous methods executing inside the same transaction. In such scenario, *WAR, RAW and WAW* conflicts might occur not only between asynchronous methods and the *continuation* (similar to what we had before with only one asynchronous method), but also between asynchronous methods of the same transaction.

- **Isolation breach** - STM systems ensure that the effects of incomplete transactions (not yet committed) are never visible to other concurrent transactions. However, the execution of asynchronous methods running inside transactions are out of the STM system control. The effects of already executed asynchronous method operations are immediately visible to other concurrent transactions in the program, even before the transaction that wraps that method commits. Isolation no longer hold in such scenario.

Besides investigating the correctness issues associated with the concurrent execution of STM transactions and Futures, another interesting research question is quantifying the scalability and efficiency achievable by such a system. A crucial part of STM systems overhead comes from conflict detection. Besides conflicts between transactions, an unified solution must also detect and resolve conflicts resulting from the execution of concurrent asynchronous methods inside transactions. Thus, the solution must add minimal complexity to conflict detection in order to lower the overhead implicit in the additional control.

We believe that by combining both systems we can achieve better results, in terms of effective parallelism, than systems that focus on exploring each one individually and at the same time take advantage of the abstractions that Futures and STM provide over concurrency issues.

The remainder of this dissertation is organized as follows. Chapter 2 gives an extensive description about the different design options that distinguishes STM systems, it also introduces
Thread-Level Speculation (TLS) relating it to Futures and ends by describing a state-of-the-art solution that combines TLS and STM systems. In chapter 3 we present the main contribution of this dissertation, a system that addresses the challenges we just described and effectively combines a state-of-the-art STM with Java Futures. In Chapter 4 we describe the experimental results performed on this system, comparing its performance to the baseline STM, identifying its main sources of overhead and its best use cases. Chapter 5 ends the document with a brief conclusion over all the work performed in this dissertation and its results.
Chapter 2

Related Work

The emergence of parallel computing architectures has pressured the research community to come up with new paradigms that ease the challenge of extracting parallelism from complex programs.

There are two prevalent models for parallel programming in multi-cores: task parallelism and data parallelism.

On the one hand, data parallelism consists in splitting computation over disjoint datasets between different processors. Every processor applies the same function to their own assigned partition of the dataset. However, data parallelism is not a universal programming model, as it is very specific to programs that rely heavily in workloads that can be split in disjoint partitions, e.g. disjoint partitions of a matrix, but difficult to apply to most data structures and programming problems (Harris, Larus, & Rajwar 2010).

On the other hand, task parallelism consists in splitting the workload of a program and assigned it to different threads, that in turn run on different processors in a multiprocessor system. Because threads might execute over shared data, synchronization is required in order to coordinate their accesses.

Researchers have tried to come up with new programming paradigms for the task parallelism model. This programming paradigms try to enhance the model’s mechanisms for abstraction and composition, which are crucial for managing complexity. Transactional Memory (TM) (Dragojević, Guerraoui, & Kapalka 2009; Hindman & Grossman 2006) and Thread-Level Speculation (TLS) (Rundberg & Stenström 2001; and Künle Olukotun 1998; Pickett & Verbrugge 2006) are two prominent examples of such.

This dissertation’s contributions apply to the context of task parallelism. Thus, hereafter we focus only on that paradigm. In the following sections we first introduce TM and give an extensive discussion about different design options that distinguishes TM systems from each other. Next we introduce automatic parallelization systems, as systems that allow programmers
to build their sequential programs independently from the parallelization process; in this category we introduce Thread-Level Speculation (TLS) and relate it to Futures. Finally we describe a state-of-the-art solution that combines TLS with TM.

2.1 Transactional Memory

Memory transactions are a similar abstraction to database transactions, for controlling access to shared memory. The critical sections of a program are wrapped in a transaction, which a runtime system coordinates in order to grant atomicity and isolation properties.

Atomicity requires that all operations wrapped in a transaction complete successfully, or that none of them appear to have been executed. Isolation requires that transactions do not interfere with each other, regardless of whether or not they are executing concurrently. This property gives the illusion that transactions are executed serially, i.e. one after the other. The effects of an incomplete transaction are never visible to other concurrent transactions.

By providing atomicity and isolation, TM systems are also able to preserve the consistency of a program. Consistency requires that, if a transaction modifies the consistent state of a program, the effects of such modifications should leave the program in another consistency state. Consistency is entirely program dependent, as it typically consists of a set of invariants, defined by the programmer, on data.

All TM systems have the goal of providing these properties to concurrent transactions. However, STM systems may have alternative implementations in order to do so. In the following sections we will discuss the main design strategies that distinguish TM systems from each other.

2.1.1 Correctness criteria

There are several correctness conditions for concurrent transactions that TM systems rely on (Harris, Larus, & Rajwar 2010):

- **Serializability** - STM systems are free to reorder or interleave transactions as long they ensure the result of their execution remains serializable. Serializability is the basic correctness condition in TM systems. It states that the result of executing concurrent transactions must be identical to a result in which these transactions executed serially, i.e. one after the other.

- **Linearizability** - Some TM systems might rely on stronger correctness criteria like linearizability, which requires that if a transaction completes before another transaction starts, then the former needs to appear to have ran before the latter. In linearizability one could
consider transactions as single atomic operations. The central distinction between serializability and linearizability is that serializability is a property of an entire history of transactions, while linearizability is a property of a single transaction. Another distinction is that linearizability includes a notion of real-time, which serializability does not: transactions must appear to take place atomically between their begin and commit times.

• **Opacity** - The previous conditions provide models for the the execution of committed transactions. However, they do not provide any definition of how running or aborted transactions should behave. Opacity (Harris, Larus, & Rajwar 2010) can be seen as a form of strict serializability, with the difference that it forces aborted transactions and the tentative work of running transactions to be part of the serial order without their effects being exposed to other transactions. Opacity has become the most consensual correctness criteria, being implemented by all recent STMs systems.

### 2.1.2 Weak and Strong Isolation

Unlike database transactions, TM correctness criteria must also consider the interactions between transactional and non-transactional access to shared data (e.g. accesses between transactions and other program threads):

• **Weak Isolation** - Some TM systems (Dragojević, Guerraoui, & Kapalka 2009) do not provide any conflict detection between their transactions and other remaining non-transactional program threads. In this situation, it is said that the TM system provides *weak isolation*. Programs that involve such conflicts can behave unexpectedly.

• **Strong Isolation** - The opposite of weak isolation is *strong isolation*, in which TM systems not only guarantee transaction semantics between its transactions (like *weak isolation*), but also between transactions and non-transactional code.

### 2.1.3 Optimistic versus pessimistic concurrency control

TM systems (Felber, Fetzer, & Riegel 2008) that employ a *pessimistic concurrency control* try to detect and prevent conflicts whenever a transaction accesses a location. In this approach, transactions claim exclusive ownership of data before proceeding, usually by acquiring a lock. The ownership lasts until the transaction either commits or aborts.

*Optimistic concurrency control* (Dragojević, Guerraoui, & Kapalka 2009) contrasts with the previous approach by allowing multiple transactions to access data concurrently and to continue executing even if a conflict occurs. Conflict detection and resolution is usually delayed until transactions wish to commit.
Experimenting results from different systems show that when conflicts are frequent, pessimistic approaches can be worthwhile, as transaction are forced to stop and wait when there is any remote chance that they will cause a conflict. This prevents transactions that are doomed to abort to proceed.

However, when conflicts are infrequent, it would be faster to just let transactions run freely without ever blocking, which would increase concurrency between them. In such situations, optimistic approaches have clearly better results.

2.1.4 Version management

In order to cancel operations when a transaction needs to abort due a detected conflict, STM systems have to manage the tentative writes that concurrent transactions execute. There are two different approaches in order to do so, direct update and deferred update.

In direct update, transactions directly modify data in memory. In order to revert operations, STM systems maintain an undo-log of the data, that hold their overwritten values. When a transaction aborts this log is used to restore the old values.

In deferred update, STM systems (Dragojević, Guerraoui, & Kapalka 2009) maintain a redo-log, usually named as write set, for each transaction. Transactional operations are buffered in this set and when a transaction wants to read a location that previously has written too, it consults its set. When committing a transaction, the values in the write set are copied to their corresponding memory addresses.

STM systems (Moore, Bobba, Moravan, Hill, & Wood 2006) that employ direct update usually incur high overheads when transactions need to abort, as the undo-log has to be checked and all previous values need to be restored.

In contrast, systems that use deferred update have very simple abort mechanisms, as they only have to discard the transaction write set. However, in contrast with direct update, they incur higher overhead in the commit process, as they have to copy the values from the write set to their correct memory locations.

2.1.5 Conflict Detection

There is a vast spectrum of techniques that different STM systems employ in order to detect conflicts between transactions. There are two aspects that one should take into account when conceiving a conflict detection technique: the granularity of conflict detection and the time at which conflicts are detected.

Conflicts can be detected at several levels of granularity. While some systems (Sreeram, Cledat, Kumar, & Pande 2007) opt for detecting conflicts at the level of complete data structures
2.1. TRANSACTIONAL MEMORY

(object-based), other systems (Felber, Fetzer, & Riegel 2008) opt to detect conflicts of individual elements of data structures (word-based).

Fine-grain approaches usually incur higher levels of memory overhead than coarse-grain ones, since they have to manage more information in the undo-log/redo-log. However, fine-grain approaches are able achieve higher throughput of committed transactions, as transactions might access the same data structure, but different elements of it (false conflicts).

The second aspect, is related to the time at which conflict detection occurs: whenever a transaction declares its intent to access data (eager conflict detection) (Felber, Fetzer, & Riegel 2008), or when a transaction wants to commit (lazy conflict detection) (Dice, Shalev, & Shavit 2006).

Eager conflict detection might incur higher overhead than lazy conflict detection, as every access performed by each transaction requires additional computation to detect conflicts. However, by detecting conflicts when they happen, eager conflict detection is able to prevent transactions that are doomed to abort from continue, whereas in lazy conflict detection, transactions that are doomed to abort continue their computation until they decide to commit.

2.1.6 Nesting

A nested transaction is a transaction (inner transaction) whose execution is contained in the dynamic extent of another transaction (outer transaction). Nested transactions can interact in many different ways, and different STM systems might implement different design choices.

• Flattened Nesting - Flattened Nesting is the simplest approach (Hindman & Grossman 2006). In this design choice, aborting the inner transactions causes the outer transaction to abort. The inner transaction sees the modifications to data made by the outer transaction and vice versa. However, committing a inner transaction has no effect over the state of shared memory until the outer transaction commits.

• Closed Nesting - In closed nesting each inner/nested transaction tries to commit/abort individually. When an inner transaction commits, its modifications to the program state become visible to the outer transaction, however those modifications only become visible to other threads/transactions when the outer transaction commits. When inner transactions abort they pass control to the outer transaction without aborting it, this allows partial rollbacks of the outer transaction.

Partial rollback allows to reduce the work that needs to be retried and increasing performance when aborts are common. However, closed nesting can have higher overhead than flattened transaction (Harris, Larus, & Rajwar 2010) and so, when commits are common, flattened nesting might be a better approach.
• *Open Nesting* - In open nesting, inner transactions are allowed to commit to shared memory (visible to all other threads/transactions) independently of the outer transaction, assuming that the outer transaction will commit. However, if the outer transaction aborts the modifications to program state performed by inner transactions have to be undone (Carlstrom, McDonald, Chafi, Chung, Cao Minh, Kozyrakis, & Olukotun 2006).

The undo operation requires that the TM system executes inverse actions of those executed by the inner transactions in reverse order. For example, to undo an add operation of a value to a data structure, the system would need to remove that value from the structure.

Open nesting breaches the isolation property between transactions, and by doing so, is able to increase concurrency and performance.

• *Parallel Nesting* - The previous models assume linear nesting, i.e. inner transactions execute sequentially, one after the other. In parallel nesting, we consider models where several inner transactions can execute in parallel within the same parent transaction.

The relations between transactions and nested transactions build an hierarchy of transactions that can be represented by a tree that we call the *nested transactional tree*. The root of this tree we call the *top-level transaction* and all the its descendants/leafs are children nested transactions.

### 2.1.7 Progress Guarantees

Rather than just trying to run transactions as fast as possible, another consideration related to performance is whether or not TM systems give any fairness guarantee when deciding which transaction should abort/delay its progress when a conflict is detected. A contention manager decides what a given transaction (attacker) should do in case it detects a conflict with another transaction (victim). Contention managers can implement several different contention resolution policies:

• *Passive* - The simplest policy where the attacker transaction aborts itself and re-executes (Felber, Fetzer, & Riegel 2008).

• *Polite* - The attacker transaction delays its progress for a fixed number of exponentially growing intervals before aborting the victim transaction. After each interval, the attacker checks if the victim has finished executing, if so the attacker proceeds without ever aborting the victim (Scherer & Scott 2005).

• *Timestamp* - The contention manager aborts any transaction that started executing after the victim transaction.
2.1. TRANSACTIONAL MEMORY

- **Greedy** - A timestamp is associated with a transaction when starts its first attempt to execute. A transaction aborts a conflicting transaction if the former has a younger timestamp than the latter, or if the latter is itself already waiting for another transaction. Unlike the previous approaches, this policy allows every transaction to commit within a bounded time, i.e. avoids starvation of transactions (Dragojević, Guerraoui, & Kapalka 2009).

There are many other policies one can apply when designing a TM system, however, no policy performs universally best in all settings (Harris, Larus, & Rajwar 2010). One should take into account the workload and the form of concurrency control used by the TM when deciding which policy to implement.

2.1.8 Providing Opacity

Some basic versions of STM allow a read-only transaction to experience a conflict and to continue executing, even though it is doomed to abort. These STMs are said to support invisible reads, where the presence of a reading transaction is not visible to concurrent transactions that might try to commit updates to objects being read. For these kind of STMs (Felber, Fetzer, & Riegel 2008), additional mechanisms have to be implemented in order to support opacity (Section 2.1.1), in which invisible transactions have the sole responsibility of detecting conflicts on shared data with transactions that write concurrently to it. Global clock and multi-version are examples of such mechanisms:

- **Global clock** - The STM (Felber, Fetzer, & Riegel 2008) systems maintains a single global counter that is incremented by every non-read-only transaction when it commits. Each transaction begins by reading this global counter, which is used to define the transaction’s position in the serial order. Additionally each data object records the counter (object version number) of the transaction which most recently committed an update to it. The transaction counter represents the instant at which transaction’s snapshot of memory is valid; the transaction aborts if it reads any object whose version number is lower than its counter.

- **Multi-version** - Instead of just storing the latest committed version of each shared object (single-version), some STM systems (Cachopo 2008) retain multiple versions of each object, each version committed at different timestamp windows (multi-version). Additionally each transaction maintains a timestamp window during which its current snapshot of the memory state is known to be valid. Whenever a transaction performs a read to an object, it is likely that a version of the object that falls within the transactional timestamp windows is available. If sufficient versions of an object are available, it is guaranteed that read-only transactions always commit. However, this approach might have high memory overheads, as it needs to maintain multiple versions of each shared object.
However, instead of implementing this additional mechanisms, some STMs (Sreeram, Cledat, Kumar, & Pande 2007) use the notion of visible reads in order to support opacity. In this approach, transactions wishing to commit updates to objects have to be aware of transactions that read from those objects in order to identify and explicitly abort the transactions with which it conflicts. However, using visible reads can be very costly has it introduces contention between readers.

2.1.9 STM vs HTM

TM has been subject of study both in hardware and software. Hardware transactional memory (HTM) approaches use processor’s hardware structures to track access to shared data and cache coherence protocols to detect data conflicts. HTM can provide several advantages over STM:

- By decoupling from the program, HTM systems can avoid the additional overhead implicit by the additional software components required by STM systems. HTM systems can operate on unmodified programs and to be, in general, more effective than STM systems.
- The flexible manipulation of pointers and the fixed data layouts implicit in low-level system code (e.g. C/C++ code) often contrains STM systems. HTM systems are more suited for these cases.
- HTM systems can provide strong isolation without requiring changes to non-transactional memory accesses.

Software transaction memory uses additional software components to track access to shared data and, detect and resolve data conflicts:

- While hardware approaches are restricted to use one scheme for all programs, STM approaches are more flexible, allowing them to adapt to different applications by implement a vast variety of different algorithms.
- Software is easier to modify and evolve than hardware. HTM approaches require changes to the processor’s architecture, which can be costly.
- STMs are not constrained by fixed-size hardware structures (e.g. caches).

2.1.10 Programming Model

Most STM systems provide simple interfaces that ease the task for programmers to handparallelize their applications. By using simple annotations, or begin/commit instructions, programmers can wrap regions of code in an atomic block.
2.1. TRANSACTIONAL MEMORY

A software component (e.g. compiler, dynamic class loader, etc) further prepares those regions of code to be run in the context of an transaction.

A runtime system, by applying the discussed techniques, ensures the serial equivalent dispatching of transactions, providing atomicity, consistency and isolation properties.

A major advantage of atomic blocks is that they do not require the need for the programmer to explicit create threads with low-level operations, as fork/join operations. Furthermore, programmers do not need to reason about which resources will be shared between threads and explicitly synchronize their access with lock-based abstractions, such as conditional critical regions or monitors. Atomic blocks synchronizes implicitly with any other atomic blocks that touch the same data.

This grants the property of *composability* to atomic blocks. *Composability* means that one can combine a set of individual atomic operations, and the result will still be atomic. Atomic blocks distinguishes from lock-based abstractions by providing this important feature. Unlike atomic blocks, lock-based abstractions often require to break the encapsulation of operations and expose the concurrency control they might use internally, or use additional concurrency control around them.

2.1.11 JVSTM

Several systems have been proposed to allow the introduction of STM into the Java environment (Hindman & Grossman 2006; Korland, Shavit, & Felber 2009). Java Versioned Software Transactional Memory (JVSTM) (Cachopo 2008) is a prominent example, consisting of a Java library for transactional memory that incorporates several desired features that cannot be found in other Java STM systems.

2.1.11.1 API

JVSTM involves programmers in the parallelization effort by requiring them to explicitly call the provided library. This system has a very simple API, as most applications need only to access two classes: `jvstm.VBox` and `jvstm.Transaction`. Listings 2.1, 2.2 and 2.3 show different use cases of the library provided by JVSTM.
Listing 2.1: Use of top-level transactions example

```java
// Top-level transactions

public class MyClass{
    VBox<Integer> i = new VBox<Integer>(); // transactional data

    public static void main(...){
        Transaction.begin();
        try{
            // transactional accesses
            Transaction.commit();
        }catch(CommitException ce){
            Transaction.abort();
        }
    }
}
```

The JVSTM introduces the concept of multi-version (Section 2.1.8) in the Java environment. This concept allows read-only transactions to never conflict with any other concurrent transaction, favouring applications where read-only transactions are predominant. The VBox (versioned box) class implements the multi-version concept and each instance of this class represents a transactional object. Each VBox holds several versions, that have been committed over time by transactions, of the correspondent transactional object. The `get` method provided by this class, returns the value of the VBox for the current transaction, and the `put` method modifies the value of the VBox for the current transaction.

With the Transaction class, programmers can control the start, commit and abort of transactions. The `begin` method starts a new transaction, and sets it as the current transaction for the current thread. The `commit` method tries to commit the current transaction, if this operation fails, an exception is thrown. Finally, the `abort` method aborts the current transaction.
2.1.11.2 Nesting transactions

Listing 2.2: Creation of a linear nested transaction example

```java
// Creation of linear nested transactions

class Myclass{
    public static void main(..){
        Transaction.begin();
        try{
            //transactional accesses
            Transaction.begin();
            //transactional accesses
            Transaction.commit();
            //transactional accesses
            Transaction.commit();
        }catch(CommitException ce){
            Transaction.abort();
        }
    }
}
```

JVSTM supports both linear and parallel nesting transactions. In the linear model, when the begin method is invoked, if a transaction was already active, a linear nested transaction is created. Transactions can have at most one nested child transaction running at any given time, thus sibling transactions execute sequentially one after the other. Because children transactions are executed by the same thread of their parent, while the children execute, the parent transaction waits until the children have finished.

Parallel nested transactions are represented by the `jvstm.ParallelTask` class, which implements the method to be executed by the parallel nested transactions. This method contains the transactional code to be performed by the parallel transaction. In the parallel model, transactions can have multiple nested child transactions running concurrently at any given time. Just like in the linear model, the parent transaction waits until all nested children have finished. The difference is that, while in the linear model the children transactions are executed by the same thread of their parent, in the parallel model each individual child nested transaction is executed by a new thread.
Listing 2.3: Creation of a parallel nested transaction example

```java
// Creation of parallel nested transactions

public class MyClass{
    public static void main(..){
        Transaction.begin();
        try{
            //transactional accesses

            List<ParallelTask<Integer>> tasks = new ArrayList<...>;
            tasks.add(new ParallelTask<Integer>(){
                @Override
                public Integer execute() throws Throwable {
                    //transactional accesses
                }
            });
            Transaction.manageNestedParallelTxs(tasks);
            //transactional accesses
            Transaction.commit();
        }catch(CommitException ce){
            Transaction.abort();
        }
    }
}
```

2.1.11.3 Versioning and Conflict Detection

Every top-level transaction in JVSTM has a *version* number which is assigned when the transaction is created. This number is fetched from a global counter that represents the *version* number of the latest read-write transaction that successfully committed. Child nested transactions also receive a *version* number which they inherit from the parent transaction.

To support parallel nesting, transactions are also associated with a *nClock*, which is an integer that is incremented by the commit of each child. Furthermore, children transactions also compute an *ancVer* map. This map is computed when the child transaction starts, by inherit the parent’s *ancVer* and adding it the parent’s current *nClock*. Thus, this map associates a *nClock* for each ancestor. Each *nClock* represents the versions of writes the transaction can read from that ancestor.

Each *VBox* contain two lists of writes, one list of values written by committed transactions (*permanent values*) and another list of values written by running transactions (*tentative values*). While the values in the *permanent list* are associated with a *version* number, the values in the...
**2.1. TRANSACTIONAL MEMORY**

tentative write list are associated with the value of the transaction’s nClock when the write was created.

When a value is written to a VBox inside a transaction, it is put in the VBox’s tentative write list. Additionally, the transaction acquires ownership of the list. As long as the transaction has ownership of the list, only this transaction and its descendants can write new tentative values. However, recall that parent transactions block execution until all children transactions have finished. This means that there is no concurrency in this lock between the parent and its descendants.

While JVSTM uses optimistic concurrency for top-level transactions, nested transactions use a pessimistic approach. This means that, if a top-level transaction finds a tentative write list locked, it continues executing, by writing the value in its local write-set. On the other hand, nested transactions do not own a local write-set, aborting when they find that the tentative write list is locked by a transaction that is not one of its ancestors in the nested transactional tree (hierarchy of transaction). These affected nested transactions are then re-executed sequentially in the context of the root top-level transaction (after all its other children are finished).

When a value of a VBox is read inside a top-level transaction, if the transaction or one of its descendants have previously written to it, the box returns the value from the tentative write list. This value corresponds to the value written by the last committed child. When a parallel nested transaction performs a read, if there is a value that was previously written by it, or if there is a value written by an ancestor before this transaction started, the value is returned from the tentative write list. This is enforced by checking the tentative value version (nClock) and the transaction version for that ancestor in the ancVer.

When transactions do not find a value inside the tentative write list, a value is returned from the top-level transaction’s local write-set. Otherwise, if no value was found neither in the tentative write list nor in the top-level transaction’s local write-set, a value is returned from the permanent write list. When the VBox fetches the value from the permanent write list, it does not always return its latest value, but a value that has equal or lower version number than the current transaction version number.

Conflict detection is done both eagerly, when read-write transactions read, and lazily, at commit time. A read-write transaction is only allowed to commit, or proceed after a read operation, if all of the VBoxes that it read have the same or a smaller version than the transaction’s current version. This means, that no other read-write transaction has committed changes to those boxes while this transaction was executing. Read-only transactions always commit, because they do not change the application state.
2.1.12 Other Java STM Examples

Similar to other STMs systems, DeuceSTM (Korland, Shavit, & Felber 2009) involves programmers in the parallelization effort by requiring them to explicitly annotate Java methods that should run in the context of a transaction. However, unlike JVSTM, DeuceSTM does not provide any way for programmers to wrap only parts of a method inside a transaction. Not all accesses to data in the heap need to be executed in a transactional context. Forcing the entire method to run in the context of a transaction might compromise the performance of the application, since transactional code has significant overhead comparing to non-transactional code.

DeuceSTM does not provide any library for programmers to use. Instead, in order to prepare annotated methods for transactional execution, each time a new class is loaded, a dynamic class loader reads the class and instruments it with bytecode-to-bytecode rewriting.

Unlike JVSTM, which supports the closed-nesting model, DeuceSTM supports flat-nesting, in which partial rollback is not possible. Furthermore, DeuceSTM does not support parallel nesting.

DeuceSTM supports a feature that distinguish it from other STM implementations. In this system, programmers are allowed to plug in different STM algorithms and the DeuceSTM runtime system queries it for further actions whenever a transactional event occurs (e.g. when transactions perform transactional read/writes, or when transactions abort/commit).

AtomJava (Hindman & Grossman 2006) is another Java STM implementation. It uses source-to-source translation to produce instrumented source code that can be compiled by any Java compiler. It provides strong isolation and uses locks in a pessimistic concurrency control. By using source-to-source translation, AtomJava imposes a major limitation on programmers preventing them from using compiled libraries.

2.1.13 Conclusion

STM transactions are able to provide several abstractions that hide complex concurrency issues away from the programmer that other programming constructs cannot offer.

Furthermore, in theory, when compared with automatic parallelization approaches that part from the original sequential program (e.g. compilers, TLS, etc), STM systems are able to attain higher levels of parallelization, as programmers can remove complex data dependencies and control-flows that often limit the number of threads that automatic systems can parallelize effectively.

However, transactions are not a panacea. When handparallelizing their programs, programmers can still use transactions incorrectly, e.g. starting a transaction but forgetting to commit it or abort it. Furthermore, programmers might also write transactions that are too short or too
long. While short transactions can incur synchronization penalties that may outweigh the performance gains of parallelization, long transactions might result in program errors, as programmers might place multiple operations inside a transaction in one thread, when the intermediate state between those operations needs to be visible to other threads.

Furthermore, often used long transaction (Zyulkarov, Gajinov, Unsal, Cristal, Ayguadé, Harris, & Valero 2009; Barreto, Dragojevic, Ferreira, Filipe, & Guerraoui 2012), might dissuade programmers from exposing the full parallelism that the program effectively contains, as long transactions often contain hidden fine-grain parallelism that is left unexplored.

2.2 Thread-Level Speculation

Automatic parallelization systems are able to work without the access to the original application source code, and automatically parallelize programs with minimal input from the programmer.

They allow the parallelization process to be done independently from the creation of the original application. Programmers can build entirely sequential programs without concerning about any issues involved in parallel programming, and in the end still have their original program parallelized.

In the following sections we will briefly introduce parallelizing compilers, which was the first approach to automatic parallelization, and then TLS, an approach that focuses in speculative parallelization.

2.2.1 Parallelizing Compilers

Parallelizing Compilers were the first approach to automatic parallelization. These systems try to automatically extract concurrency from a sequential program by statically analyzing its source code.

The Polaris (Blume, Eigenmann, Faigin, Grout, Hoeflinger, Padua, Petersen, Pottenger, Rauchwerger, Tu, & Weatherford 1995) compiler is one of the most successful examples of such systems. It consist of a basic infrastructure that takes Fortran programs as input and combines several techniques to try overcome the limitations of other parallelizing compilers.

Examples of techniques employed by the Polaris are interprocedural symbolic program analysis, scalar and array privatization, symbolic dependence analysis, and advanced induction and reduction recognition and elimination.

Interprocedural analysis is done by inline expansion performed by a Polaris driver routine that repeatedly expands subroutines and function calls in a top-level program unit. This analysis enables more precise analysis information and overhead elimination of small routines calls.
CHAPTER 2. RELATED WORK

Recognition and removal of inductions and reductions are two techniques applied to loops. They inhibit the parallel execution of different iterations of loops by breaking data dependencies between them.

Symbolic dependence analysis is another crucial analyses technique to determine what statements or loops can be safely executed in parallel.

Scalar and array privatization is a transformation targeting loops in order to parallelize them. It identifies scalars and arrays that are used as temporary work space by an iteration of a loop, and allocates local copies of them.

In more recent works (Jenista, Eom, & Demsky 2011; hun Eom, Yang, Jenista, & Demsky 2012), further progresses were made to this model of automatic parallelization. However, they still suffer from the intrinsic model’s limitations. By relying exclusively in static analyzes, this kind of systems fail to work with many irregular applications [34, 35] that employ data and interprocedural dependencies, that in turn are hard to analyze in a full static way.

2.2.2 Thread-Level Speculation

Meanwhile, new automatic parallelization models emerged in order to overcome the limitations of parallelizing compilers. One example of such models is Thread-level speculation (TLS) that distinguishes from parallelizing compilers by allowing regions of code to run in parallel, even though they cannot be statically proven to preserve the sequential semantics under parallel execution.

TLS systems differentiate from one another by parallelizing different regions of code. Those regions can either be loops (loop level speculation (LLS)) or function calls (method level speculation (MLP)).

Those regions of code, that might contain true dependencies, are executed concurrently out of sequential order by fine-grained tasks. Figure 2.1 shows the executions differences between the original single-threaded programs in Listing 2.4 and its modified version using speculation at method calls.

As execution reaches the call to method1, a new child thread/task is spawned. While one of the threads enters method1’s body, the other one begins executing speculatively the first instruction past the callsite (code block2), also known as continuation.

When the thread executing method1 returns, two things may happen with the operations performed by the speculation: if the results from the concurrent execution between method1 and the continuation are equivalent to the results of their original sequential execution, then speculation is committed; if speculation fails, the continuation is normally re-executed, i.e. not speculatively, because it is already in the sequential order.
2.2. THREAD-LEVEL SPECULATION

TLS ensures that the program executed the same way that it did originally. The correctness criterion of TLS requires that the concurrent execution of its tasks has the same results as their sequential execution in the original sequential program version. Note, this is different from the correctness criteria of TM.

Listing 2.4: Pseudo-code of a single-threaded program

```java
method()
    //code block1
    method1();
    //code block2
}
```

Figure 2.1: Executions differences between original sequential version and parallel version using STLS depicted in Listing 2.4.

In order to fulfill this criterion, TLS systems must satisfy the following requirements.

- Detect conflicts RAW, WAW and WAR between concurrent threads;
- Rollback speculative writes and restart speculative tasks whenever any of the above conflicts occurs.

Just like TM systems, TLS must have mechanisms to detect and resolve conflicts resulting from the concurrent execution of its tasks. In order to do so, TLS uses mechanisms that are
similar to those used in TM. In the following sections, we will briefly discuss hardware and software implementations of TLS, focusing on the latter. Finishing with a comparison between Futures and TLS.

### 2.2.3 Hardware Thread-Level Speculation

TLS has often been subject of hardware research, and a variety of general purpose machines have been proposed and simulated (Steffan, Colohan, Zhai, & Mowry 2005; Steffan & Mowry 1998; Akkary & Driscoll 1998; Chen & Olukotun 2003).

Most current hardware designs could however be classified as hybrid hardware and software approaches, as they rely on the assistance of various software component extensions. Compilers and runtime processing help these systems identify parallel regions and insert appropriate TLS directives for the hardware.

Hardware TLS approaches have always used one same basic infrastructure: private caches that are kept consistent by a cache coherence protocol. Validation of speculative reads and writes becomes simpler than those used in software approaches, because all conflicting accesses result in cache invalidation or coherence misses.

Checking data dependencies requires extra complexity in terms of more state information per cache block, a more complex cache protocol, and using part of the cache space to keep shadow copies where speculative operations are buffered in order to prevent speculation from corrupting program state. Exhausting this space causes a performance impact, which restricts gains.

### 2.2.4 Software Thread-Level Speculation

Another way of supporting TLS is to implement it entirely in software (STLS). In general, software approaches require explicit code and use deferred update in order to buffer reads from, and writes to, main memory by speculative threads. To the best of our knowledge, deferred update is the only approach used in STLS solution proposed so far, since it prevents WAR conflicts, as all writes are stored inside thread local buffers. Note, this prevention is not possible with direct update. In the following sections we introduce some STLS implementations and discuss their main strengths that we believe it make them stand out from other implementations.

#### 2.2.4.1 S-TLS

One of the first software TLS systems was proposed by Rundberg and Stenstrom (Rundberg & Stenström 2001). Their framework is composed by a compiler and a runtime system.
For speculative reads and writes whose addresses cannot be disambiguated statically, the compiler adds an additional data structure (buffer for deferred update) for each shared data and checking code that detects and resolves data dependence violations in runtime.

Similar to TM systems that employ deferred update, S-TLS has to write back speculative writes back to their corresponding memory locations when speculation ends. Some STLS systems conservatively commit all objects updated by each task one-by-one across all tasks. This potentially costly approach is named serial commit.

S-TLS distinguishes from this systems by employing a parallel commit. Even if several threads have modified a variable, only the latest task in serial order needs to have its value committed. S-TLS is able to track this task by assigning numbers to tasks in a way it reflects their sequential order of appearance. Furthermore, every buffered speculative write is associated to a task number. When speculation ends, for each memory value only the speculative writes with the highest number need to be committed.

2.2.4.2 SableSpMT

The SableSpMT (Pickett & Verbrugge 2006) is a Java automatic parallelization framework that employs method-level speculation and results from an extension to SableVM, an open source VM. It relies in ahead-of-time static analysis and virtual machine modifications in order to modify program bytecode, and manage concurrent task execution at runtime.

This framework distinguishes from other Java speculation systems by considering the full Java semantics, including all bytecode instructions, garbage collection (GC), synchronization, exceptions, native methods and dynamic class loading.

In addition to preparing method bodies for speculative execution, SableSpMT provides various TLS support facilities that interact with the application bytecode, there is, at least, one facility worth mentioning, return value prediction (RVP) (Hu, Bhargava, & John 2003). Many STLS systems that employ method-level speculation, force speculation to stop, when the return value from the method that triggered speculation is needed but it is still not available. A more efficient alternative is the use of return value prediction. A return value is predicted for non-void methods, which allows the speculative thread to proceed. This can significantly improve the performance of a Java STLS that employs MLS.

Technically, any arbitrary value can be used as a prediction, although the chance of speculation success is greatly reduced by doing so.

Return value predictors are associated with individual callsites, and use context, memoization, and hybrid strategies, amongst other.
2.2.4.3 JaSPEx

Ivo Anjos et al. (Anjo & Cachopo 2009) propose the JaSPEx system, which is able to parallelize automatically sequential Java programs by employing method-level speculation using a Software Transactional Memory system.

JaSPEx does not require any modification to the Java VM or to the Java bytecode specification.

The system distinguishes from other STLS systems by using a STM system to back up speculative execution. In order to detect when speculation violates sequential execution semantics, speculative threads run in the context of a STM transaction.

2.2.5 Hardware vs Software

Hardware TLS proposals have the advantage of operating on unmodified binaries and are in general more effective than software ones. Although, they involve complex and expensive changes to the basic cache-protocol and are limited since they must decide how to break programs into speculative threads without knowledge of high-level program structure. In addition, the potential for speculative storage overflow limits these implementations from exploiting the full fine-grain parallelism available in applications. As a result, commercial chip-multiprocessors do not yet offer TLS support.

Both Hardware and Software approaches have their unique flaws and advantages. Studies (Oancea, Mycroft, & Harris 2009) show that there is no universal solution for TLS. While hardware approaches are restricted to use one scheme for all programs, software TLS has the comparative advantage that it may adapt to different applications by composing instances of various TLS solutions, increasing the potential for extracting parallelism.

While hardware approaches bind their support for speculation into the physical hardware, software systems can provide their speculative support on applications that run on hardware that does not support speculative execution.

2.2.6 Java Continuation

Many Thread-Level Speculation systems face the similar problem of, when conflicts are detected, having to revert the program execution to a certain previous point in time. Let us recall the example in Figure 2.1. When speculation fails the continuation must revert execution to the point right after the invocation of method1. One way of doing this is through the use of Continuations, which is a data structure that represents the computational process at a given point in the process’s execution. Until now we have used the term of continuation to refer to the code that precedes the invocation of an asynchronous method. From here on now until the
2.2. THREAD-LEVEL SPECULATION

end of this Section 2.2.6 we will use this term with this new meaning. After a *Continuation* is created it can be used by the programming language in order to restore the program execution to the state represented by that *Continuation*.

There are different ways of implementing *Continuations*. For example, in the C language this feature can be provided by the `setjmp` library. In this library, the `setjmp` function saves the current program state into a platform-specific data structure named `jmpbuf`. This data structure, can be used at some later point of program execution by the function `longjmp` to restore the program state saved in `jmpbuf`.

In the Java environment, *Continuations* are implemented by *First-class continuations*, which are programming languages constructs that give the ability to save the program’s execution state, i.e. the program counter and the stack state, at any point in time and return to that point at a later in the program execution. To the best of our knowledge, there are two ways of doing this in Java:

- **Javaflow** - Javaflow (The Apache Software Foundation 2008) is a Java library that provides routines which programmers can use to specify the moment where the program state should be saved. This state is encapsulated in an object that can subsequently be used with another routine to restore that program execution. Javaflow also relies in bytecode modification to enhance the classes that run inside continuation-enabled environment.

- **Modified Java Virtual Machines** - Some versions of Java Virtual Machine (JVM) provide programmers with *Continuations* support. One such example is a modified version of OpenJDK Hotspot VM (Anjo & Cachopo 2013) that was extended with support for first-class continuations. However, this modified version still inherits Hotspot’s high-performance features such as just-in-time compilation, adaptive optimization, garbage collection, and support for the latest Java versions. Similary to Javaflow, this JVM provide users an additional library to save and restore the program state, the main difference is that it doesn’t require changing the program classes bytecode, which can increase the class files size and slow down the program execution (The Apache Software Foundation 2008).

2.2.7 Futures

Java futures can be seen as a form of MLS, as they can be used to explore parallelization in programs by forking at method calls. A Future represents an asynchronous method call that executes in background, and the program can later use the Future to retrieve the result of the asynchronous method computation. However, if the computation of the method is not yet complete, then a synchronization point is formed and the program is forced to block until the result is ready.
Futures provide several abstractions for adding concurrency to a sequential program similar to TLS. Programmers can abstract from using complex fork/join instructions and synchronization operations while parallelizing their applications. However, unlike TLS, the basic implementation of Futures provided in the Java Development Kit (JDK) (Oracle 2011) lacks concurrency control between the asynchronous work going on in different future tasks.

Safe futures (Welc, Jagannathan, & Hosking 2005) uses techniques similar to those used in STLS to avoid these problems. With safe futures, even though some parts of the program are executed concurrently and may access shared data, the equivalence of serial execution is safely preserved.

However, under the current programming model of safe futures, this safety does not extend to cover the interaction between Futures and other program threads (e.g., STM transactions). This feature has been purposed multiple times (Welc, Jagannathan, & Hosking 2005; Harris, Larus, & Rajwar 2010), but was never implemented.

2.2.8 Conclusion

TLS is a great promise as an automatic parallelization technique, as speculation allows parallelization of a program into tasks, even without prior knowledge of where true dependencies between tasks may occur. All tasks simply run in parallel until a true dependence is detected while the program is executing. This greatly simplified the parallelization of programs because it eliminates the need for human programmers or compilers to statically place synchronization points into programs by hand or at compilation time. However, departing from the original sequential program, TLS systems face complex program semantics, data dependencies and control-flows that hinder its ability to increase the number of tasks that can be parallelize effectively (Welc, Jagannathan, & Hosking 2005; Oancea, Mycroft, & Harris 2009; Anjo & Cachopo 2012).

2.3 Parallel Nesting vs Speculation of Asynchronous Methods

Parallel nesting is a very similar concept to what we wish to support in a STM that supports Futures:

- Exploring the inner-parallelism of transactions - Both parallel nested transactions and futures have the similar goal of exploring the parallelism still present inside transactions. However, unlike the model we wish to support, in the model of parallel nested transactions, child transactions may end up being serialized in an order that is different than the sequential order by which they appear in the original parent transaction’s code.

If, with regard to the underlying application’s semantics, both nested transactions are commutative, serializing them out of program order does not violate correctness (according
2.3. PARALLEL NESTING VS SPECULATION OF ASYNCHRONOUS METHODS

However, using the parallel-nested model places a new burden on the programmer - the burden of inferring whether the nested transactions are commutative or not. If not, nested parallelism is not an option to the programmer. Furthermore, if in doubt about the above question, the programmer should also opt for not spawn nested-parallel transactions.

In contrast, the transaction futures model does not require the programmer to reason about commutativity when parallelizing transactions with asynchronous methods using the transactional futures abstraction.

Nevertheless, there are situations where the sequential semantic of the parent transaction code needs to be preserve in the execution of its parallel version. In such situations, the transaction’s code cannot be parallelized by using parallel nested transaction, as this semantic is not guaranteed to be preserved with the concurrent execution of parallel nested transactions. Take the example of the pseudo-code in Listing 2.5. Assume that, in the `analizeGraph` method, the transaction travels a graph and, with a low probability for each node, it changes the content of its nodes. Afterwards, in the `reStructGraph` method, the transaction, once again, travels the graph, but this time, based of the changes performed in the previous method, it changes its structure, by deleting or changing the position of its nodes. This is similar to the algorithms used in a red-black tree structure or a mark and sweep garbage collector (Smith & Nair 2005). If we wrap each method inside a parallel nested transaction, the nested transaction responsible for the execution of the `reStructGraph` method might be serialized before the nested transaction responsible for the execution of the `analizeGraph` method. This compromises the correct execution of the parent transaction, since the structure of the graph will not be changed accordingly the changed performed by the `analizeGraph` method. Hence, in this situation, we cannot parallelize the transaction by using parallel nested transactions.

However, since in the model we wish to support the sequential semantic is always preserved, one can parallelize both methods by speculating them with asynchronous methods. Furthermore, if the `analizeGraph` method, in most cases, does not perform any changes on the nodes, the speculation of both methods will often succeed. In such situation, speculation might extract good performance results.

Listing 2.5: Example of a transaction that cannot be parallelized with parallel nested transactions.

```java
Transaction.begin();
analizeGraph();
reStructGraph();
Transaction.commit();
```

- Atomicity and Isolation - In a STM that supports Futures, to preserve the atomicity and
isolation of a transaction invoking an asynchronous method, the execution of that method needs to be contained in a transactional context. However, the transactional context of the asynchronous method should not be independent of the top-level transaction were it was invoked. If the top-level transaction aborts, so the execution of the transaction running the asynchronous method must abort. Furthermore, the top-level transaction can see the effects over shared data produced by the transaction running the asynchronous method and vice-versa, but those effects can only be visible to concurrent transactions when the top-level transaction commits. This scenario describes a very similar scenario to the one supported by parallel nested transactions.

- **Validating asynchronous methods** - Before they commit, parallel nested transactions need to validate themselves to ensure that no conflicts occurred between their execution and the concurrent execution of other parallel nested transaction inside the same nested transactional tree. In a STM that supports Futures, WAR, RAW and WAW conflicts can occur inside a transaction that invokes asynchronous methods. The transaction invoking asynchronous methods can only commit once those methods have been detected and resolved.

Because of these similarities, we chose to use JVSTM as our baseline STM algorithm, since, to the best of our knowledge, it is the only Java STM implementation that supports parallel nesting. With this decision we can re-utilize several data structures and algorithms used by JVSTM:

- **Atomicity and Isolation** - JVSTM provides several features we can re-use in order to preserve the isolation and atomicity of transactions running asynchronous methods:

  - **Invocation of asynchronous methods** - JVSTM uses a specialized thread executor to submit parallel transactions for execution (Figure 2.2). This executor extends the Java Executor class and, before the new thread starts running the transactional code (execute method), it runs the call method of the jvstm.ParallelTask class. Once inside this call method, the new thread wraps the transactional code inside a child parallel transaction. This is done by invoking Transaction.begin() before invoking the execute method. We will re-use this executor, so we can wrap the asynchronous method inside a child transaction, before the new thread starts running it.

  - **Transactional context** - The relation between an ancestor and a descendant parallel transaction allows them to read each other writes. This also happens between a transaction and the asynchronous methods it invokes. This often implies that transactions have to access other transaction’s data structures in order to find the correct value to read. This is a very crucial operation, since iterating over several data structures of different transactions may degrade the performance of the transaction’s read operation. JVSTM as already proven to extract considerable good results with its parallel
2.3. PARALLEL NESTING VS SPECULATION OF ASYNCHRONOUS METHODS

Figure 2.2: Sequence diagram illustrating the invocation of the parallel nested transactions of Listing 2.3 in Section 2.1.11.2.

nesting algorithm (Diegues & Cachopo 2013). For this reason, just like parallel nested transactions, we will use VBoxes tentative write list to buffer the transactional operations performed by transactions running asynchronous methods. However, recall from Section 2.1.11.3 that, when a transaction writes to this list, it takes its ownership, aborting all other transactions that do not descend from it. As we will explain later, we decided to allow all transactions write to this list, as long as they belong to the same transactional tree of the first transaction writing to the list. This means, that the lock becomes local to the tree. We decided to change this because it could result in a better trade-off between the size of the list (which influences the performance of the transactional read operation) and the level of parallelism allowed inside a top-level transaction.

- **Finishing an asynchronous method** - Similar to parallel nested transactions, when a transaction running an asynchronous method performs a transactional read, that read can only target an ancestor write. If there are several asynchronous methods running inside one top-level transaction, one could think that the best solution would be to fetch the most recent write from the transaction running the asynchronous method that precedes (in the sequential order) the transaction attempting the read. The problem with this solution is that we can never be sure if the transaction owner of that write will not try to write again and therefore invalidate the read. For this reason, we will use part of the commit algorithm used by JVSTM, to make the writes of asynchronous methods visible in the
ancestor’s of other concurrent asynchronous methods that appear after in the sequential order. Tentative writes inside the VBoxes tentative write list are associated with an ownership record which contains information about the owner of the write (a transaction) and the version of the write ($nClock$, recall Section 2.1.11.3). Whenever a transaction commits it simply propagates its $orec$ to its parent transaction. This entails setting the owner to the parent, and the version to the version ($nClock$) the transaction has for the parent, inside its $ancVer$, plus one. As a result, the commit procedure of parallel nested transactions/asynchronous methods can be very lightweight in practice.

With all these similarities, one could think that a basic solution to support futures inside transactions, would be to wrap the execution of asynchronous methods inside a child parallel nested transaction. Taking as example Figure 1.1, this means wrapping the execution of asyncMethod inside a parallel nested transaction. Such solution would be incorrect for the following reasons:

- In the model we wish to support, asynchronous methods run concurrently with the continuation, which corresponds to part of the top-level transaction’s code. However, JVSTM does not allow the code from the parent transaction to run concurrently with its nested children.

- Parallel nesting provide a weaker correctness criteria (opacity) to its transactions than the one we need to provide to asynchronous methods running inside a transaction. In the model we wish to support, the effects over shared data of the concurrent execution of an asynchronous method and its following continuation, needs to be equivalent to their sequential execution.

### 2.4 Combining TLS and STM

To the best of our knowledge, there is only one work that tried to use speculation as a way of exploring fine-grain parallelism hidden in STM transactions (Barreto, Dragojevic, Ferreira, Filipe, & Guerraoui 2012). The authors describe an algorithm, TLSTM, that leverages an existing STM with STLS capabilities.

By using this system, programmers first handparallelize their application into coarse-grained threads using STM. STM threads are then further handparallelized into finer-grained parallel tasks using STLS.

In their C++ implementation, the authors extend a C++ STM system (SwissTM) (Dragojević, Guerraoui, & Kapalka 2009) by adding new types of conflict detection and new data
structures. With this extension, SwissTM is, not only able to resolve conflicts between transactions, but also resolves \textit{WAW} and \textit{WAR} conflicts between STLS threads of the same/different Swiss transaction.

However, unlike Futures, this system fails to achieve any kind of abstraction. In order to use TLSTM, programmers have to explicitly create STLS tasks with fork instructions, assign and schedule methods to be executed by STLS tasks, and add wait barriers and join instructions to coordinate STLS tasks.

This work makes it hard for non-experienced programmers to use this system, because to do so, one needs to be aware of many complex concurrency issues involved in fork-join multi-threaded programing.

Futures hide these complex concurrency issues away from the programmer by providing an elegant and minimalist interface that automatically creates and schedules threads, and automatically deals with join and return value synchronization.

More than the possibility of concurrent programs that expose as much parallelism as the ever increasing hardware thread count, we want concurrency issues to be encapsulated and hidden as much as possible away from the programmer.

We believe TLSTM still has not yet reached a level of maturity sufficient to allow it to be used by developers without high prior level of expertise in parallel processes.

Our system will distinguishes from TLSTM for being a Java implementation of the unified STM+STLS solution. Furthermore, by using Futures, programmers that are not aware of many of the concurrency issues of using threads can still explore the fine-grain parallelization in their STM transactions, as Futures allow them to abstract of such issues.
Chapter 3

Java Transactional Futures runtime system

The main contribution of this dissertation is an unified runtime middleware for Java, called Java Transactional Futures (JTF), that unifies STM and Java Futures. In our runtime system, Futures invoked inside transactions are called Transactional Futures. These Transactional Futures, are managed by the JTF runtime system which addresses the problems discussed in Section 1.2.3: the atomicity and isolation breach of transactions running asynchronous methods; and the WAR, RAW and WAW conflicts between the concurrent execution of those methods. JTF runtime addresses these problems by extending the JVSTM with TLS which ensures that the result of the concurrent execution of asynchronous methods invoked with Transactional Futures is equivalent to the result of executing those methods sequentially.

3.1 Architecture

The JTF runtime is fully implemented in software and it runs over a modified version of OpenJDK Hotspot Java Virtual Machine, allowing JTF runtime to take advantage of first-class continuations support (Anjo & Cachopo 2013). Furthermore, by running on top of this virtual machine, the runtime system can take advantage of every functionality and optimization of any modern Java Virtual Machine (JVM). Examples of such are, dynamic compilation, garbage collection, Java 6 support and all Java optimized concurrent primitives.

An overview of the JTF runtime architecture is depicted in Figure 3.1. With JVSTM, programmers can handparallelize their application by wrapping parts of the program’s code in transactions. Furthermore, programmers can explore the fine-grain parallelism inside their transactions by using the Java java.util.concurrent.Future class inside the Java Class Library (JCL). This class allows programmers to turn synchronous invocations of methods into asyn-
chronous invocations. However, as discussed in Section 1.2.3, without any additional control, JVSTM and Futures will run independently and the discussed conflicts may occur.

The JTF runtime consists in an additional module in JVSTM. This runtime manages the concurrent execution of Java Futures invoked inside JVSTM transactions, preserving the correctness criteria of the STM. For this reason, in the JTF runtime, we call Java Futures running inside transactions as Transactional Futures. By positioning itself inside JVSTM, the JTF runtime hides from the programmer and allows the re-use of JVSTM interface, without requiring an additional API in order to combine both systems. This is an important property, since extending the interface would require additional learning from the programmer and consequently would complicate the parallelization process. Furthermore, this allow us to preserve the simplicity and the abstractions that the JVSTM and Futures interfaces provide over concurrency issues such as thread creation, scheduling, joining and return value synchronization, as well as synchronization on concurrent accesses over shared data.

3.2 API

Listing 3.1 shows an example of how programmers can use Transactional Futures to hand-parallelize their application. Both transactions and asynchronous methods are managed the same way as before. The only exception is that the Callable instance should be submitted in the jvstm.Transaction class instead of the traditional Java Executor class. This modification is necessary in order for JVSTM to have control over the asynchronous method execution.
Listing 3.1: Application parallelized with JTF

1 // JTF API
2
3 public class MyClass implements Callable<...>{
4 
5     public ... call(){ //the asynchronous method to be executed by the
6         Transactional Future
7             //transactional accesses
8     }
9 
10     public static void method(...){
11         Callable<...> asynchronousMethod = new MyClass(); //class containing the
12             asynchronous method (call())
13 
14             Transaction.begin();
15             try{
16                 //transactional accesses
17                 Future tasks =
18                     Transaction.manageParallelTask(asynchronousMethod); //submission
19                     of the Transactional Future
20                     //continuation
21                     Transaction.commit();
22             }catch(CommitException ce){
23                 Transaction.abort();
24             }
25         }
26 }
27
28
CHAPTER 3. JAVA TRANSACTIONAL FUTURES RUNTIME SYSTEM

Figure 3.2: Sequence diagram illustrating the invocation of the Transactional Future of Listing 3.1.

To submit the Transactional Future inside this class, programmers should call the Transaction.manageParallelTask(Callable) method (line 16), which was the only extension introduced by the JTF runtime in JVSTM’s interface.

Figure 3.2 shows how Transactional Futures are submitted in JTF. This is very similar of how parallel nested transactions are submitted for execution in JVSTM (Figure 2.2 of Section 2.3). Once inside the jvstm.Transaction class code, an instance of jvstm.ParallelTask is created. The creation of this class, accepts as argument the asynchronous method passed by the client application (Callable c). The ParallelTask class contains the call method to be executed by the new thread immediately after it starts running. Inside this method, the new thread calls the Transaction.begin method before calling the asynchronous method (c.call()) passed by the client application. From the point on the Transaction.begin method is invoked, all the execution of the new thread, which includes the asynchronous method, is contained in a new child transaction.

3.3 Algorithm

As mentioned in Section 1.2.3, in order to preserve the isolation and atomicity of transactions, asynchronous methods invoked inside a transaction need to run under the STM control. More precisely, asynchronous methods need to run in the same transactional context of the transaction where they were invoked. The JTF runtime accomplishes this in a very similar way as parallel nested transactions are managed in JVSTM. Once an asynchronous method is submitted, a new
3.3. ALGORITHM

Figure 3.3: Example of a transactional tree, the root ($T_0$) represents a top-level transaction, the nodes marked as $F$ represent a transaction running an asynchronous method and nodes marked as $C$ represent a transaction running the code that follows the invocation of that asynchronous method.

child transactional context is created. This new transaction will run the asynchronous method concurrently with the rest of its parent transaction (continuation).

Running in the context of a child transaction makes the execution of the asynchronous method dependent of the top-level transaction were it was invoked. If the top-level transaction aborts, so the execution of the transaction running the asynchronous method will abort. Furthermore, the top-level transaction can see the effects over shared data produced by the transaction running the asynchronous method and vice-versa, but those effects can only be visible to concurrent transactions when the top-level transaction commits.

In the presence of conflicts between the continuation and the asynchronous method, the continuation must discard all effects over shared data and re-execute. To accomplish this, JTF starts another child transactional context to run the continuation. This way we can discard all effects performed by the continuation and still preserve the effects of the parent transaction before the invocation of the asynchronous method (i.e. partial rollback).

The relation between transactions that run asynchronous methods and transactions that run the continuations can be represented by a tree structure, called transactional tree. Figure 3.3 shows an example of this relation. When the begin method is invoked and there is no transaction active, a top-level transaction is created ($T_0$). This transaction represents the root of a new transactional tree and upon the invocation of an asynchronous method, two new child transactions are created ($F_1$ and $C_1$). The asynchronous method will be executed by a new thread and will run in the context of one of those child transactions ($F_1$), while the continuation will be executed by the same thread of its parent, but in the context of the other child transaction ($C_2$).

In this tree, the relation between child and parent transactions, is the same as the relation of transactions that compose a transactional tree of nested transactions. This means that child
transactions, can read the writes performed by their ancestors, however transactions cannot read writes performed by their active siblings. In this model, the only conflicts that can break the sequential semantic of the top-level transaction’s code (Section 1.2.3) are WAR conflicts. More precisely, this conflict can occur between transactions running asynchronous methods ($F_2$ can conflict with $F_1$, and $F_3$ can conflict with $F_2$ and $F_1$), and between sibling transactions ($C_1$ can conflict with $F_1$, and $C_2$ can conflict with $F_2$).

Whenever a transaction finishes its execution it must then check for conflicts. However, there is a sequential dependence between transactions running asynchronous methods and continuations. Because of this dependency, the only way for these transactions to know they do not conflict with other transactions that precedes them in the sequential order, is to wait for them to validate and commit first. In practice, this means that when a transaction running an asynchronous method or a continuation finishes execution, and before it validates, it must wait that all other transactions that precedes it in the sequential order have validated and committed.

When these transactions finally reach their turn to commit, they must then check if there is an intersection between their reads and the writes of the transactions (in the same tree) that have committed while the transaction attempting to commit was executing. In that case, it means that a conflict that broke the sequential semantic occurred, and the transaction must now re-execute.

The commit procedure of transactions running asynchronous methods and continuations ensures that the writes the transaction performed are passed to the parent transaction. From that point on, those writes can be seen by new child transactions the parent might spawn. This process allows transactions that re-execute, due to a sequential conflict, to read the writes they missed on their previous execution. For example, assume the transactional tree in Figure 3.3. Assume that $C_1$ writes to a transactional object and then spawns $F_2$ and $C_2$. Then, assume that $F_2$ writes to that same transactional object and afterwards $C_2$ tries to read it. $C_2$ can only read the writes performed by its ancestors and so it will miss $F_2$’s write, causing a conflict that break the sequential semantic of $T_0$’s code. When $C_2$ finishes and validates (after $F_2$ had already committed), it will detect this conflict and will have to re-execute. In the re-execution the write performed by $F_2$ already belongs to $C_1$ which now can be seen by $C_2$.

All the execution of child transactions running asynchronous methods and continuations is managed by the JTF runtime. This runtime, ensures that the concurrent execution of these transactions respects the sequential semantic inside the top-level transaction’s code. Once all transactions in the transactional tree have committed, the control is passed to the JVSTM. At that point, JVSTM will finish the execution of the top-level transaction, by validating it against other top-level transactions in the system and committing it.
3.3. ALGORITHM

Listing 3.2: Example of invocation of two Transactional Futures

```java
public void method(..){
    Transaction.begin();
    try{
        Future future1 = Transaction.manageParallelTask(asynchronousMethod1);
        // creation of transaction F1 and C1
        Future future2 = Transaction.manageParallelTask(asynchronousMethod2);
        // creation of transaction F2 and C2
        Transaction.commit();
    }catch(CommitException ce){
        Transaction.abort();
    }
}
```

3.3.1 Metadata

3.3.1.1 Transaction Metadata

In JVSTM all top-level transactions are associated with a version number, which is assigned when the transaction is created. This number is fetched from a global counter that represents the version number of the latest read-write transaction that successfully committed. Transactions running asynchronous methods or continuations also get a version number which they inherit from the top-level transaction in which they were invoked.

In order to support the invocation of asynchronous methods inside transactions, we need to associate additional metadata to transactions. As already mentioned, we need to preserve the sequential semantics of the top-level transaction’s code. This dependency forces transactions running asynchronous methods and transactions running continuations, inside a top-level transaction, to validate and commit according their sequential order of appearance. To ensure this order, we associate a sequential identifier (seqID) to every transaction. This identifier represents their order of creation/appearance inside the top-level transaction’s code. Thus, transactions commit according the ascending order of seqID. Take as an example Figure 3.4, illustrates how these identifiers are attributed to all the transactions running inside method() of Figure 3.2. In this transactional tree the order in which transactions must commit is F1, F2, F3, C3, C2, C1 and finally T0.

Figure 3.5 shows three other important fields kept in each transaction: nClock, seqClock and ancVer map:

- the nClock is an integer that is incremented by the commit of each child;
Figure 3.4: Example of how seqIDs (represented by red numbers) are attributed to all transactions created inside the method of Listing 3.2.

Figure 3.5: Example of how transaction metadata is managed in JTF. Transactions $C2$ and $F2$ were spawned after the commit of $F1$. Thus, at the time of their creation, they save $T0$’s $nClock$ with the value 1 in their $ancVer$ map.

Figure 3.6: VBox structure.

- the seqClock is an integer that represents the seqID of the last child that has committed and takes the value 0 when no child has committed yet;

- finally, the $ancVer$ is a map containing a copy of the $nClock$ field of each ancestor of the transaction. Each $nClock$ as the exact values they had when the transaction started. This map represents the versions of the ancestor’s writes that the child transaction can read.

3.3.1.2 Object Metadata

Just like any other transaction in JVSTM (Section 2.1.11.3), transactions running asynchronous methods and continuations use VBoxes to buffer and fetch the transactional data values. As depicted in Figure 3.6, VBoxes contain two lists of writes: one list of values written by committed transactions (permanent write list) and another of values written by running transactions.
3.3. ALGORITHM

Listing 3.3: Read procedure pseudo-code, used by transactions running asynchronous methods or continuations

```java
public read(vbox){
    tentativeWrite = vbox.tentativeWriteList[0];
    OwnershipRecord ownerOrec = tentativeWrite.orec;
    if (ownerOrec.status != RUNNING && ownerOrec.version <= this.version) {
        return readGlobal(vbox);
    }

    while(true){
        if(ownerOrec.owner == this && ownerOrec.status != ABORTED){
            return tentativeWrite.value;
        }
        else if(ancVer.contains(ownerOrec.owner) &&
                ownerOrec.txTreeVer <= ancVer.get(ownerOrec.owner)){
            nestedReads.put(vbox);// the transaction's read-set
            return tentativeWrite.value;
        }
        tentativeWrite = getNextTentativeWrite();
        if(tentativeWrite == null)
            break;
        ownerOrec = tentativeWrite.orec;
    }
    if(topLevelTx.writeSet.contains(vbox))
        return topLevelTx.writeSet.get(vbox);
    return readGlobal(vbox);
}
```

(tentative write list). Additionally, each tentative write points to an ownership record (orec), which contains information about the owner of the write (a transaction), the version of the write (txTreeVer) and the status of the owner (running, committed or aborted). Every transaction as an orec of its own, which becomes associated with every new tentative write they create. The txTreeVer field of each transaction orec starts with the value 0 when the transaction is created.

3.3.2 Transactional procedures

3.3.2.1 Read procedure

Unlike top-level transactions, transactions running asynchronous methods or continuations do not own a write-set. Instead they buffer their writes inside VBoxes, more precisely inside the tentative write list. Listing 3.3 shows the pseudo-code of the read procedure used by these transactions.

When reading a VBox, transactions need to take into account a possible read-after-write situation. This corresponds to the situation when the transaction attempting the read or one
of its ancestors has previously written to the VBox. We can be sure there is no read-after-write situation when the last tentative write was made by a top-level transaction that finished (committed or aborted) before this one started. This case corresponds to the code between lines 4-6 and a permanent value is returned (committed by a top-level transaction).

The reason why the transaction reads a permanent value and not the tentative write, is because it needs to make sure the value was not committed after the root transaction, of the tree which the transaction attempting the read belongs to, started. This is done by checking the version of the permanent value. If that version is higher than the transaction’s version, the whole transactional tree is aborted. Otherwise, the value is read. When reading from the permanent write list, the transaction uses the version number that it inherited from its top-level transaction in order to find the correct version of the value to read.

However, if this path is not used, then the algorithm iterates (line 8) over the tentative writes of the VBox until one of the following conditions is verified:

- The transaction attempting the read (T) is the owner of the tentative write. In this case, the transaction also checks if the write does not belong to a previous aborted execution (line 9). This previous execution corresponds to the case in which the transaction failed validation due to a detected WAR conflict that broke the sequential semantic of the top-level transaction’s code, forcing re-execution. If the write was performed in the transaction’s current execution, then no further checks are needed and the procedure returns that value (lines 11).

- The owner of the tentative write is an ancestor of (T). When this happens, T may read that entry only if the entry was made visible by its owner before T started (lines 12-15). This is enforced by looking up in the ancVer what is the maximum version of the ancestor’s write the transaction can read and comparing it with the version of the tentative write (txTreeVer).

If no valid value was found in the tentative write list, then we can be sure there is no read-after-write situation. Therefore, the transaction attempting the read either fetches the value from the top-level transaction’s write-set (lines 21-22), if the latter has written to the VBox, or it fetches a permanent value (line 24). Additionally, whenever a transaction reads an ancestor’s write or a permanent write, it also inserts that write in its local read-set (line 13).

One could think that a better solution would be to fetch the value written by the transaction with the highest seqID, but lower than the seqID of the transaction attempting the read. Taking the transactional tree in Figure 3.4 as example, this means that if F1 was the only transaction writing to the VBox that C2 is attempting to read, the latter could read the write performed by the former. This is what it would happen if the top-level transaction’s code ran sequentially. The problem with this solution is that we can never be sure if the owner of that write will not
try to write again and therefore invalidate the read. Furthermore, another transaction with a higher seqID than F1 can write to the VBox, for example transaction F2, invalidating the read performed by C2. The only possible solution would be to force C2 to block until the values written by F1 or F2 are available in C2’s ancestors, for simplicity we decided not to do this and let C2 detect the conflict at commit time.

3.3.2.2 Write procedure

Listing 3.4 presents the pseudo-code of the write procedure used by transactions running asynchronous methods or *continuations*. When writing to a VBox, the transaction (T) fetches the tentative write at the head of the tentative write list and reads its ore<sub>c</sub> to tell whether it owns that write or not (lines 2-5). If the transaction owns the write, it simply overwrites the previous write.

Otherwise, after line 8, the algorithm checks if the transaction that owns the write finished before transaction T started, in which case T attempts to acquire ownership of the tentative write at the head of the list (lines 10-18). To do so, T attempts a compare-and-swap (CAS) to change the ownership of the first tentative write. If the CAS fails, it means some other transaction acquired the ownership of the tentative write, in which case T must check if the new owner belongs to a different transactional tree by comparing the roots of the trees. If the owner belongs to a different tree, then no transaction in this tree (and particularly T) will ever be able to write a tentative write. In this case, the transaction uses a fallback mechanism (*ownedbyAnotherTree* method) (lines 20-23) and we say that an *inter-tree conflict* occurred.

After line 23, we can be sure that the owner of the tentative write at the head of the list is a transaction of the same transactional tree of T. In this case, the algorithm iterates over all writes in the list until it finds a place to insert the new tentative write. The tentative write list is organized by a descending order of seqID, where the write in the tail of the list corresponds to the write performed by the transaction with the lowest seqID. The place where the transaction places the write must respect this organization. The insertion is performed with a simple CAS operation over pointers (line 27). If this CAS operation fails, it means another transaction from the same tree of T managed to insert a new tentative write first, in which case T continues iterating the list to find a new place to insert its write.

The organization of the tentative write list by seqID, allows better performance of the read procedure. Let us recall the read procedure pseudo-code in Listing 3.3. A transaction can read any write from any of its ancestors, as long as those writes have a valid version (*txTreeVer*) (lines 12-15). If several of the transaction’s ancestors wrote to the same tentative write list, it means there are several possible writes the transaction can read (as long they have valid versions). However, instead of having to iterate all the list to find all the ancestor’s writes, it returns the first ancestor write found (with a valid version). The read procedure can do this and still be
Listing 3.4: Write procedure pseudo-code, used by transactions running asynchronous methods or continuations

```java
public write(vbox, value){
    tentativeWrite = vbox.tentativeWriteList[0];
    OwnershipRecord ownerOrec = tentativeWrite.orec;

    if (ownerOrec.owner == this) {
        tentativeWrite.value = value;
        return;
    }

    if (ownerOrec.status != RUNNING && ownerOrec.version <= this.version) {
        if (tentativeWrite.CASowner(ownerOrec, this.orec)) {
            tentativeWrite.tempValue = value;
            boxesWritten.add(vbox);
            return;
        }
        tentativeWrite = vbox.tentativeWriteList[0];
        ownerOrec = tentativeWrite.orec;
    }

    if(ownerOrec.owner.treeRoot != this.treeRoot){
        ownedbyAnotherTree(vbox,value);
        return;
    }

    for(tentativeWrite: vbox.tentativeWriteList){
        if(tentativeWrite.owner.seqID < this.seqID){
            tentativeWrite.CASnext(new TentativeWrite(value));
            return;
        }
        if(tentativeWrite.owner.seqID == this.seqID){
            tentativeWrite.value = value;
        }
    }
}
```
Figure 3.7: Example of how a possible inefficiency can occur in the read procedure if tentative writes are not sorted. Both State 1 and 2 represent the tentative write list of the same VBox. The relation between transaction $C2 F2, F1$ and $T0$ is the same as the one in the transactional tree of Figure 3.3.

To illustrate the performance benefits for sorting writes, take the example of the tentative write list states of Figure 3.7 where writes are not sorted in any way. Since the thread running continuations runs concurrently with transactions running asynchronous methods, State 1 is a possible organization of this unsorted list. With the commit of transactions $F2$ and $F1$, the writes performed by these transactions are passed to its ancestors. Therefore, the state of the list changes to the one represented by State 2. Now assume that $C2$, which was running concurrently with $F2$ and $F1$, spawns $F3$ by invoking an asynchronous method. If $F3$ decides to read the VBox, there are three possible valid values for it to read. However, only one is the correct one in respect to the sequential semantic of the top-level transaction’s code, which is the one at the tail of the list.

With writes sorted according the seqID of transactions the correct value would be at the head of the list, making the read procedure of $F3$ faster. Furthermore, this type of sort ensures that when the top-level transaction commits, it can find the values that must be written back to the permanent write list on the head of the tentative write list.

The fallback mechanism (ownedbyAnotherTree method, in line 21) used when a inter-tree conflict is detected, prevents that a high number of transactions write to the same tentative write list. This is done for simplicity and for performance reasons. If we had all transactions running in the system writing to the list, we would be forced to have a more complex management of the tentative write list and a more complex and slower read procedure in order to find the correct value to read.

This fallback mechanism consists of passing control back to the top-level transaction, which will then re-execute the affected asynchronous method (including the conflicting write). The key
difference is that top-level transactions maintain a traditional write-set to use when a tentative write list is controlled by another transactional tree. However, recall that the thread running the continuation is the same thread that was running the top-level transaction before the asynchronous method was invoked. Only this thread can perform the write that triggered the conflict in the context of the top-level transaction.

Figure 3.8 depicts an example of how an inter-tree conflict is resolved. When a transaction running an asynchronous method fails to write to a VBox (F2), due to this type of conflict, it simply flags the parent transaction (C1) and sends it the Callable instance that holds the implementation of the asynchronous method. Once the thread running the continuation (C2) ends, and before it commits, it travels all the transactional tree up to the root and checks if any inter-tree conflict was flagged. If so, instead of committing all the transactional tree, it commits a sub-tree that starts from the root (excluding, because the top-level transaction can only commit at the end of its code execution) and ends on the transaction that experienced the conflict (F2). All the other transactions in the tree (F2 and C2) abort. This process allows us to save valid work performed by asynchronous methods (F1) and continuations (C1) that happened, in respect to the sequential order, before the conflicting write. Once the sub-tree is committed, the thread running the last continuation (C2) uses the first-class continuation support provided by the underlying JVM to jump back to the execution state before the invocation of the asynchronous method where the conflict occurred. This time, instead of invoking that method asynchronously, it invokes it synchronously, executing sequentially both the asynchronous method and the following continuation.
3.3. ALGORITHM

3.3.3 Committing transactions

All transactions commit upon the invocation of the `Transaction.commit` method. The top-level transaction code should always contain the invocation of this method, which should have been inserted by the programmer of the client application. However, asynchronous methods do not need to be instrumented by the programmer. In order to commit transactions running this methods, when the method returns, the control is given back to JVSTM (more precisely the `jvstm.ParallelTask` class) which will then commit the transaction that ran the asynchronous method (recall Figure 3.2).

Listing 3.5 shows the `commit` procedure pseudo-code of transactions running asynchronous methods or `continuations`. When trying to commit, transactions need to validate their read-set in order to detect `WAR` conflicts that may have broken the sequential semantic of the top-level transaction’s code. However, recall that there is a sequential dependence between transactions. Therefore, one transaction can only validate its execution when all other transactions preceding it in program order, have validated and committed. For this reason, when trying to commit, a transaction must first wait that all other child transactions with lower `seqID` (excluding its ancestors) have committed (line 3). Let us recall Figure 3.4. In practice, this means that transactions running asynchronous methods must wait that all other transactions running asynchronous methods, and with lower `seqID`, have committed. This is forced by waiting for the `seqClock` of the transaction’s grandparent to equal the `seqID` of the transaction wanting to commit minus two. However, if the transaction is running a `continuation`, this means that before committing, the transaction must wait for its sibling transaction (a transaction running an asynchronous method) to commit first. This is forced by waiting for the `seqClock` of the transaction’s parent to equal the `seqID` of the transaction wanting to commit minus one.

When a transaction finally reaches its time to commit, it must then check for `WAR` conflicts (line 4). This process consists in, for every write in the transaction’s read-set, iterating over the `VBox’s tentative write list`. If an entry is found belonging to an ancestor, the read is only valid if that entry is the one that it was read. Otherwise, if another ancestor’s write is found, it means there is a newer version of that write. That version was committed by a transaction running an asynchronous method with lower `seqID` after the transaction attempting the commit started. In this situation, the transaction attempting the commit must abort and re-execute, because a `WAR` conflict occurred. More precisely, the transaction attempting the commit must abort, because it did not read the write performed by the transaction running the preceding (in the sequential order) asynchronous method.

Upon failed validation, if the transaction was running an asynchronous method, it simply calls the `abort` method and re-executes the asynchronous method from the beginning. Otherwise, if the transaction was running a `continuation`, it aborts and uses the `first-class continuation` support in order to restore the execution state to the point where the `continuation` started.
Listing 3.5: Commit procedure pseudo-code, used by transactions running asynchronous methods or continuations

```java
public commit(){
    try{
        waitTurn();
        validate(readset);
    }catch(CommitException e){
        this.abort();
        if(speculationCheckpoint %2 == 0){
            Continuation.resume(startOfContinuation);
        }
        this.abort();
        reexecuteCallableMethod();
    }
    this.orec.txTreeVer = ancVer(getParentTransaction()) + 1;
    this.orec.owner = getParentTransaction();
    for (childTransaction childrenCommit : childrenToPropagate) {
        childrenCommit.orec.txTreeVer = commitNumber;
        childrenCommit.orec.owner = parent;
    }
}
```

The transaction can finally commit if it passes validation (line 13). The key idea is that a child transaction propagates to its parent only the orecs that it controls, which means its own orecs (lines 13-14) and the orecs that belonged to its child transactions (lines 16-19). The propagation is done through a simple change of the owner field of each orec. This also entails updating the txTreeVer of those orecs to the version acquired from the nClock of the parent plus one. As a result, the commit procedure performs independently of the write-set size and is very lightweight in practice.

### 3.3.4 Aborting transactions

When aborting, transactions must revert the writes they performed when their write is at the head of the tentative write list. Recall that, when a transaction is performing a write (Section 3.3.2.2), it checks if the owner of the write at the head of the list has already finished (line 10 of Listing 3.4), in which case the transaction writes a new tentative write. From that point on, only transactions from the same transactional tree of the transaction attempting the write can write to the list.

Take as an example the state of the tentative write list of Figure 3.9. After the abort of C2, if some concurrent transaction from another transactional tree attempts to write to the list, it will find that an aborted transaction owns the write at the head of the list. Therefore, it will be able to write a new tentative write and take control of the list for the transactions running on
3.3. ALGORITHM

Listing 3.6: Abort procedure pseudo-code, used by transactions running asynchronous methods or continuations

```java
public abort()
{
    waitTurn();
    for (VBox vbox : boxesWritten) {
        tentativeWrite = vbox.tentativeWriteList[0];
        if (tentativeWrite.orec.owner == this) {
            revertOverwrite(vbox);
        }
    }
    this.orec.version = OwnershipRecord.ABORTED;
    for (childTransaction : childrenTransactions) {
        childTransaction.orec.version = ABORTED;
    }
    return;
}
```

Figure 3.9: Example of a tentative write list state that requires removal of overwrites. The relation between transaction T0 C1 and C2 is the same as the one in the transactional tree of Figure 3.3.

Figure 3.10: Example of how transactions revert their overwrite. Transaction T0 is the top-level transaction of C1.

that tree. However, the transactional tree of C2 was still supposed to control the list, because there was still writes in the list belonging to the ancestor’s of C2. Thus, when aborting, the transaction must check, for every VBox that it wrote, if it owns the write at the head of the tentative write list. If it does, the transaction must delete its write from the tentative write list (lines 4-9). To revert the write (Figure 3.10), the transaction sets the owner and the value of its write to the owner and the value of the ancestor’s write and makes it point to the write that follows the ancestor’s write in the list.

In order to make this operation lock-free, we need to make sure that no transaction changes any of the writes between the write at the tail of the list and the write of the transaction reverting its writes. To ensure this, a transaction only aborts when all other transactions from the same transactional tree (excluding its ancestors) and with a lower seqID have finished executing (either aborted due to an inter-tree conflict or committed) (line 3). This is done in the same way of how transactions wait for their time to commit, by checking the seqClock of their ancestors.

Finally, the transaction finishes the abort procedure by changing the status of the oreces it controls (its own ore and its children transactions ore) to abort (lines 10-13).
3.4 Optimizing read-only transactions

JVSTM implements the notion of multi-versions. This property allows read-only transactions to never conflict with any other concurrent transaction. Because of this property, this type of transactions do not need to validate themselves, allowing them to immediately commit when they finish executing. This feature allows JVSTM to extract good results in applications with a high read/write ratio.

However, even if some Transactional Futures are marked as read-only, we cannot be sure that the transactions running the corresponding read-only asynchronous method can skip validation. A transaction running the read-only asynchronous method needs to validate to ensure that it did not miss the write performed by a preceding (in the sequential order) transaction running a read-write asynchronous method.

Yet, instead of validating every transaction executing a read-only asynchronous method, we decided to, before the creation of the transaction, check if the transaction can effectively run as read-only and skip validation at the end of its execution. A read-only transaction running an asynchronous method can skip validation if all other transactions with lower seqID have already committed before this read-only transaction started, or if all of those transactions are read-only as well.

In order to support this optimization, we added a new list in every top-level transaction, which contains the seqID of every read-write transaction spawned inside the top-level transaction’s code. Whenever a transaction running an asynchronous method, and marked as read-only, is to be spawned, it first checks if there is any read-write transaction in this list. If the list is empty, then the transaction will skip validation at the end of execution. Otherwise, for every read-write transaction inside the list, the read-only transaction must check if they have already committed. This is done by traversing the transactional tree and checking the seqClock value of the corresponding ancestors.

As any other read-only transaction in JVSTM, if transactions running asynchronous methods are to be executed as read-only transactions, they need to be explicitly marked as read-only by the programmer. We managed to do this with a minimal change in the interface, overloading the method Transaction.manageParallelTask(Callable) with Transaction.manageParallelTask(Callable c, Boolean read-only).

We decided to apply this optimization only to transactions running asynchronous methods, because the creation of transactions running continuations is transparent to the programmer and, in order for the programmer to mark continuations as read-only, we would need to break this abstraction and add new interfaces to the API.
Chapter 4

Experimenting Results

In this Chapter we evaluate the performance of JTF runtime. In Section 4.1 we describe the settings used to evaluate the system. More precisely, we introduce the benchmarks used and the platform used to run them. Section (4.2) finishes the chapter by presenting and discussing the results of the benchmarks.

4.1 Experimental Settings

Two benchmarks were used to evaluate JTF: a modified version of the Vacation benchmark and a Red-Black Tree benchmark.

4.1.1 Vacation benchmark

The Vacation benchmark from the STAMP suite implements a travel agency. The system maintains a database implemented as a set of tree structures. This database is used to store the identification of clients and their reservations for various travel items.

A single client initiates a session in which a set of operations are issued. The benchmark measures how long it takes to process a given session.

There are three different operations that the client can issue within a session. Furthermore, each operation is considered to be an atomic action. In the same session an operation can be issued multiple times on (possibly) different parts of the system’s database objects.

In this modified version of the benchmark, the cycle that performs the operations that compose the client’s session was parallelized, allowing the operation’s to run concurrently. In order to preserve the atomicity of the operations, each operation is executed in the context of a transaction.
In our evaluation, we parallelized this cycle in three different ways:

C.1 by parallelizing the operations between top-level transactions;

C.2 by parallelizing the operations between top-level transactions, each further parallelized with parallel nested transactions;

C.3 and finally, by parallelizing the operations between top-level transactions, each further parallelized with Transactional Futures.

For each of these conditions, we measure the time it takes to process all the operations that compose the client’s session. In all executions the total number of operations that compose the client session is the same.

In all conditions the overall number of threads used is the same. For example, for condition C.1, when we parallelize operations using 8 top-level transactions, in condition C.2 and C.3 we use 1 top-level transaction to perform all operations, but with 8 inner parallel transactions/Transaction Futures.

In all conditions, whenever a top-level transaction aborts, the affected operation is re-executed, forcing the re-execution of the top-level transaction. This means that the total amount of committed top-level transactions/operations is the same in all executions.

The benchmark allows parametrizing the level of contention for the objects of the graph. In our evaluation we consider two scenarios: High contention, which uses 1% of the graph of objects; and low contention, which uses 90% of the graph of objects.

4.1.2 Red-Black Tree benchmark

In this benchmark, we simulate a server that maintains a database and serves requests from local client processes. The database consist in a Red-Black Tree structure containing 1.000.000 integers between the interval [0-2.000.000]. Each request comes with a value and for each request the server starts a top-level transaction that searches which integers between the interval [value - 100.000, value + 100.000] exists in the database. Furthermore, each time the transaction searches a value of the interval, it also calculates a probability of performing a write on the tree. This write consists in either removing the value, if the value was found, or adding it to the database, if it was not found. For the following experiments, all requests contain the same value (500.000). This means that it is very likely that two transactions will update at least one same item. Also the likelihood of contention between 2 concurrent transactions is very high, since they all access an interval around the very same number.

We measure how much time it takes for the server to compute different number of concurrent requests (1,2,4,8), with different write probabilities (0,1%;1,0%;10%) and in different conditions:
4.2. RESULTS

C.1 when those requests are computed inside one top-level transaction each, i.e. without any type of inner-parallelism;

C.2 when those requests are computed inside one top-level transaction each, but each transaction is further parallelized with parallel nested transactions;

C.3 finally, when those requests are computed inside one top-level transaction each, but each transaction is further parallelized with Transactional Futures.

For conditions C.2 and C.3, the workload of each top-level transaction (200,000 values to search) is divided in equal parts between its child parallel nested transactions/Transactional Futures. We also increase the number of parallel nested transactions/Transactional Futures that parallelize each top-level transaction and measure how that influences the time to complete all requests. A key difference between this benchmark and the Vacation benchmark is that, in this benchmark, we are not sharing the total workload of the benchmark among different number threads, as we were on Vacation. In this benchmark, as we increase the number of top-level transactions we also multiply the total workload (number of requests) by the same amount of top-level transactions used.

4.1.3 Platform

The results presented in the next section were obtained on a machine with four AMD Opteron 6272 processors (64 cores total) with 32GB of RAM. Every experiment reports the average of five runs of each benchmark.

4.2 Results

4.2.1 Vacation

The Vacation benchmark of the STAMP suite represents a scenario where, under high contention, it becomes increasingly hard to obtain improvements in terms of performance by adding more threads. Figure 4.1 (a) shows evidence of this difficulty, where we may see that the increasing number of top-level transactions only yields modest sub-linear scale ups. This results are expected, since the abort rate of transactions increases with the increasing number of top-level transactions used (see Figure 4.1 (b).

However, we can decrease the abort rate by running fewer top-level transactions. Furthermore, in order to maintain high levels of parallelism, we can parallelize each top-level transaction with Transactional Futures. In this approach, we can run fewer top-level transactions at a time with each one spawning an increasing number of Transactional Futures. With this approach...
CHAPTER 4. EXPERIMENTING RESULTS

(a) Speed-ups in high contention.

(b) Abort rate of top-level transactions in high contention.

Figure 4.1: Speedups (a) and abort rate (b) of using top-level transactions parallelized with parallel nesting or Transactional Futures relative to the execution of using top-level transactions with no inner-parallelization. The threads used are shown as the number of top-level transactions and number of parallel transactions/Transactional Futures each execution spawns. In the approach of using only top-level transactions, the number of top-level transactions used is the multiplication of those two numbers, so that the overall number of threads used is the same in all approaches.

We are able to decrease the abort rate and obtain better results, with up to 4.6 times better performance than top-level transactions.

We can also see some differences between the speed-ups obtained by using parallel nested transactions or Transactional Futures to parallelize the top-level transactions. We did not experience any abort rate of parallel nested transactions or Transactional Futures inside top-level transactions. The reason for this, is because most top-level transactions spawn only read-only parallel nested transactions or read-only Transactional Futures inside them. Furthermore, read-only Transactional Futures and read-only parallel nested transactions have very similar read and commit procedures, which also does not explain the differences in performance. The only significant difference is that, in the context of Transactional Futures, whenever an asynchronous method is invoked, the following continuation must capture the execution state of the current thread with the first-class continuation support. Recall from Section 3.3.3 that continuations must revert to this state whenever they fail validation. The higher the number of Transactional Futures used inside a top-level transaction, the higher is the number of execution states that must be captured. This could explain why the difference in speed-ups is higher when we spawn a higher number of parallel nested transaction/Transactional Futures inside top-level transactions. We believe the differences in the speed-ups come from the overhead of using the first-class continuation support. However, we do not have objective data to support this statement.
4.2. RESULTS

(a) Speed-ups in low contention.  
(b) Abort-rate in low contention.

Figure 4.2: Speedups (a) and abort-rate (b) of using top-level transactions parallelized with parallel nesting or Transactional Futures relative to the execution of using top-level transactions with no inner-parallelization. The threads used are shown as the number of top-level transactions and number of parallel transactions/Transactional Futures each execution spawns. In the approach of using only top-level transactions, the number of top-level transactions used is the multiplication of those two numbers, so that the overall number of threads used is the same in all approaches.

On the other hand, Figure 4.2 exemplifies a workload with low contention. In this case, the top-level transactions approach is already achieving reasonable performance as the thread count increases. Thus, the alternative of applying parallelization inside transactions and run fewer top-level transactions does not yield any extra performance. As a matter of fact, we may actually see that there is some overhead from executing the transactions with Transactional Futures, since we get worse speed-ups with this approach. However, we can see that after a certain threshold the number of top-level of transactions starts to increase drastically the abort rate of transactions, which also affects the performance of the benchmark. After this threshold, the alternative of parallelize top-level transactions with Transactional Futures and run fewer top-level transactions starts to achieve better performance.

Unlike the high contention execution, in a low contention execution, for a high number of parallel nested transactions and Transactional Futures spawned, the two approaches achieve similar speed-ups. We believe the reason for this difference is because, since there are lower re-execution of top-level transactions, due to a lower abort rate, there is also less Transactional Futures to be spawned. With less Transactional Futures being spawn, the lower is the number of states being captured. For this reason the overall overhead of using first-class continuations has less impact in the overall performance of the benchmark.

Across these experiments we can see that added benefit is obtained by exploiting both the inter- and the intra-parallelism of transactions. This supports the idea of using STM and
Figure 4.3: Speedups of using top-level transactions parallelized with parallel nesting or Transactional Futures relative to the execution of using top-level transactions with no inner-parallelization.

Transactional Futures combined in order to obtain higher levels of parallelism and performance than using each one individually.

4.3 Red-Black Tree

Figure 4.3 shows the speed-ups obtained with C.2 and C.3 relative to the execution with condition C.1. We measured a high abort rate of child transactions when parallelizing requests (top-level transactions) with parallel nested transactions, even in the presence of no contention (one top-level transaction). These aborts come from the write-write contention when writing to tentative write lists of VBoxes. Recall from Section 2.1.11.3 that a parallel nested transaction
4.3. RED-BLACK TREE

acquires ownership of the list when it writes to it. As long as the transaction has ownership of the list, only this transaction and its descendants can write new tentative values. The affected nested transactions that find this list locked, must be re-executed sequentially by the top-level transaction. This explains why parallelizing top-level transactions with parallel nested transactions does not yield any extra performance in this benchmark. In practice, by parallelizing a top-level transaction with \( x \) nested transactions, \( x-1 \) of those transactions end up being executed sequentially. This happens because they all tried to write to at least one same VBox.

JTF runtime also uses a similarly lock, but instead of being local to the transaction and its descendants, it is local to the whole transactional tree. This means that, unlike parallel nested transactions, there is no write-write contention between Transactional Futures of the same tree. Because of this we were able to experience a lower abort rate of Transactional Futures and extract better speed-ups. However, with the increase of concurrent requests, and therefore the increase of concurrent top-level transactions, so does the \textit{inter-tree conflicts} (recall Section 3.3.2.2) between Transactional Futures increase. Because of this the higher the number of concurrent top-level transactions running, the lower are the benefits of running Transactional Futures inside each transaction (Graphic D).

We also experience higher abort rates of Transactional Futures with the increase of the write probability. The higher the number of writes, the higher is the probability of two Transactional Futures experience a WAR (recall Section 3.3.3) conflict that breaks the sequential semantic of the top-level transaction. This forces a higher number of Transactional Futures to re-execute which will degrade the performance of the system. For this reason, when executing the benchmark with a write probability of 1% and 10% we experienced lower speed-ups than the ones obtained for 0.1% probability.
Chapter 5

Conclusions

The increasing core count in modern devices allow software companies to explore complex applications that require powerful processing which traditional single-core computers cannot offer. In business computing, the ability to extract parallelism from applications becomes a competitive advantage, as it allows those applications to perform faster. However, parallel programming is far from trivial, which makes it hard for software developers to take advantage of this increasing computational power.

From the beginning of this dissertation, we defended that a combination of two of the most prominent examples of fork-join multi-threaded programming paradigms (STM and Futures), could extract higher levels of parallelization from applications than by just using one individually. Furthermore, we believed that this combination could be done without breaking the abstractions that both systems provide over complex concurrency issues. Examples of such issues are thread creation, scheduling, joining and return value synchronization, as well as synchronization on concurrent accesses over shared data.

However, we showed such combination requires great care, as it is not trivial to design a system that can effectively cope the two mechanisms without endangering correctness. We have addressed the inherent problems of such combination and proposed a runtime middleware that combines STM and STLS strategies in order to allow this promising combination. The proposed solution manages to do so with minimal changes to the interface of both systems, preserving the abstractions they provide.

We evaluated our runtime middleware and showed that combining Futures in STM transactions could effectively extract higher performance benefits than just using STM transactions to parallelize applications. Furthermore, has we have showed in our evaluation, different degrees of inter- and the intra-parallelism of transactions influence the performance one can obtain with the combination of these two systems. This is evidence that it is necessary to adapt the degree of parallelism to the data contention level of applications. This is not surprising, as it is also the case for traditional STM systems (Didona, Felber, Hermanci, Romano, & Schenker 2013).
However, this tuning problem becomes much more complex in a system that combines STM and Futures, as one needs to identify the correct setting of the number of top-level and futures transactions. We believe this middleware has showed enough evidences that this combination of systems is a viable option to be further explored.
Bibliography


67


