Transactional Java Futures

José Pereira

Instituto Superior Técnico, Lisboa, Portugal
jose.m.pereira@ist.utl.pt

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 article proposes an unified system that supports the combination of both models, STM and Futures. In this article 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.

Keywords: Software Transactional Memory, Thread-Level Speculation, Java Futures, Java Versioned Transactional Memory

1 Introduction

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 [1, 2, 3].

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 [4, 5]. 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 [4, 5].

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 [6, 7], 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 [8, 9, 10].

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. 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.

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 [5].

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 [11, 12]. 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.

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. 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 article is organized as follows. Chapter 2 gives a description about the different design options that distinguishes STM systems, it also introduces Thread-Level Speculation (TLS) relating it to Futures. In chap-

ter 3 we present the JTF runtime middleware, a system that effectively combines a state-of the-art STM with Java Futures by addressing the inherent problems of such combination. In Chapter 4 we describe the experimental results performed on this system, comparing its performance to the baseline STM. Chapter 5 ends the document with a brief conclusion over all the work described in this article.

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.

Researchers have tried to come up with new programming paradigms that enhance parallel programming with mechanisms for abstraction and composition, which are crucial for managing complexity. Transactional Memory (TM) and Thread-Level Speculation (TLS) are two prominent examples of such.

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 [5].

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.

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.

Optimistic versus pessimistic concurrency control TM systems [13] 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 [14] 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.

Correctness criteria There are several correctness conditions for concurrent transactions that TM systems rely on [5]:

- *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 [5] 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.

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 [13], additional mechanisms have to be implemented in order to support opacity, 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 [13] 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 [15] 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.

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 [16]. 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 [5] and so, when commits are common, flattened nesting might be a better approach.
– *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.

JVSTM Several systems have been proposed to allow the introduction of STM into the Java environment [16, 17]. Java Versioned Software Transactional Memory (JVSTM) [15] 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.

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*.

The JVSTM introduces the concept of multi-version (Section 2.1) in the Java environment. 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.

2.2 Thread-Level Speculation

Thread-level speculation (TLS) allows 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. TLS ensures that the program executed the same way that it did originally. The correctness criterium 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. 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.

2.3 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) [18] lacks concurrency control between the asynchronous work going on in different future tasks.

3 Java Transactional Futures runtime system

In this article we present an unified runtime middleware for Java, called *Java Transactional Futures* (**JTF**), that unifies STM and Java Futures. Asynchronous methods invoked with Futures inside transactions, are managed by the JTF runtime system which addresses the inherent problems of combining STM and Futures. 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 futures is equivalent to the result of executing those methods sequentially. 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.

3.1 Algorithm

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. With Transactional Futures, once an asynchronous method is invoked, a new 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).

Whenever a transaction finishes its execution it must then check for conflicts that may have broken the sequential semantics of the top-level transaction's code. 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.

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.

3.2 Metadata

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 it 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

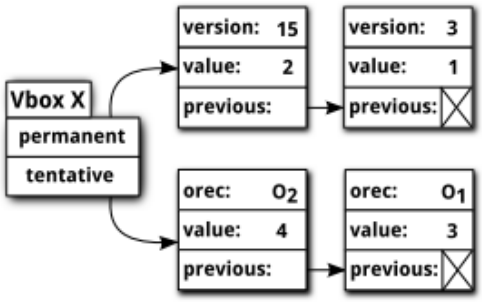

Fig. 1. VBox structure.

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*.

There are 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;
- 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.

Object Metadata Just like any other transaction in JVSTM, transactions running asynchronous methods and continuations use *VBoxes* to buffer and fetch the transactional data values. As depicted in Figure 1, *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 (*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 Transactional procedures

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*.

Read procedure 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.

However, if this path is not used, the reasding transaction iterates 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. 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.
- 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. 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, if the latter has written to the *VBox*, or it fetches a permanent value.

Write procedure When writing to a *VBox*, the transaction 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 organization of the *tentative write list* by *seqID*, allows better performance of the read procedure.

4 Experimental results

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. We evaluated JTF with two benchmark: the Vacation benchmark and a Red-Black Tree benchmark.

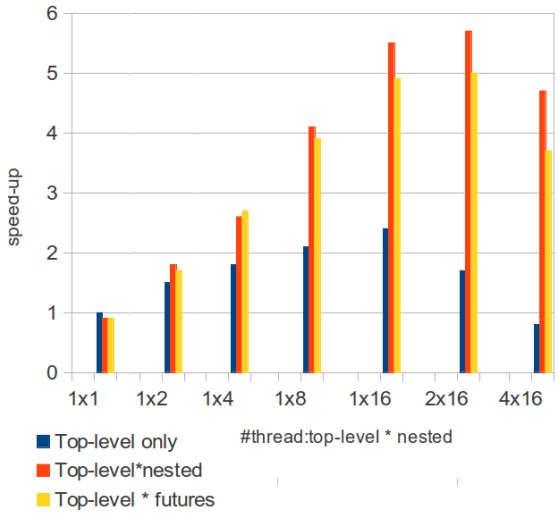

Fig. 2. Speedups of using top-level transactions parallelized with parallel nesting or Transactional Futures relative to the execution of using top-level transactions with no inner-parallelization. The threads used are shown as the number of top-level transactions and number of parallel transactions/Transactional 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.

4.1 Vacation Benchmark

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 2 shows evidence of this difficulty, where we may see that share the total workload among different number of top-level transactions does not scale properly. This results are expected, since the abort rate of transactions increases with increasing number of top-level transactions used.

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 we are able to obtain better results, with up to 4,6 times better performance than top-level transactions.

On the other hand, Figure 3 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

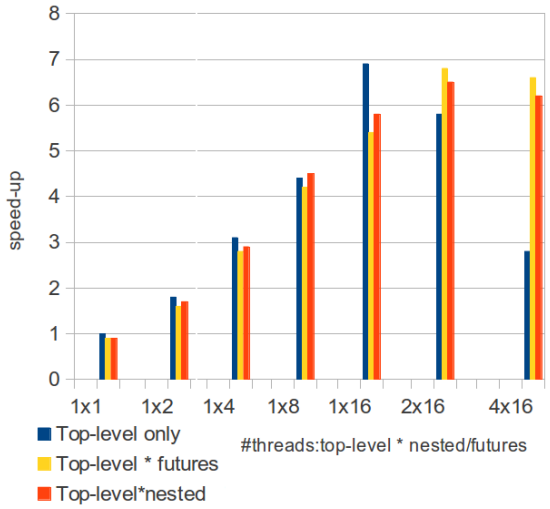

Fig. 3. Speedups of using top-level transactions parallelized with parallel nesting or Transactional Futures relative to the execution of using top-level transactions with no inner-parallelization. The threads used are shown as the number of top-level transactions and number of parallel transactions/Transactional 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.

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.

4.2 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

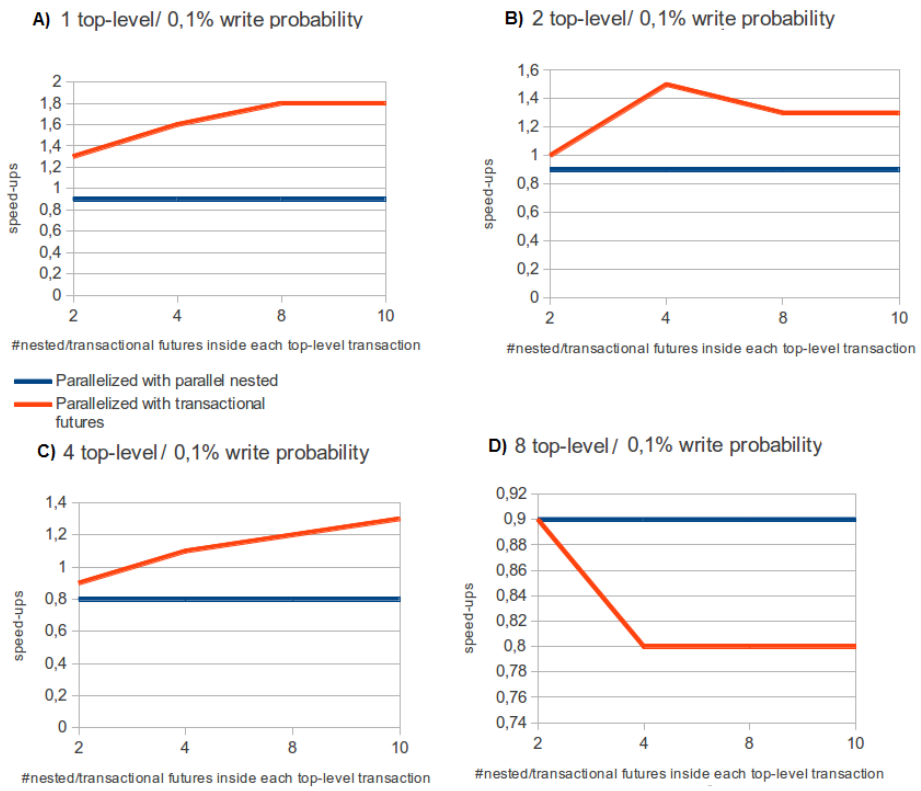

Fig. 4. Speedups of using top-level transactions parallelized with parallel nesting or Transactional Futures relative to the execution of using top-level transactions with no inner-parallelization.

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:

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.

Figure 4 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 between sibling transactions. 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*.

Unlike parallel nested transactions, there is no write-write contention between Transactional Futures of the same transactional tree. Because of this we were able to experience a lower abort rate of Transactional Futures and extract better speed-ups.

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 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 1% probability.

5 Conclusion

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. We believe this middleware has showed enough evidences that this combination of systems is a viable option to be further explored.

Bibliography

[1] Rajkumar Buyya. The design of paras microkernel. `http://www.buyya.com/microkernel/`, June 2000. Accessed December 3, 2013.

[2] Tom Spyrou. Why parallel processing? Why now? What about my legacy code? `http://software.intel.com/en-us/blogs/2009/08/31/why-parallel-processing-why-now-what-about-my-legacy-code`, August 2009. Accessed December 3, 2013.

[3] Jason Howard, Saurabh Dighe, Yatin Hoskote, Sriram R. Vangal, David Finan, Gregory Ruhl, David Jenkins, Howard Wilson, Nitin Borkar, Gerhard Schrom, Fabric Pailet, Shailendra Jain, Tiju Jacob, Satish Yada, Sraven Marella, Praveen Salihundam, Vasantha Erraguntla, Michael Konow, Michael Riepen, Guido Droege, Joerg Lindemann, Matthias Gries, Thomas Apel, Kersten Henriss, Tor Lund-Larsen, Sebastian Steibl, Shekhar Borkar, Vivek De, Rob F. Van der Wijngaart, and Timothy G. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *ISSCC*, pages 108–109. IEEE, 2010.

[4] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education Group, 2003.

[5] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[6] Oracle Corporation. Thread (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html/`, 2013. Accessed December 18, 2013.

[7] Oracle Corporation. Concurrency utilities. `http://docs.oracle.com/javase/1.5.0/docs/guide/concurrency/`, November 2010. Accessed December 18, 2013.

[8] Oracle Corporation. Future (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html`, 2013. Accessed December 18, 2013.

[9] Oracle Corporation. Executor (Java Platform SE 7). `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Executor.html`, 2013. Accessed December 18, 2013.

[10] Adam Welc, Suresh Jagannathan, and Antony L. Hosking. Safe futures for java. pages 439–453. ACM, 2005.

[11] Ferad Zyulkyarov, Vladimir Gajinov, Osman S. Unsal, Adrián Cristal, Eduard Ayguadé, Tim Harris, and Mateo Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 25–34, New York, NY, USA, 2009.

[12] João Barreto, Aleksandar Dragojevic, Paulo Ferreira, Ricardo Filipe, and Rachid Guerraoui. Unifying thread-level speculation and transactional memory. In Priya Narasimhan and Peter Triantafillou, editors, *Middleware*

2012, volume 7662 of *Lecture Notes in Computer Science*, pages 187–207. Springer Berlin Heidelberg, 2012.

[13] Pascal Felber, Christof Fetzer, and Torvald Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 237–246. ACM, 2008.

[14] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. *SIGPLAN Not.*, 44(6):155–165, June 2009.

[15] J. Cachopo. Jvstm - java versioned software transactional memory. `http://inesc-id-esw.github.io/jvstm/`, 2008. Accessed December 18, 2013.

[16] Benjamin Hindman and Dan Grossman. Atomicity via source-to-source translation. In *Proceedings of the 2006 Workshop on Memory System Performance and Correctness*, pages 82–91. ACM, 2006.

[17] G. Korland Felber, P. and N. Shavit. Deuce: Noninvasive concurrency with a java stm. In *Electronic Proceedings of the workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, pages 5–18, 2010.

[18] Oracle Corporation. Java^{TM} platform, standard edition 6 api specification. `http://docs.oracle.com/javase/6/docs/api/overview-summary.html`, November 2011. Accessed December 18, 2013.