ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/308567467

The Future(s) of Transactional Memory

Conference Paper · August 2016

DOI: 10.1109/ICPP.2016.57

CITATIONS
2

READS
106

5 authors, including:

João Barreto
Technical University of Lisbon
40 PUBLICATIONS **120** CITATIONS

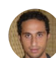
SEE PROFILE

Seif Haridi
KTH Royal Institute of Technology
227 PUBLICATIONS **4,088** CITATIONS

SEE PROFILE

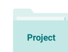
Luís Rodrigues
Technical University of Lisbon
352 PUBLICATIONS **4,223** CITATIONS

SEE PROFILE


Paolo Romano
University of Lisbon and INESC-ID, Lisbon, Portugal
155 PUBLICATIONS **1,357** CITATIONS

SEE PROFILE

Some of the authors of this publication are also working on these related projects:


DEAR-COTS: Distributed Embedded Architectures using Commercial Off-The-Shelf Components View project

SafeCloud View project

All content following this page was uploaded by Paolo Romano on 13 November 2017.

The user has requested enhancement of the downloaded file.

The Future(s) of Transactional Memory

Jingna Zeng*†, João Barreto†, Seif Haridi*, Luís Rodrigues†, Paolo Romano†
*KTH/Royal Institute of Technology, Sweden
†INESC-ID/Instituto Superior Técnico, Universidade de Lisboa, Portugal

***Abstract*—This work investigates how to combine two powerful abstractions to manage concurrent programming: Transactional Memory (TM) and futures. The former hides from programmers the complexity of synchronizing concurrent access to shared data, via the familiar abstraction of atomic transactions. The latter serves to schedule and synchronize the parallel execution of computations whose results are not immediately required.**

While TM and futures are two widely investigated topics, the problem of how to exploit these two abstractions in synergy is still largely unexplored in the literature. This paper fills this gap by introducing Java Transactional Futures (JTF), a Java-based TM implementation that allows programmers to use futures to coordinate the execution of parallel tasks, while leveraging transactions to synchronize accesses to shared data. JTF provides a simple and intuitive semantic regarding the admissible serialization orders of the futures spawned by transactions, by ensuring that the results produced by a future are always consistent with those that one would obtain by executing the future sequentially.

Our experimental results show that the use of futures in a TM allows not only to unlock parallelism within transactions, but also to reduce the cost of conflicts among top-level transactions in high contention workloads.

I. Introduction

Transactional Memory (TM) is probably among the abstractions for parallel programming that have garnered the largest interests of the research community over the last decade. TM relieves programmers from the complexity of defining lock-based inter-thread synchronization mechanisms to safe guard concurrent accesses to shared data. With TM, programmers need simply to define which code blocks should be executed as *transactions*. Under the hood, the TM run-time system automatically performs concurrency control ensuring that transactions only commit if their execution is equivalent to a serial one. TM is implementable in software [1], hardware [2], or combinations therof, and its relevance has been strongly amplified by the recent integration of hardware TM support in the latest generations of CPUs by Intel and IBM, and by the inclusion of programming constructs for TM in the standard C/C++ languages.

The adoption of TM into mainstream programming will strongly depend on how easy it will be to integrate memory transactions into existing programming languages, abstractions and patterns. Unfortunately, the current state-of-the-art implementations of TM are not necessarily compatible with important programming constructs that are widely used in the programming community. One striking example of such a conflict can be found with the powerful and widely-used abstraction of futures [3], [4]. Futures are a convenient programming construct that activates a parallel computation (typically encapsulated by a method in an object-oriented programming language) and returns a placeholder, called future (or sometimes promise [5]). The returned future can be used by the subsequent instructions on the calling thread (also called *continuation*) to check whether the computation has already completed, and to obtain the result of the computation once it becomes available. Futures are part of C++, Java and the .Net platform (among others), hence a familiar construct to the average parallel developer in such environments. Yet, perhaps surprisingly, futures are not compatible with any current state-of-the-art implementations of TM. This paper fills this gap by proposing a powerful abstraction, the *transactional future*. Transactional futures correctly combine TM with futures, allowing programmers to exploit intra-transaction parallelism via the abstraction of futures, while delegating to TM the complexity of regulating concurrent access to shared data.

The design of an efficient run-time system supporting the transactional future abstraction introduces two subtle issues. First, enabling transactions to launch futures implies that computations executing as futures can access shared data and generate conflicts with their continuations, as well as with other futures belonging to the same or to a different transaction. Handling these conflicts requires using specialized techniques, which are otherwise unneeded in conventional TM systems. Second, the integration of futures and transactions opens a relevant question regarding which serialization orders are admissible between futures and their continuations, as well as among futures. In order to preserve the ease of use of TM, we support a consistency model of transactional futures that is simple and intuitive, namely, the equivalence to sequential execution not using futures.

Besides specifying the semantics and consistency model of the transactional future abstraction, we present the design of a software-based TM, called Java Transactional Futures (JTF), that implements the proposed abstraction for the Java run-time. The JTF system transparently wraps futures and continuations into atomic blocks, which run as parallel sub-transactions and are guaranteed to be serialized as if they were instantaneously executed at the moment in which they were spawned.

We evaluated JTF by parallelizing, using transactional futures, two popular benchmarks for transactional systems (i.e., both TM and database systems) and tested it using a 48-core machine. The results of our experimental study show that, by exploiting intra-transaction parallelism, JTF can not only significantly reduce the execution time of long running transactions, but also strongly benefit throughput in contention-prone workloads, by reducing the likelihood of conflicts among

transactions and the performance penalty due to aborts.

II. Defining the Semantics of Transactional Futures

The transactional future abstraction reconciles futures and transactions to enable intra-transaction parallelism while synchronizing the parallel computations that result from the activation of a future. Computations executing as futures, just like continuations, can perform concurrent access to shared data, and the TM run-time system is responsible for regulating such accesses. To achieve this goal, it is necessary to extend the conventional TM execution model and define how futures interact with continuations and with other futures, belonging to the same or to a different encompassing transaction.

A transactional future is *submitted* (or spawned) in the context of a transaction.[1] We designate the transaction that submits a transactional future as its parent transaction. After submitting a transactional future, the parent transaction obtains a reference that can be invoked to *evaluate* the transactional future, i.e., to obtain the return value that the future produced after it committed. This invocation is blocking, i.e., the thread that requests the evaluation waits until the transactional future commits. A transactional future can be evaluated by other threads/transactions than its parent; in fact, the transaction that submits a transactional future and obtains a reference to it can then share that reference with other threads.

As the name suggests, the transactional future runs as a transaction, more precisely a child transaction of the parent transaction. As such, its accesses to shared memory can be tracked and regulated by the TM system. Running in the context of a child transaction makes the execution of a transactional future dependent on the top-level transaction where it was invoked: if a top-level transaction aborts, a cascade abort of its transactional future(s) will apply.

After submitting a future, the parent transaction halts its execution and runs another sub-transaction, called *continuation*. Associating a new transactional context with the continuation, distinct from its parent's one, enables rolling back the continuation if it conflicts with itsl future without having to roll back the entire parent transaction (and its descendants).

Submitting transactional futures is not limited to top-level transactions; transactional futures can be submitted from within *any* transactional context, including continuations or transactional futures. Hence, the execution of a program yields a set of binary trees of transactions, as the one illustrated in Fig. 3a. Top-level transactions are the roots of each tree, which can be arbitrarily deep. Each node corresponds to a submit point that produces always two sibling sub-transactions: a transactional future and the corresponding continuation.

When a transactional future is submitted, it inherits the current *snapshot* of its parent and, by recursion, its ancestors. More precisely, when a transactional future *t* reads from a memory location that had been written by any of its ancestors, *t* should observe the most recently written value by its ancestors. The same rule applies to continuations.

As for isolation guarantees among each top-level transaction (including any sub-transaction that descends from that top-level transaction), we consider *opacity* [6], a standard consistency criterion for TM. Informally, opacity guarantees strict serializability [7] for committed transactions. Further, it prevents transactions that abort from observing arbitrary snapshots not producible by a sequential execution of a subset of the committed transactions.

However, defining isolation between transactions from different top-level trees is not enough, as we also need to consider the isolation between sub-transactions descending from the same top-level transaction. The proposed model considers that any transactional future and continuation executes in isolation relatively to its own sibling. When a sub-transaction (a transactional future or a continuation) commits, its write set becomes immediately visible to its sibling transaction. Hence, the continuations and futures of a given top-level transaction will only observe their mutual updates if that is compatible with their serialization orders. Furthermore, the read and write sets of a sub-transaction that commits are consolidated by the parent transaction (i.e., merged with the parent's read and write sets). Consequently, the updates performed by the continuations and the transactional futures of a top-level transaction T are only made (atomically) visible to other top-level transactions when T commits.

Let us now discuss how futures and continuations of a top-level transaction should be serialized. Intuitively, both the future and the continuation sub-transactions ought to appear as executed atomically and serialized after their parent (sub-)transaction, i.e., they should observe values written in the context of the parent (sub-)transaction. However, the decision of whether a future should be serialized before or after its continuation is a less obvious one. We advocate that serializing a transactional future at the time of its submission (i.e., before the continuation), which we call *strong ordering* semantics, is definitely more appealing for two reasons: *i)* It ensures that the parallel execution of a method in a future is equivalent to a sequential run in which the method is executed synchronously, and not via a future. *ii)* In scenarios in which transactional futures can be evaluated out of the context in which they were submitted, it may be hard to reason about which sets of operations shall appear as atomically executed with respect to a transactional future.

We provide two concrete examples of why weaker semantics are problematic. Consider first the execution in Fig. 1, in which transaction T_0 submits transactional future T_{F1}, which in its turn submits transactional future T_{F2}, which is then evaluated by T_0. In this execution, if T_{F2} were allowed to be serialized at the time of its evaluation, it should observe both w(x,x1), issued by T_{F1}, and w(y,y0), issued by T_0. In other words, the transactional context associated with T_{F2}'s continuation would have to encompass both writes although they belong to two distinct sub-transactions. Consider now the example in Fig. 2,

[1] A transactional future submitted/evaluated outside of a transactional context can be considered as submitted from within an empty top-level transaction that encompasses solely its submission/evaluation.

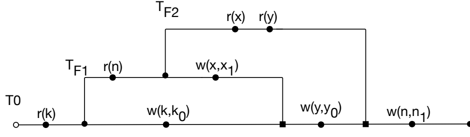


Fig. 1. Chained transactional futures.

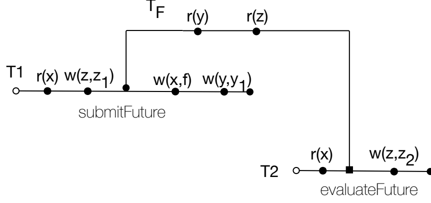


Fig. 2. Top-level transactions communicate via a transactional future.

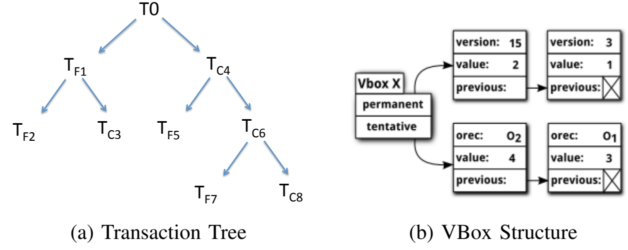
in which two top-level transactions, T_1 and T_2, communicate with each other by means of a transactional future, T$_F$, which is submitted by T_1 and evaluated by T_2: If T$_F$ were allowed to be serialized at the time of its evaluation, we argue that it would be unclear to even define its continuation (with respect to which, we recall, T$_F$ should appear as atomic).

Strong ordering semantics avoid the ambiguities illustrated above. Furthermore, with strong ordering semantics, the evaluation time of a future has no impact on its serialization order; only the submission time counts. This allows to use transactional futures in a safe and intuitive manner. Consider for instance the family of futures and continuations captured by the tree depicted in Fig. 3a. It is easy to see that the continuation of T_{F1} (i.e., T$_{C4}$) has to be serialized after T$_{F1}$, T$_{F2}$ and T$_{F2}$'s continuation (i.e., T$_{C3}$). Also, all the sub-transactions in the right sub-tree of T_0 can only commit after all the sub-transactions in T_0's left sub-tree have committed. Finally, the results produced by all transactional futures and continuations of this family are only visible to other top-level transactions once T_0 commits.

III. Overview of The JTF system

Analogously to other Java-based STM [8], [9], JTF requires programmers to explicitly identify the objects whose accesses need to be tracked by the transactional system. This is achieved by storing references to objects or primitive data types within a data container, called *VBox* (versioned box), that exposes methods to retrieve (*get*) and update (*put*) the current value of the object. These methods are intercepted by the JTF runtime, which uses them to track and regulate concurrent data accesses from within the transactional contexts.

Like classic TM systems, JTF exposes methods to begin, commit and abort transactions. In addition to conventional TM systems, though, it provides methods also to request the execution of a method (via the standard *Callable* interface in Java) as a transactional future, and to evaluate transactional futures' results. When a transactional future is submitted, the JTF runtime schedules its execution on an internal thread pool and transparently wraps the execution of the future's method in



(a) Transaction Tree (b) VBox Structure

Fig. 3. Data Structure under JTF System

the context of a new (sub-)transaction. Further, JTF also starts another child transactional context to run the continuation. In this way, JTF can discard all the effects produced by the continuation and still preserve the effects of the parent transaction before the invocation of the transactional future (i.e., support partial rollback).

JTF allows for evaluating the future's result at any time, without requiring the evaluation to occur from within a transactional context. The evaluation semantic of a transactional future is identical to that of a plain future, blocking the calling thread until the future's execution has completed. A reference to the future can be propagated among different threads (e.g., via writes to shared memory), which allows to use transactional futures as a channel to support inter-thread communication. This mechanism does not interfere with Java's garbage collection mechanism, as a transactional future and its result can be safely garbage collected as soon as no other object stores its reference.

A. Concurrency control

We start by discussing how JTF treats top-level transactions, then we discuss how transactional futures are handled.

Top-level transactions. Every top-level transaction in JTF has a *version* number which is assigned when the transaction is created, and establishes the data snapshot visible to that transaction during execution. This number is fetched from a global counter that represents the version number of the latest read-write transaction that successfully committed. Child transactions also receive a version number, which they inherit from the parent transaction.

JTF stores in the VBoxes the set of committed (or permanent) data versions that may be required to process reads from concurrent transactions. The committed versions are stored in a sorted list, which we call *permanent list*, in descendent order of recency and are tagged with a version number that defines the serialization order of the top-level transaction that committed this version (see Fig. 3b).

The write operation of a top-level transaction is implemented by simply storing the value in a private writeset. Reads are managed by first doing a lookup in the transaction's writeset, and, if the requested data item is not found there, the most recent version created by a transaction that committed before the transaction started is returned.

JTF integrates a lock-free commit algorithm, first introduced in JVSTM [8], which uses a helping mechanism to implement the following two steps in a non-blocking, yet atomic, fashion: increasing the global counter and writing-back the values from the transaction's write-set to the corresponding VBoxes.

Transactional Futures: submission. As already mentioned, JTF transparently wraps the execution of a transactional future and its continuation into two sub-transactions. These sub-transactions inherit the *version* number from their parent, which ensures that they will be serialized in the same order as their parent with respect to other top-level transactions.

JTF supports partial roll-back of a transactional tree, e.g., if a continuation misses the write generated by its corresponding future[2], only the sub-tree rooted on the continuation sub-transaction is aborted. To this end, the JTF run-time check-points the control state of the thread submitting the future, by means of a *first-class-continuation* (FCC) [10]. FCCs should not be confused with the continuations of (transactional) futures; FCCs are a lightweight checkpointing mechanism that allows to reify, save and restore the program control state (e.g., its stack and the JVM registers' value). In JTF, we use the FCC support provided by the OpenJDK Hotspot VM [10], which introduces very limited overhead and, unlike other approaches [11], does not require any byte-code rewriting.

Transactional Futures: read and write operations. Unlike top-level transactions, sub-transactions do not maintain a private write-set. When a sub-transaction writes, it acquires a lock (valid for the entire transactional tree) and inserts its version in a second list, called *tentative list*, in the corresponding VBox (see Fig. 3b). Unlike the versions in the permanent list, each tentative version is associated with the *orec* (ownership record [12]) of the (sub-)transaction that created it. The *orec* maintains a pointer to the (sub-)transaction (the owner), the version of the write (*txTreeVer)* and the owner status. Each sub-transaction creates the *orec* when the write occurs and propagates it to its parent when the sub-transaction commits.

In order to determine which tentative data item versions are visible to the sub-transactions of a transactional tree, sub-transactions maintain two additional meta-data: *nClock*, a counter initialized with 0 and incremented whenever a direct child transaction commits; *ancVer*, a map computed when the sub-transaction starts, which includes the parent's *ancVer* extended with the parent's current *nClock* value.

A sub-transaction T can only observe the tentative version of a transaction T' if, at the time in which T started: i) T' has already committed, and ii) the commit of T' has been already propagated to an ancestor of T. This is verifiable by checking in the *ancVer* of T whether the owner of the *orec* of a data version is an ancestor of T (*orec.owner*$\in T$.*ancVer*), and verifying whether such ancestor had already witnessed the commit of T' at the time in which T had started (*orec.txTreeVer*$\leq$T.*ancVer*(*orec.owner*)).

[2]We say that a continuation, T_C, misses the write of its corresponding future, T_F, if T_C issues a read on a data item x, for which a new version, x_F, is produced by T_F, and T_C does not return x_F.

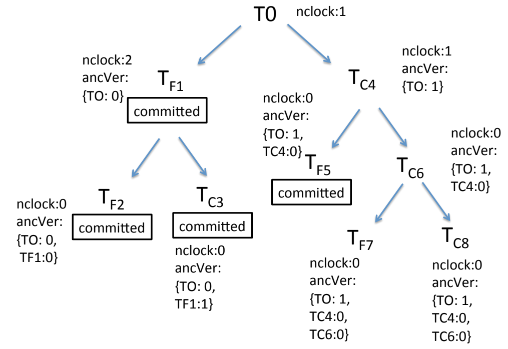


Fig. 4. Metadata reflecting the visible snapshots in a transactional tree.

This mechanism is illustrated in Fig. 4, which depicts the metadata maintained at a given point of execution of the transactional tree originally considered in Fig. 3a. $T0$'s *nClock* reflects the commit of its left sub-tree, having as root T_{F1}. By T_{C4}'s *ancVer*, we get that when T_{C4} started, T_{F1} had already propagated its commit to $T0$. T_{C4} and its descendants can hence observe the tentative writes of any of the sub-transactions of the left sub-tree of $T0$. T_{C6}, however, cannot see the tentative versions of T_{F5}, although T_{F5} has already committed. This is because at the moment in which T_{C6} started, T_{F5} had not yet committed (as the T_{C4}'s entry in the *ancVer* of T_{C6} is 0).

Transactional Futures: commit phase. Whenever a sub-transaction (either a transactional future or a continuation) 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 transactional futures and continuations. In practice, this implies that when a sub-transaction running a transactional future or a continuation finishes execution, and before it validates, it must wait that all other sub-transactions that precedes it according to the strong ordering semantics have validated and committed.

When a child transaction T reaches its turn to commit, it must then check if there is an intersection between its reads and the writes of some other sub-transactions (in the same transaction tree) that have committed while T was executing. In that case, it means that a conflict that broke the sequential semantic occurred, and T must be re-executed.

As already mentioned, the commit procedure of a sub-transaction (either running a transactional future or a continuation) ensures that the writes it performed are propagated to the parent transaction, which becomes the new owner of these versions. From that point on, those writes are made visible to new child transactions the parent might spawn. This allows transactions that re-execute, due to a conflict, to read the writes they missed on their previous execution.

Once all transactions in the transactional tree have committed, the control is passed to the top-level transaction. At that point, the top-level transaction will validate its execution

Algorithm 1 Write Procedure by Transactional Future or Continuation

```
1: Write(T,vbox,value):
2: pointerWrite ← vbox.tentative
3: orec ← pointerWrite.orec
4: status ← orec.status
5: if orec.owner == T then
6:         ▷ T owns the tentative version
7:     pointerWrite.value ← value
8:     return
9: end if
10: if status ≠ RUNNING then
11:     if pointerWrite.CASorec(orec,T.orec) then
12:         pointerWrite.value ← value
13:         return
14:     end if
15:         ▷ T fails to acquire ownership
16:     pointerWrite ← vbox.tentative
17:     orec ← pointerWrite.orec
18: end if
19: if orec.owner.root ≠ T.root then
20:         ▷ write-write conflict between top-level trans.
21:     ownedbyAnotherTree(T,vbox,value)
22:     return
23: end if
24: for all pointerWrite in vbox.tentative do
25:         ▷ Tentative ver. list owned by another sub-txn
26:         ▷ T iterates over the list to insert its version
27:     if follows(T,pointerWrite.owner) then
28:         tentativeWrite.CASnext(new TentativeVersion(value))
29:         return
30:     end if
31:     if pointerWrite.orec.owner == T then
32:         pointerWrite.value ← value
33:         return
34:     end if
35: end for
```

Algorithm 2 Read Procedure by Transactional Future or Continuation

```
1: Read(T,vbox):
2: pointerWrite ← vbox.tentative
3: orec ← pointerWrite.orec
4: status ← orec.status
5:         ▷ fast path when no read-after-write
6: if status ≠ RUNNING then
7:     return readFromPermanent(vbox)
8: end if
9: while pointerWrite ≠ null do
10:     if pointerWrite.owner == T ∧ status ≠ ABORTED then
11:         return pointerWrite.value
12:     end if
13:     if T.ancVer.contains(orec.owner)
        ∧ orec.txTreeVer ≤ T.ancVer.get(orec.owner) then
14:             ▷ add to the child transaction T's read-set
15:         T.readSet.put(vbox)
16:         return pointerWrite.value
17:     end if
18:     pointerWrite ← pointerWrite.previous
19:     orec ← pointerWrite.orec
20: end while
21:         ▷ confirm no read-after-write could happen
22: if T.root.writeset.contains(vbox) then
23:     return T.root.writeset.get(vbox)
24: end if
25: return readFromPermanent(vbox)
```

against other top-level transactions in the system and commit.

IV. JTF's CONCURRENCY CONTROL IN DETAIL

This section presents the pseudo-code of JTF's concurrency control algorithm. For space constraints, we describe the behaviour of transactional futures and continuations, omitting the pseudo-code of top-level transactions.

A. Write operations

Alg. 1 reports the pseudo-code for managing the write operation of a sub-transaction. When writing to a *VBox*, the (sub)transaction T fetches the tentative version at the head of the tentative list and reads its ownership record *orec* to tell whether it owns that version or not (line 7). In this positive case, it simply overwrites the previous write.

Otherwise, T checks if the transaction that created that tentative version has already completed execution, in which case T attempts to acquire ownership of the tentative write at the head of the list (line 11) using a compare-and-swap (CAS). If the CAS fails, we know 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 transaction trees. If the owner belongs to a different transaction tree, i.e., inter-tree conflict occurs, the transaction uses a fallback mechanism, *ownedbyAnotherTree()* (line 21). This mechanism aborts the sub-transaction, up to the root ancestor. Then it re-executes the affected sub-transactions in the context of the root top-level transaction. The key difference is that top-level transactions maintain a traditional write-set, called *rootWriteSet*, to use when a VBox is locked by another transactional tree.

If the owner of the tentative version at the head of the list belongs to the same transactional tree (line 24), T iterates over the tentative version's list until it finds a proper place to insert the new tentative write. Recall that the tentative version list is organized by descending order of recency, according to the serialization order imposed by the strong ordering semantics. In order to determine whether T should be serialized after the owner, say T', of a tentative version, the *follows()* functions is used, which compares the *ancVer* of T and T' and identifies their first (closest to the root) ancestor of T not in common with T', say T''. If T'' is a continuation, or if $T'' = \emptyset$ (i.e., T and T' have the same parent) and T is a continuation, then T follows T' in the serialization ($T' \rightarrow T$).

As we will discuss shortly, maintaining the tentative version list sorted by serialization order, allows better performance of the read procedure as reads can return the first value encountered in the *tentative write list* which meets the value visibility rule, avoiding iterating the whole list. Furthermore, maintaining the tentative version list sorted ensures that when the top-level transaction commits, it can find the values that must be written back to the permanent write lists on the head of the tentative write lists.

B. Read operations

The pseudo-code for managing the read operations of a sub-transaction is reported in Alg. 2.

When processing a read, a transaction T checks whether the latest tentative write was performed by a transaction that

Algorithm 3 Wait commit rule used to enforce strong ordering semantics

```
 1: WaitTurn(T):
 2: if T is a Continuation then
 3:     wait until T.parent.nClock == 1
 4:     return
 5: end if
 6:             ▷ T is a Transactional Future
 7: Set ancCont ={T' ∈ T.ancVer : T' is a continuation}
 8: Transaction anc = closest ancestor of T in ancCont
 9: if anc = ∅ then
10:     return
11: end if
12: wait until anc.parent.nClock==1
13: return
```

has already committed or aborted (line 6). This means that this VBox does not contain tentative versions created by sub-transactions belonging to the same transactional tree of T. Hence, the read is simply served by selecting the most recent version in the permanent list committed before T started (this logic is encapsulated in the *readFromPermanent*() primitive).

Otherwise, T needs to check whether itself or any of its ancestors have previously written to the *VBox*. At this point, T iterates over the tentative version list of the *VBox* until one of the following conditions is verified: (1) T is the owner of the tentative version. In this case, the transaction also needs to make sure that the write does not belong to a previous aborted execution. This previous execution corresponds to the case in which the sub-transaction failed validation and was forced to re-execute. If the write was performed in T's current execution, then no further checks are needed and the procedure returns that value (line 11). (2) The owner of the tentative write is an ancestor of T. Under this circumstance, 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* (line 13-19).

If no valid value was found in the tentative version list, then there are no tentative versions produced by sub-transactions in T's transactional tree that are visible to T. Therefore, T checks whether its top-level transaction had already written to the same VBox (lines 21-22). If this is the case, it reads from the top-level transaction's write-set. Else, it fetches a committed value from the permanent version list.

C. Commit procedure

The pseudo-code for committing and aborting a sub-transaction is reported in Alg. 4.

When a sub-transaction T tries to commit, it needs to validate its read-set in order to detect if it has missed the writes produced by other sub-transactions belonging to the same transactional tree and preceding T in the serialization order imposed by the strong ordering semantics. To this end, before activating its validation procedure, a sub-transaction first waits for the commit events of every sub-transaction that should be serialized before it. We postpone shortly the description of the

Algorithm 4 Commit and Abort Procedure by Transactional Future or Continuation

```
 1: Commit(T):
 2: waitTurn(T)
 3: if ¬ validate(readSet) then
 4:     Abort(T)
 5:     return
 6: end if
 7: T.parent.nClock += 1
 8: T.orec.txTreeVer ← parent.nClock
 9: T.orec.owner ← parent
10: for all t in committedChildren do
11:     t.orec.txTreeVer ← parent.nClock
12:     t.orec.owner ← parent
13: end for

14: Abort(T):
15: waitTurn(T)
16: for all vbox in T.boxesWritten do
17:     pointerWrite ← vbox.tentative
18:     if pointerWrite.orec.owner == T then
19:         revertOverwrite(vbox)
20:     end if
21: end for
22: T.orec.status ← ABORTED
23: for all t in childrenTransactions do
24:     t.orec.status ← ABORTED
25: end for
```

logic used by JTF to enforce this serialization order, which is encapsulated in the *waitTurn* function.

The validation phase, encapsulated by the *validate*() primitive (line 3), scans the tentative version lists of the VBoxes read by the transaction. If a version is found belonging to an ancestor of T, and this version does not coincide with the one in T's readset, validation fails and the transaction must be aborted and re-started. If the transaction was running a transactional future, it simply calls the abort method and re-executes the transactional future from the beginning. Otherwise, if the transaction was running a continuation, it aborts and uses the FCC support in order to restore the execution state to the point where the continuation started.

If a sub-transaction T passes the validation phase, it increases by one the *nClock* of its parent transaction (line 7). Next T propagates its own writeset, and that of its child transactions, to the parent (sub-)transaction. To this end, T updates its own *orec*, and the ones of its child transactions, by setting : i) the owner field to point to its parent transaction; ii) the *txTreeVer* field to *nClock*'s value of its parent transaction.

When aborting, transactions must revert the writes they performed when their write is at the head of the tentative version list. In this case, an aborting transaction cannot directly mark its tentative version as aborted. In fact, the head of the tentative list is used to establish a lock that grants access to the entire transactional tree[3]. If some other sub-transaction of the same transactional tree had stored a version v in the second position of the tentative list, the head of the tentative version list has to be substituted with v. In order to make this operation

[3]Recall that, when a transaction is performing a write, it checks if the owner of the write at the head of the list has already finished (line 10 of Alg. 1). Once the lock is established, only transactions in the same transactional tree of the transaction are allowed to write to the list.

lock-free, we need to make sure that no transaction changes any of the version between the tail of the list and the version of the aborting transaction [9]. To ensure this, a transaction T only aborts when every transaction that could precede T in the serialization order has finished executing (either aborted due to an inter-tree conflict or committed). This is done, just like for the case of commits, by using the *waitTurn*() function.

Finally, the transaction completes its abort by changing the status of the *orec*s it controls (its own *orec* and its children transactions' *orec*) to ABORTED (lines 22-25).

D. Transaction serialization order

To enforce a transaction serialization compliant with the strong ordering semantics defined in Section II, JTF commits, at any point in time, at most one sub-transaction of a transactional tree, and only after having ensured that all the sub-transactions that precede it in the serialization order have already committed.

It should be noted that, since JTF supports nesting of transactional futures, it is, in general, not possible to establish the serialization order of sub-transaction *a priori*, i.e., upon their creation. Consider, for instance, the transactional tree in Fig. 3a: had T_{F1} not submitted T_{F2}, T_{C4} would have received a serialization order equal to 2 (and not 4).[4] JTF tackles this problem by establishing the commit order of transactional futures/continuations *a posteriori*, via two simple and lightweight waiting rules — one for transactional futures and one for continuations (see *waitTurn*() in Alg. 3).

The waiting rule for a continuation, say T, is very simple: it suffices to wait till the *nClock* of its parent becomes equal to 1. This means that the sub-tree rooted in the corresponding transactional future, which directly precedes T in the serialization order, has already committed.

The waiting rule for a transactional future, say T_F, is slightly more complex, but it is also based on the same principle: identifying the transaction that immediately precedes T_F in the serialization order. Let T_C be the ancestor of the first continuation encountered by traversing "upwards" (from the closest to the furthest ancestor) T_F's *ancVer*. If $T_C = \emptyset$ then it means that T_F is the first transactional future to commit according to the strict ordering semantics, and can commit without incurring any wait. Else, if such a T_C exists, it is safe for T_F to validate and commit as soon as the left-subtree of T_C's parent has completed execution. In fact, no other sub-transaction can, from that point on, ever commit and serialize before T_F. Hence, a transactional future can only commit when the *nClock* of T_C's parent is equal to 1.

E. Optimizing read-only transactions

Since JTF uses a multi-versioning concurrency control, the snapshot observed by read-only top-level transactions is guaranteed to be consistent (although possibly obsolete, i.e., not reflecting the effects of update transactions that committed during the read-only transaction's execution). Thanks to this property, read-only top-level transactions can skip validation, and immediately commit when they finish executing.

However, even if transactional futures were marked as read-only, it is in general not safe to skip their validation. In fact, a read-only transactional future may miss the write performed by a preceding (according to the serialization order) sub-transaction. On the other hand, in JTF, we detect and take advantage of the situation in which it is actually safe to skip the validation of a read-only transactional future. This is true whenever all other sub-transactions that precede it in the serialization order have already committed before it started, or if all of those transactions are read-only as well. To support the latter optimization, we added a new list in every top-level transaction, which contains the identifier of every committed read-write sub-transaction of a transactional tree. Whenever a read-only transactional future reaches the commit phase, it waits for its commit turn using the *waitTurn*() primitive. Then it checks if there is any read-write sub-transaction in this list. If the list is empty, then the transaction can safely skip validation at the end of execution.

V. Evaluation

In this section we conduct an experimental evaluation aimed to seek answers to the following key questions:

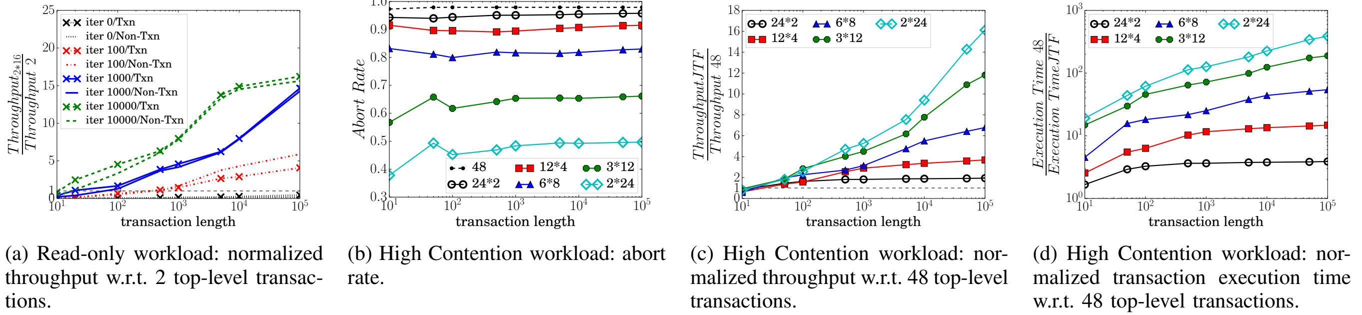
- how large are the overheads introduced by JTF to coordinate the execution of transactional futures, and how do they compare with the inherent costs of *non-transactional* futures?
- what is the minimum granularity of a transaction for which it is beneficial, from a performance perspective, to use parallelization via transactional futures?
- what performance gains can be achieved by JTF in presence of diverse workloads encompassing both synthetic benchmarks and well-known transactional benchmarks?

The results presented in this section are the average of five runs, executed on a machine with a AMD Opteron 6168 processors (48 cores total), 128GB of RAM, Linux 2.6.32 and an OpenJDK 64-bit Server 1.7.0 (build 19.0-b03) JVM.

Synthetic benchmarks. We start by presenting the results obtained by using a synthetic benchmark that we designed to exert tight control on the characteristics of the generated workload and to emulate extreme scenarios that allow us to assess the overheads and potential gains of JTF. This benchmark generates, in each transaction, a configurable number of read/write memory accesses to an array of 1M elements. In between two memory accesses, the benchmarks emulates the execution of CPU-bound computations via a loop that executes a tunable number of iterations, called *iter*, in which we issue register-based arithmetic operations. Tuning the value of *iter* allows to generate diverse load pressure for the CPU and memory subsystems, and synthesize CPU-bound rather than memory-bound workloads.

We start by considering, in Fig. 5a, a workload composed solely by read-only transactions, in which we vary the transaction length (i.e., number of read accesses within a transaction)

[4] On the other hand, if nesting of futures was not to be supported, futures would be submitted only by the thread executing the top-level transaction, and their serialization order would coincide with their submission order.



(a) Read-only workload: normalized throughput w.r.t. 2 top-level transactions.

(b) High Contention workload: abort rate.

(c) High Contention workload: normalized throughput w.r.t. 48 top-level transactions.

(d) High Contention workload: normalized transaction execution time w.r.t. 48 top-level transactions.

Fig. 5. Synthetic Benchmarks.

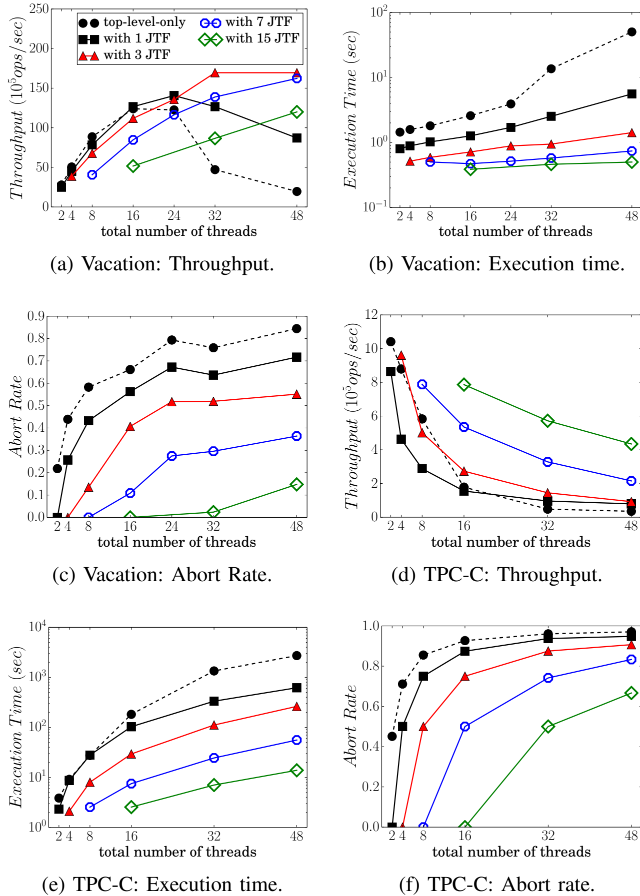
from 10 to 100K and the number of CPU-bound iterations between two consecutive memory accesses from 0 to 10K. The items targeted by read operations are selected uniformly at random across the whole array. It should be noted that, since we consider a workload composed exclusively of read-only transactions, synchronization is, in fact, unnecessary, and correctness could be ensured even by using a plain/non-transactional future implementation. Also, with such a conflict free workload, futures can provide no benefit by reducing the likelihood or the cost of aborts. Hence, by comparing the performance of JTF with that of a non-transactional future in this workload, we can isolate and quantify the overhead induced to enforce the transactional future semantics and the costs that are inherent to the use of futures, such as inter-thread communication and increased contention on the memory bus among the threads executing the future(s) and the continuation. On the y-axis, we report the normalized throughput achievable by parallelizing the transactional code across 16 threads (i.e., 15 executing futures and 1 executing in the continuation) and running concurrently two top level transactions. The normalization is with respect to a baseline that does not exploit futures and uses two threads. Hence, the maximum throughput increase expectable amounts to 16×.

The data in Fig. 5a shows that JTF achieves close to optimal performance if transactions have at least 10K memory accesses and the workload contains sufficiently long CPU-bound computations. In fact, when setting the value *iter* to 0 (which yields a completely memory bound workload), increasing the degree of intra-transaction parallelism just hampers performance even for long-running transactions that execute 100K memory accesses. This is somewhat unsurprising, as by increasing the total number of concurrently active threads from 2 to 32, the memory subsystem is subject to a 16× larger load. In a memory-bound workload, this causes a surge in the average number of stall cycles incurred by CPUs due to memory accesses (a fact which we experimentally verified via the *perf* profiler). Yet, it is interesting to highlight that these overheads are not caused by the concurrency control scheme used in JTF. In fact, Fig. 5 shows that the performance of JTF is quite close to that of a non-transactional future implementation. These data confirm that most of the overheads incurred by JTF are inherent to the usage of futures, and that the additional overheads introduced by JTF are modest even in such a challenging scenario (the average slowdown w.r.t. non-transactional future is $<$1%).

Next, we set *iter* to 1k and consider conflict prone workloads. In this case, transactions perform a variable length prefix of read accesses, followed by 10 update operations on a set of 20 "hot spot" items (selected uniformly at random with restitution). We report the normalized throughput achieved by JTF when using a total of 48 threads, which we allocate to execute either futures or top-level transactions/continuations. In Fig. 5a, we use the $i * j$ notation to indicate that we execute i top level transactions, each parallelized via j threads ($j - 1$ executing futures and one executing the continuation). We use as baseline for normalization a configuration in which we use 48 concurrent top-level transactions (i.e., no futures). This allows us to compare whether, given a fixed pool of available threads, it is more beneficial to exploit them in order to pursue inter-transaction or intra-transaction parallelism.

As shown in Fig. 5b, in such a high contention workload, the usage of transactional futures significantly reduces the likelihood of conflict between transactions. This is due to two main factors: i) the more threads are allocated to futures, the smaller the number of concurrent top-level transactions, and, consequently, the lower the probability that the latter incur a conflict; ii) also, intra-transaction parallelism (via futures) allows for reducing the execution time of a single-top level transaction, which, in its turn, reduces the transaction vulnerability window as well as the cost of a transaction's restart. Such a reduction of the abort rate amplifies remarkably the gains achievable by JTF, when compared to the previously considered read-only workload: the throughput gains (Fig. 5c) start to be significant with much shorter transactions and the transaction execution latency, which accounts also for transactions' retries due to abort, is reduced by up to 400×. This is explainable considering that, in this high contention scenario, transactions can be re-executed tens of times before being committed, whereas the average number of transaction re-executions is around one in the 2*24 configuration.

Vacation and TPC-C. We now consider two well known and more realistic benchmarks for transactional systems: Vacation of the STAMP benchmark's suite [13] and TPC-C [14].



(a) Vacation: Throughput. (b) Vacation: Execution time.

(c) Vacation: Abort Rate. (d) TPC-C: Throughput.

(e) TPC-C: Execution time. (f) TPC-C: Abort rate.

Fig. 6. Vacation and TPC-C Benchmarks.

The former is a popular benchmark for TM systems, which emulates the activities of a travel agency; the latter is an OLTP benchmark for database systems that mimics the activities of a warehouse supplier. We adapted these benchmarks to be parallelized using JTF. In both benchmarks, we use transactional futures to parallelize long running transactions that execute a long cycle, during which they read a number of domain objects and compute various functions, e.g., they identify travels within a given price range, or compute the total amount of money raised by the warehouse.

For both the Vacation and the TPC-C benchmark we report in Fig. 6 the average throughput, execution time and abort rate. We use as independent variable in this experiment the total number of active threads, and report data for five different thread allocation strategies: a baseline in which no transactional future is used, and all available threads execute independent top-level transactions; four alternative configurations, which allocate the available threads to parallelize top-level transactions by means of, respectively, 1, 3, 5, and 7 transactional futures (along with one continuation thread).

By comparing the throughputs achievable by the two benchmarks (see Figs. 6a and 6d), we observe that, if transactional futures are *not* used, Vacation can scale up to approximately 16 threads; TPC-C, on the other hand, generates an inherently non-scalable workload, for which using more than one top-level transaction causes a quick surge in the transaction conflict probability (see Fig. 6f) and only hinders performance.

Overall, the use of transactional futures produces an improvement on performance, which is similar in both benchmarks. By allocating available threads to parallelize (a smaller number of) top-level transactions, rather than to activate additional, conflict-prone, top-level transactions, the likelihood of contention is strongly reduced. In other words, at high thread counts, the transactional future abstraction allows to utilize available computational resources in a much more effective way. From the throughput's perspective, transactional futures allow to extend the maximum achievable throughput by nearly 50% in Vacation and to scale up to 48 threads. In relative terms, the throughput gain when using 12 top-level transactions parallelized with 3 transactional futures (plus 1 continuation) is 8.4× larger than when using 48 non-parallelized top-level transactions. With TPC-C, where contention among top-level transactions is higher than in Vacation, the relative throughput gains deriving from transactional futures, at parity of totally used threads, are, unsurprisingly, even larger, extending up to 10.7× (again at 48 threads). Even more impressive are the benefits in terms of reduction of the time required to commit a transaction, see Figs. 6b and 6e, with gains peaks of up to two orders of magnitude for both benchmarks. Such striking gains can be explained, as already discussed when analyzing the synthetic benchmarks, by considering that transactional futures not only allow for reducing the average number of transaction re-starts due to aborts, but also for reducing the cost incurred when aborting transactions (i.e., the mean execution time of aborted transactions).

VI. Related Work

Futures [4] have reached a broad adoption among mainstream programmers for their convenience as a tool to easily expose fine-grained parallelism [15]. Java futures [16] can be seen as a form of method-level parallelism, as they can be used to explore parallelism in programs by forking at method calls. However, current implementations of Futures provided in the Java Development Kit lack suport for concurrency control among the asynchronous work performed by different future/ continuation tasks.

Safe futures [15] use techniques similar to those used in software-based thread-level speculation [17] to avoid pathological side-effects between a future and its continuation when both touch shared locations. 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, safe futures assume that the underlying program is single-threaded. In other words, that the only parallel threads consist of the future and its continuation. This constitutes a crucial hindrance for the adoption of safe futures in multi-threaded applications, which are parallelized using coarse-grained threads. To the best of our knowledge, our contribution is the first to propose safe support for futures in the scope of multi-threaded programs.

Another interesting line for exploring futures is their usage on concurrent shared data structures. Kogan and Herlihy [18] studied how futures can be used to optimize type-specific, long-lived concurrent data structures. Indeed, these authors had already discussed, as an interesting future work direction, the possibility of combining the abstractions of futures and transactions. Our work explores this idea and proposes, to the best of our knowledge, the first generic solution (i.e., not restricted to shared data structures) that supports the execution of arbitrary code in futures that can be executed within the context of atomic transactions.

A number of recent proposals advocate TM as an attractive paradigm for exploiting fine-grained nested parallelism. In a nested-parallel mode, each transaction is allowed to spawn sub-transactions that run in parallel. Most proposals in this direction consider nesting fine-grained parallel transactions in the context of traditional fork-join constructs [19]–[23]. Among these, the one whose design shares the major similarities with our proposed solution is JVSTM [9]. In fact, both JVSTM and JTF target the Java programming language and use a multi-version based concurrency control scheme. Yet, JTF's concurrency control provides stringent semantics regarding the commit order of futures, whereas JVSTM (and parallel nesting in general) does not impose any restrictions on the serialization order of sub-transactions. We refer to our recent work [24] for a broader discussion on possible semantics of transactional futures, as well as a comparison between the types of concurrent computations that can be supported via transactional futures and parallel nested transactions.

While appealing in theory, programming in a nested-parallel fashion using TM, when compared to flat-parallel programming, introduces new challenges that programmers must be aware of in order to build correct and efficient programs. The nested-parallel model is substantially more complex than the flat one, making it a cumbersome and error-prone programming model for the average programmer [25]. Furthermore, as we discuss earlier in the paper, supporting fine-grained transactional nesting with future introduces additional challenges that are far from trivial and simply do not exist in the context of fork-join nested parallelism.

VII. Conclusion

This paper investigated how to combine two powerful abstractions for concurrent programming, Transactional Memory (TM) and futures, by introducing the notion of transactional futures. Transactional futures allows programmers to leverage futures to coordinate the execution of parallel tasks, and transactions to synchronize accesses to shared data. The semantics of transactional futures are conceived with the objective of preserving the intuitiveness and ease of use of TM: the results produced by the parallel execution of transactional futures are always consistent with those that one would obtain by executing the futures sequentially.

Besides introducing the transactional future abstraction, we presented also JTF, a Java-based implementation of transactional futures. The core of JTF is an innovative concurrency control algorithm, which strives to maximize parallelism by using multi-versioning techniques, while enforcing strong guarantees regarding the serialization order of futures.

We evaluated JTF by using synthetic benchmarks as well as two popular benchmarks for TM and database systems. The experimental results show that the use of futures allows not only to untap parallelism within transactions, but also to reduce the cost of conflicts among top-level transactions in high contention workloads.

Acknowledgments

The authors are grateful to José Pereira, Nuno Diegues, and Ricardo Filipe for their valuable contributions to preliminary versions of JTF.

References

[1] N. Shavit and D. Touitou, "Software transactional memory," *Distributed Computing*, 1997.
[2] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *ISCA*, 1993.
[3] R. Halstead Jr, "Multilisp: A language for concurrent symbolic computation," *TOPLAS*, 1985.
[4] H. Baker Jr. and C. Hewitt, "The incremental garbage collection of processes," *ACM SIGPLAN*, Aug. 1977.
[5] D. Friedman and D. Wise, "The impact of applicative programming on multiprocessing," in *ICPP*, 1976.
[6] R. Guerraoui and M. Kapalka, "Opacity: A correctness condition for transactional memory," EPFL, Tech. Rep., 2007.
[7] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems (TODS)*, 1983.
[8] S. Fernandes and J. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *ACM SIGPLAN*, 2011.
[9] N. Diegues and J. Cachopo, "Practical parallel nesting for software transactional memory," in *DISC*, 2013.
[10] I. Anjo and J. Cachopo, "Improving continuation-powered method-level speculation for jvm applications," in *ICA3PP*, 2013.
[11] "The javaflow component." [Online]. Available: http://commons.apache.org/sandbox/commons-javaflow/
[12] V. Marathe and M. Scott, "A qualitative survey of modern software transactional memory systems," U. of Rochester, Tech. Rep., 2004.
[13] C. Minh *et al*, "Stamp: Stanford transactional applications for multi-processing," in *IISWC*, 2008.
[14] "Tpc-c." [Online]. Available: http://www.tpc.org/tpcc/
[15] A. Welc, S. Jagannathan, and A. Hosking, "Safe futures for Java," in *ACM SIGPLAN*, 2005.
[16] Oracle, "Interface future." [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html
[17] L. Hammond, M. Willey, and K. Olukotun, "Data speculation support for a chip multiprocessor," in *ASPLOS*, 1998.
[18] A. Kogan and M. Herlihy, "The future (s) of shared data structures," in *PODC*, 2015.
[19] K. Agrawal, J. Fineman, and J. Sukha, "Nested parallelism in transactional memory," in *ACM SIGPLAN*, 2008.
[20] J. Barreto *et al.*, "Leveraging parallel nesting in transactional memory," in *PPOPP*, 2010.
[21] H. Volos *et al.*, "Nepaltm: design and implementation of nested parallelism for transactional memory systems," in *ECOOP*, 2009.
[22] W. Baek and C. Kozyrakis, "NesTM: Implementing and Evaluating Nested Parallelism in Software Transactional Memory," in *PACT*, 2009.
[23] R. Kumar and K. Vidyasankar, "HParSTM: A hierarchy-based stm protocol for supporting nested parallelism," in *TRANSACT*, 2011.
[24] J. Zeng, P. Romano, L. Rodrigues, S. Haridi, and J. Bareto, "In search of semantic models for reconciling futures and transactional memory," 7th Workshop on the Theory of Transactional Memory, 2015.
[25] R. Filipe and J. Barreto, "Nested parallelism in transactional memory," in *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Springer, 2015.

View publication stats