Investigating the Semantics of Futures in Transactional Memory Systems

Jingna Zeng
KTH Royal Institute of Technology
Stockholm, Sweden
IST, University of Lisbon
Lisbon, Portugal

Shady Issa
INESC-ID & IST, University of Lisbon
Lisbon, Portugal

Paolo Romano
INESC-ID & IST, University of Lisbon
Lisbon, Portugal

Luis Rodrigues
INESC-ID & IST, University of Lisbon
Lisbon, Portugal

Seif Haridi
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

This paper investigates the problem of integrating two powerful abstractions for concurrent programming, namely futures and transactional memory. Our focus is on specifying the semantics of execution of "transactional futures", i.e., futures that execute as atomic transactions and that are spawned/evaluated by other (plain) transactions or transactional futures. We show that, due to the ability of futures to generate parallel computations with complex dependencies, there exist several plausible (i.e., intuitive) alternatives for defining the isolation and atomicity semantics of transactional futures. The alternative semantics we propose explore different trade-offs between ease of use and efficiency. We have implemented the proposed semantics by introducing a graph-based software transactional memory algorithm, which we integrated with a state of the art JAVA-based Software Transactional Memory (STM). We quantify the performance trade-offs associated with the different semantics using an extensive experimental study encompassing a wide range of diverse workloads.

CCS Concepts: • **Computing methodologies → Shared memory algorithms**; • **Theory of computation → Parallel computing models**.

Keywords: Software Transactional Memory, Parallel Programming, Futures, Synchronization, Concurrency Control

ACM Reference Format:
Jingna Zeng, Shady Issa, Paolo Romano, Luis Rodrigues, and Seif Haridi. 2021. Investigating the Semantics of Futures in Transactional Memory Systems. In *26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '21), February 27-March 3, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3437801.3441594

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PPoPP '21, February 27-March 3, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441594

1 Introduction

Transactional memory (TM) [24, 36] is regarded as an attractive paradigm to simplify the development of concurrent applications. TM borrows the abstraction of the atomic transaction from the database literature and applies it as a first-class abstraction in the context of generic (i.e., not sandboxed) parallel programs: by requiring programmers only to identify which code blocks should be executed atomically, and not how atomicity should be achieved. TM simplifies the development of concurrent applications [12, 13, 19, 26, 28, 33, 42, 43, 46], delivering performance on par with (and sometimes even higher than) complex, hand-crafted locking mechanisms [15, 38].

Over the last years, TM literature has focused on studying transaction execution models that assume transaction to issue operations *sequentially*. The problem of how to support intra-transaction parallelism has, conversely, garnered limited attention and, in this context, most of the works we are aware of have focused on investigating the scenario of a specific paradigm for expressing parallelism among multiple sub-tasks of a transaction, namely *parallel nesting* [2, 4, 5, 14].

This work aims at filling a gap in the literature by investigating the semantics that should be enforced by a TM system that allows expressing intra-transaction parallelism via another popular abstraction, which allows for generating a broader range of concurrent programming patterns than parallel nesting, namely *futures*. More precisely, we focus on defining desirable atomicity and isolation semantics for TM systems in which futures are used to coordinate the execution of parallel tasks, which we call *transactional futures*, whose accesses to shared data are synchronized via transactions.

Analogously to parallel nesting, futures allow programmers to express when parallelization is useful: at which point in an otherwise sequential program a parallel task should be started (i.e., when the future is created). Differently from the parallel nesting model, though, the future abstraction does not block the execution of the main thread, also called *continuation* in the context of futures, till the parallel sub-task it spawned completes execution. Conversely, a *future* is returned, i.e., a reference to the result that will be eventually produced by the task encapsulated in the future. This allows futures to achieve a greater flexibility in defining *when* the results of a parallel sub-task are actually required by the application: futures can be *evaluated* (i.e., their promise is queried to retrieve the result) in an asynchronous fashion and in an order that is totally unrelated with the orders in which they were spawned — whereas parallel nesting abides by a fork-join model, according to which the spawning thread remains blocked until the last sub-transaction completes, and only then resumes its execution.

Using a set of examples of concurrent computations of increasing complexity, we introduce the spectrum of issues that need to be addressed when defining the semantics of transactional futures. We show that, given the futures' ability to generate parallel computations with complex dependencies, there exist several plausible (i.e., intuitive) alternatives for what concerns the isolation and atomicity semantics of a transactional future and its continuation.

Based on these considerations, we characterize four different semantics over two dimensions: the degree of atomicity between futures and continuations, and their admitted serialization orders. The alternative semantics we propose explore different trade-offs between ease of use (simplicity of reasoning on the equivalent sequential histories), and efficiency (ability to avoid aborts or stalls by enforcing a different type of constraints on the serialization order of transactional futures).

We formalize the semantics utilizing a graph-based characterization of the logical dependencies that (sub-)transactions develop by accessing shared variables and creating/evaluating futures; the resulting graph is then used to formalize alternative definitions of continuations and impose different constraints on the serialization orders of transactional futures.

We show how to implement the proposed semantics by introducing a software transactional memory (STM) algorithm that orchestrates the execution of transactional futures via a novel graph-based concurrency control scheme. We implement this algorithm by extending a state of the art multi-versioned STM (JVSTM [7, 16]), and evaluate the performance trade-offs of the proposed semantics via an experimental study encompassing diverse workloads.

2 Related work

Futures were first introduced by Halstead [23], as a synchronization and scheduling language mechanism. The future's abstraction is nowadays widely supported in mainstream programming languages [10, 27, 35]. In Java, for instance, futures [27] can be used to explore parallelism by forking at the method calls. These implementations of futures lack support for regulating concurrent access to shared resources (e.g., shared variables) among different futures and continuations, delegating this responsibility to the programmer. The concept of safe futures [41] has been proposed to preserve the equivalence of serial execution although the future and its continuation may execute concurrently and access shared data. However, safe futures fail to unlock the full potential of the future abstraction, as they assume the underlying program to be single-threaded: the only two threads that may ever run in parallel are the ones that run a future and its continuation.

Kogan and Herlihy [25] studied how to leverage futures to parallelize concurrent data structures, exploiting *a priori* knowledge on operations' semantics and commutativity to combine and eliminate concurrent operations. Our work considers a more generic model, which supports the execution of arbitrary code in futures that execute as atomic transactions.

Most of the literature on TM that has looked at how to exploit intra-transaction parallelism has focused on a specific execution model, which is typically referred to as parallel nesting [2, 4, 17, 40]. In parallel nesting, top-level transactions can spawn (recursively) one or several nested sub-transactions. Unlike futures, parallel nesting is as an instance of the classic fork-join model, which supports a smaller class of parallel computations than futures (see Section 3.3).

We are aware of only 2 works [39, 44] that attempted to reconcile the abstractions of futures and transactions. JTF [44] is a TM that provides support for transactional futures, but assumes simplistic semantics (which we call *strongly ordered* in this paper), i.e., it imposes the serialization of futures at their submission point. Chocola [39], conversely, imposes futures to be serialized upon evaluation and forces them to be evaluated by their spawning transaction. Further, Chocola only considers write-write conflicts (thus it implicitly assumes no-blind writes) and targets serializability, and not opacity [22] as consistency criterion (as typical in TM environments). In this paper, we investigate multiple definitions of the isolation and atomicity semantics of transactional futures, motivated by the observation that a future offers two intuitive serialization points for its execution, namely upon submission *and* upon evaluation. This flexibility raises interesting opportunities as well as subtle issues. On the one hand, allowing transactional futures to be serialized according to orders that do not match the spawning order enables building more efficient TM systems that can reduce the abort rate and transactions' stall times — benefits that we shall quantify in Section 5. On the other hand, allowing futures to be serialized upon evaluation raises non-trivial issues related to the definition of the atomicity and isolation of *escaping futures*, i.e., futures that are evaluated in a different (sub-)transaction than the one in which they were spawned — as we shall discuss more in detail in Section 3.

Finally, the proposed formalization of transactional future and its STM implementation make use of a graph-based characterization of conflicts to determine serializability of transactional futures. As such, our work is related to the literature in the DBMS and TM domains that exploits conflict graphs both to implement permissive [21] concurrency control mechanisms [3, 30, 32, 45] and to formalize different transactional semantics [1, 20, 29]. The fundamental difference with respect to this body of literature is that our work extends and leverages the transaction dependency graph to reason on the relations between transactional futures and their continuations.

3 Semantics of transactional futures

As a first step to reason on the integration of the future abstraction in the TM paradigm, we first define the assumed model of execution of transaction and futures. We consider a set $\mathcal{TH} = \{Th_1,...,Th_n\}$ of *threads* which can communicate by reading and writing a set of shared variables V.

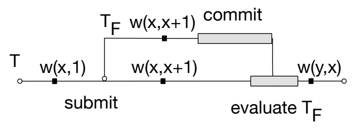
In the conventional transactional model, transactions start by issuing a *begin* operation, which can be followed by a sequence of *read* and *write* operations, and are finally completed by either a *commit* or *abort* operation. We say that two operations conflict if they access the same variable and at least one of them is a write operation. In case a read operation observes the value written by a write operation, we say that a read-after-write dependence has been developed by the two operations. Each operation has an associated *execution interval*, which starts when the operation is issued and ends when the TM returns the operation's result.

In order to integrate futures and transactions, we extend this model in a twofold way. First, we allow transactions to return values: this is done since the future abstraction supports the execution of tasks that generate results, and we intend to encapsulate the operations executed by a future within a transaction. Second, we allow transactions to issue two additional operations, i.e., *submit* and *evaluate*. The *submit* operation takes as input a transaction T, activates a parallel thread in which T will be executed, and returns a *future* object $f \in \mathcal{F}$. The returned future f can be passed as an input parameter to an *evaluate* operation to obtain the return value of T.

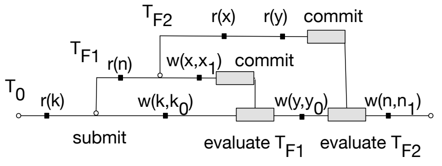
We assume that a future can only be submitted or evaluated within the context of a transaction. This can be enforced by wrapping any non-transactional submit and evaluate call within an otherwise empty transaction. We initially assume that a future is evaluated at most once and that *evaluate* blocks until the transaction associated with the future f completes its execution, i.e., when it successfully commits. This assumption will be relaxed in Section 3.2.

As in typical TM environments, we assume that if a transaction aborts due to contention, it is re-executed automatically. This implies that if an *evaluate* primitive associated with transaction T returns, then T has either been committed (possibly after several aborts due to conflicts and subsequent re-executions) or T has aborted due to an explicit decision of the program to abort T (via the *abort* operation).

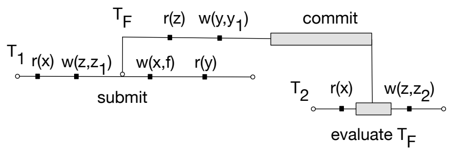
Transactions activated by threads that do not run in the context of a future are denoted *top-level* transactions. This transaction execution model supports an arbitrary deep nesting of calls to transactional futures in a top-level transaction.



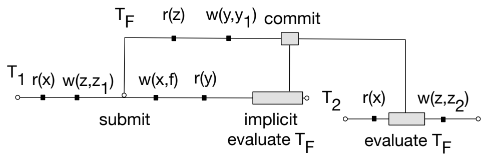
(a) A Simple Example of Transactional Futures (TF).


T_F2 r(x) r(y) commit
T_F1 r(n) w(x,x_1) commit
T_0 r(k) w(k,k_0) w(y,y_0) w(n,n_1)
submit
evaluate T_F1 evaluate T_F2

(b) Escaping TF that is evaluated within the same top-level transaction.



(c) Escaping TF across top-level transactions with GAC semantics.



(d) History of Figure 1c but with LAC semantics.

Figure 1. Example executions with Transactional Futures.

Note also that transactional futures are not required to be evaluated by the same transaction/thread that submit them.

3.1 A basic example

Figure 1a shows a simple example that we use to set the ground in our search for plausible semantics of transactional futures. The top-level transaction T first writes value 1 to variable x and then submits a transactional future T_F, which reads and increments x by 1. In parallel with T_F, i.e., before evaluating it, transaction T reads and increments x by 1. Finally, after evaluating T_F, T reads x and writes its value to variable y.

Given the simplicity of this scenario, it is intuitive to define both which sets of operations should be executed atomically and which are their admissible serialization orders: the read and write operations of T_F should *all* be serialized either before or after the operations of T that follow the creation of T_F and precede T_F's evaluation. We call this set of operations of T the continuation of T_F, and denote it as $C(T_F)$. Motivated by this example, we restrict a future T_F to appear as atomically executed with respect to its continuation $C(T_F)$, i.e., to be mutually isolated as if they ran encapsulated in two transactions. The choice of enforcing the atomicity between a future and its continuation aims at preserving the ease-of-use of TM, which represents arguably one of its main attractive features. Nevertheless, as we will discuss in Section 3.3, the proposed semantics allows for generating complex parallel computations that cannot be expressed using the conventional fork-join parallel nesting model.

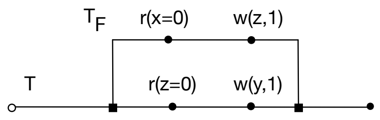


Figure 2. This continuation aborts with SO, but not with WO.

In this example, the serialization orders $T_F \rightarrow C(T_F)$ and $C(T_F) \rightarrow T_F$ provide the same outcome, since the operations of T_F and $C(T_F)$ commute. Since this is not the case in general, we argue that it is desirable to allow programmers to specify restrictions on the serialization order of transactional futures. These considerations led us to consider two different semantics regarding the serialization order of transactional futures:

• *Weakly Ordered Transactional Futures (WO)*: A future and its continuation should appear as executed atomically, i.e. the future should be serialized before or after its continuation.

• *Strongly Ordered Transactional Futures (SO)*: A future and its continuation should appear as executed atomically with the future serialized before its continuation.

The SO semantics ensures that a transactional future yields the same result as if it executed in a sequential version of the program (not using futures). WO semantics, conversely, require programmers to determine whether application's correctness is preserved independently of the order in which transactional futures and their continuations are serialized.

On the other hand, the ability of WO to establish different serialization points for a future brings two benefits:

• *Reduction of abort rate*: as exemplified in Figure 2, the continuation of T_F can be spared from aborting in case it misses to observe the updates produced by T_F (whereas the continuation would be aborted with SO semantics in such a history).

• *Stragglers avoidance*: with SO semantics, a transactional future, T_F^i, can only be committed if any previously submitted future, say T_F^j with $j < i$ (where the superscript denotes the spawning order of the transactional future), has first completed its own execution. As such, even a single relatively slow future can become a straggler for the whole set of futures concurrently submitted by the same top-level transaction. This phenomenon is exemplified in Figure 3, which illustrates a scenario in which a top-level transaction, logically composed by a total of 8 (commutative) sub-tasks, is parallelized using up to 3 concurrent futures, i.e., a new future is activated only whenever the continuation detects that a previously submitted future has completed its execution. The diagram clearly illustrates that, thanks to the use of WO semantics, the heterogeneity of the execution speed of transactional futures does not expose the system to the risk of stragglers.

Further, the choice of WO vs SO has implications on the definition of the upper bound of the execution interval of the *commit* operation of a future and of its spawning transaction.

With SO semantics the serialization order of a future is defined *prior* to the future's activation. As such, whenever a

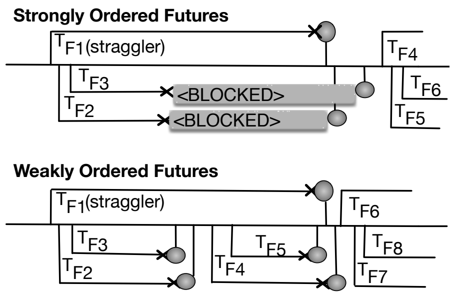


Figure 3. SO, unlike WO, suffers from stragglers.

SO future requests to commit, it is immediately possible to determine the outcome of the future (i.e., if this can be serialized upon submission) and return from the commit call. Conversely, the SO semantics demands that any future spawned by a transaction T is serialized before its continuation, i.e., within T. It follows that T's commit request has to be necessarily blocked until all the futures spawned by T have committed.

With WO semantics, the opposite is true: a transaction T that spawns a future T_F (and does not evaluate it) can commit without waiting for T_F, as T_F can be serialized upon evaluation, i.e., after T. However, whenever a future is serialized upon evaluation, the return call of its *commit* operation must follow the call of its *evaluate* operation (else, its serialization point would be undefined). Thus, the T_F's commit request may be blocked for an arbitrarily long time, i.e., until T_F is evaluated.

The latter case is illustrated in Figure 1a, which depicts the execution interval of the *commit* and *evaluate* operations of T_F (for simplicity all the other operations are assumed instantaneous). In this example, T_F requests to commit in real time before it is evaluated and assumes that the TM opted for serializing T_F upon evaluation. As such, the commit request of T_F has to be blocked until T_F is evaluated by T.

3.2 Non-blocking and repeated evaluations

So far we have assumed that a transactional future is evaluated at most once. The semantics we propose for the case of multiple evaluations are based on the common assumption that a transaction is only committed, and serialized, once. Analogously, repeated *evaluate* calls for T_F should be idempotent, i.e., always return the same result that corresponds to the result produced by the (only) execution of T_F that did commit.

This is done to guarantee that a transactional future, independently of the number of times it is evaluated, will be associated with a single serialization point, i.e., either upon its submission or, if *WO* semantics are assumed, upon its first evaluation.

A possible scenario in which a transactional future, say T_F, could be evaluated more than once occurs in case T_F is evaluated by a transaction T that later aborts. Upon its restart, T is likely to re-evaluate T_F. In such a case, T_F returns to a "non-evaluated" state and if T_F is later evaluated (e.g., by a reincarnation of T), its re-evaluation succeeds iff T_F can be serialized upon submission or in the new evaluation point.

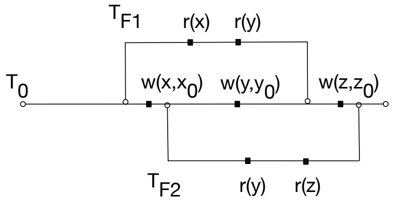


Figure 4. Concurrent computation not supported by parallel nesting.

Supporting a non-blocking variant of *evaluate* does not raise significant issues. In fact, any attempt to evaluate a future that is still executing has no impact on its possible serialization orders, as the call does not externalize the future's result.

3.3 Beyond parallel nesting

The simple example considered above could also have been implemented using parallel nesting [2] based on the classic fork-join model. However, futures support a broader class of concurrent computations than parallel nesting. Unlike parallel nesting, in fact, futures do not force blocking a "spawning" thread until *all* its nested sub-transactions complete their execution, but rather allow for the arbitrary interleaving of submissions and evaluations of different futures.

Figure 4 presents an example of a parallel computation that can be generated by futures but that cannot be expressed by parallel nesting: after submitting future T_{F1}, T_0 writes variable x in T_{F1}'s continuation, and before evaluating T_{F1}, it submits a second future T_{F2} and writes to y. Finally, T_0 writes variable z, evaluates T_{F2} and commits. Regarding atomicity between futures and continuations, we argue that in such a scenario the natural semantics is to enforce the atomicity of the writes to x and y by T_0 with respect to the operations issued by T_{F1}, i.e., either w(x,x_0) $\longrightarrow$ w(y,y_0) $\longrightarrow$ r(x) $\longrightarrow$ r(y), or r(x) $\longrightarrow$ r(y) $\longrightarrow$ w(x,x_0) $\longrightarrow$ w(y,y_0). Equivalently, T_{F2} should not be serialized between the write to y and the one to z by T_0, i.e., either w(y,y_0) $\longrightarrow$ w(z,z_0) $\longrightarrow$ r(y) $\longrightarrow$ r(z), or r(y) $\longrightarrow$ r(z) $\longrightarrow$ w(y,y_0) $\longrightarrow$ w(z,z_0). In this case, it is worth highlighting that the continuations of T_{F1} and T_{F2} are partially overlapping (they share the write to y), yet distinct.

In the following we focus on another type of executions that are allowed by the transactional futures and that cannot be supported with parallel nesting, which we call *escaping transactional futures*, i.e., transactional futures that are not evaluated by the same transaction in which they are submitted. As an example, consider an e-commerce application where adding an item to the cart triggers a transaction that updates the cart and, to hide user-perceived latency, it spawns a future to check for shipping costs using different sellers. This transaction commits before showing the next page to the user, but the future it generated is only evaluated at a later stage, when the purchase is finalized. The use of an escaping future provides two main advantages in this scenario. First, it reduces latency by overlapping the shipping cost computation (in the future) with the user's shopping interactions. Second, it ensures the atomicity of the whole purchase process, e.g., by aborting and restarting the future if the shipping cost of any item in the cart is modified by some transaction that commits before the future is evaluated.

We show an example of escaping futures in Figure 1b, in which T_{F2} is activated by T_{F1}. The latter writes to x, but commits without evaluating T_{F2}, whose reference is communicated to T_0 via T_{F1}'s return value. Next, T_0 issues a write on y and evaluates T_{F2}. This example highlights that the definition of continuation for escaping transactional futures is very subtle. We argue that, in this case, the "natural" continuation of T_{F2} (i.e., the sequence of causally-related operations that leads from the start of T_{F2}'s continuation to its evaluation) is composed by the write on x by T_{F1} *and* by the write on y by T_0, i.e. T_{F2}'s continuation spans two (sub-)transactions (associated with the same top-level transaction). Hence, T_{F2} should observe either both writes on x and y or none of them.

Figure 1c depicts another interesting programming pattern in which escaping transactional futures are used as a communication means by two distinct top-level transactions. In this case, the top-level transaction T_1 submits a future T_F. In T_F's continuation, T_1 writes a reference of the future returned by *submit*(T_F) to variable x, reads y and commits. With WO semantics, it is possible to serialize T_F after T_1. This allows T_1 to commit without having to block until T_F commits (see Section 3.1), making the reference to T_F available to other top-level transactions, e.g., T_2 in Figure 1c.

Following the same rationale used when analyzing Figure 1b, one may argue that by communicating the reference of T_F via x, a logical causality has been established between the operations issued by T_1 after submitting T_F and the operations issued by T_2 before evaluating T_F. If these operations were, in the light of this reasoning, considered as continuation of T_F, then these operations would have to appear as an atomic block to T_F, despite they span two top-level transactions. We term this atomicity model *Globally Atomic Continuation* (GAC).

The above example could be easily generalized, e.g., to include in the continuation of T_F, after T_1 and before T_2, an arbitrarily long chain of transactions propagating the reference to T_F to each other. As discussed in Section 3.1, with WO semantics, this would force a TM that opts for serializing T_F upon evaluation to stretch the execution interval of its commit operation for an arbitrarily long time, i.e., until T_F is evaluated. This can have implications both on the efficiency of TM implementations, which will have to maintain resources (e.g., locks held by the future) for prolonged periods of time, and on the likelihood for the T_F to be aborted (since by stretching its execution interval, T_F becomes more likely to conflict with concurrent transactions).

This led us to consider alternative atomicity semantics, called *Locally Atomic Continuation* (LAC), which limits the boundaries of a continuation to its spawning top-level transaction. With LAC semantics any top-level transaction T is requested to *implicitly* evaluate, during its commit phase, any escaping futures F that T spawned either directly or indirectly, i.e., T triggered the spawning of a chain of futures that led eventually to the submission of F. We call this evaluation "implicit", since it is not requested by programmers, but is rather imposed by the LAC semantics. The LAC semantics ensures that a future is necessarily serialized within its spawning top-level transaction: any future that escapes from its top-level transaction T is serialized either upon submission or (if WO semantics are assumed) upon its "implicit" evaluation, i.e., as the last (sub-)transaction of T right before T's commit. Note that, if multiple transactional futures escape from the same top level transaction, no constraint is imposed on the order in which they are implicitly evaluated by the TM.

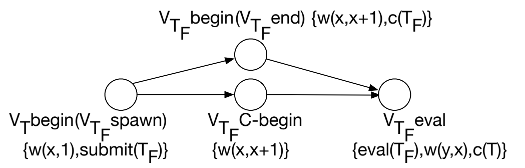
Figure 1d illustrates the LAC semantics for the same history considered in Figure 1c: an implicit evaluation of T_F is added as the last operation of its spawning top-level transaction T_1. As a consequence, T_F can commit earlier than if GAC semantics were considered. However, this comes at a cost for T_1, which is now forced to wait for T_F's completion before being able to commit. Figure 1d also shows an example of repeated evaluations of a future, namely T_F, which is first implicitly evaluated by T_1 and then (explicitly) evaluated by T_2. As discussed in Section 3.2, the second evaluation is required to return the same value as in the first evaluation.

Finally, let us discuss the relations between the proposed semantics, which were defined over two dimensions: the atomicity between transactional futures and continuations (GAC vs LAC), and their admitted serialization orders (WO vs SO). We note that the choice of SO semantics, which demand futures to be serialized at submission time, renders the distinction between globally and locally atomic continuations irrelevant: establishing which set of operations to include in a future's continuation is only relevant if futures can be serialized at their evaluation, i.e., after their continuation, as allowed by the WO semantics; with SO semantics, in fact, a future's serialization point is known *a priori* and is not affected by the set of the operations included in its continuation.

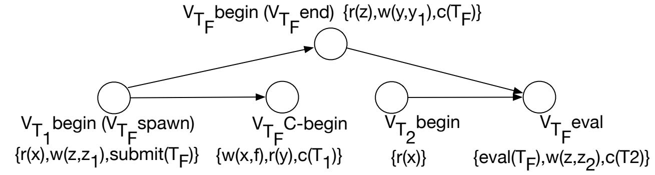
3.4 Formalizing the proposed semantics

In this section we introduce a framework to establish the set of feasible serialization orders among transactional futures and continuations. We propose a graph-based characterization, called Future Serialization Graph (FSG), that captures the possible serialization orders among transactional futures and continuations. The FSG formalization is similar in spirit to the DAG computation model used to capture task parallelism [6, 11]. A key distinction is that, since FSG incorporates transactions, a node in FSG corresponds to a sub-transaction as opposed to a strand (a sequence of instructions without parallel control constructs). Further, the FSG incorporates conflict relations between transactions.

Let $\mathcal{H}(\mathcal{T},\mathcal{S})$ be a history defined over: i) a set of transactions $\mathcal{T}=\mathcal{T}_{top}\cup\mathcal{T}_{fut}$, where $\mathcal{T}_{top}$ denotes the set of top-level

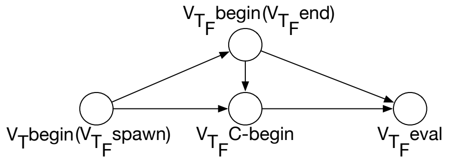


(a) FSG of the History in Fig. 1a (no ordering semantics).

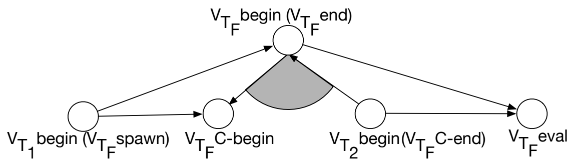


V_{T_F} begin (V_{T_F} end) {r(z),w(y,y_1),c(T_F)}
V_{T_1} begin (V_{T_F} spawn)
{r(x),w(z,z_1),submit(T_F)}
V_{T_F} C-begin
{w(x,f),r(y),c(T_1)}
V_{T_2} begin
{r(x)}
V_{T_F} eval
{eval(T_F),w(z,z_2),c(T2)}

(b) FSG of the history in Fig. 1c (no ordering semantics).



(c) Extending the FSG of the History in Fig. 1a with an edge from $V_{T_F}^{end}$ to $V_{T_F}^{C-begin}$ imposes SO semantics to T_F.



V_{T_F} begin (V_{T_F} end)
V_{T_1} begin (V_{T_F} spawn)
V_{T_F} C-begin
V_{T_2} begin(V_{T_F} C-end)
V_{T_F} eval

(d) Extending the FSG of the History in Fig. 1c with a bipath to impose WO semantics to T_F.

Figure 5. Example of FSG-based representations.

transactions and $\mathcal{T}_{fut}$ denotes the set of transactional futures; ii) a partial order $\mathcal{S}$ over the operations issued within each transactions in $\mathcal{T}$, extended to include real-time order relations between transactions. FSG($\mathcal{H}$) is defined as a directed graph having the following vertexes:

- A vertex V_T^{begin} for each transaction $T\in\mathcal{T}$, which is associated with all the operations executed by T since its begin until (and including) the first occurrence of the first of the following operations {*submit*,*evaluate*, *abort*, *commit*}.
- For each transactional future $T\in\mathcal{T}_{fut}$, two additional vertexes are defined: (1) $V_T^{C-begin}$, associated with the read-write operations issued by the thread that submitted T after T's submission (excluded) (i.e., the initial part of T's continuation) until the first occurrence of a {*submit*, *evaluate*, *abort*, *commit*}. This last operation is associated with $V_T^{C-begin}$, except if it is an *evaluate*, for which a dedicated vertex (V_T^{eval}, defined next) is added to the graph. (2) V_T^{eval}, associated with the operations issued by some thread that starts with the (possibly implicit) evaluation of the future T (included) and ends with the first occurrence of a {*submit*, *evaluate*, *abort*, *commit*} operation. Also in this case, this last operation is associated

with V_T^{eval}, except if it is an *evaluate*(T'), for which a $V_{T'}^{eval}$ vertex is added to the FSG.

The $V_T^{C-begin}$ and V_T^{eval} vertexes demarcate the logical boundaries of a continuation and serve as natural "checkpoints" to enable partial rollbacks, e.g., in Figure 1a, if the continuation is aborted due to a conflict with T_F, only the *sub-transaction* associated with the continuation is restarted and not the whole top-level transaction. To this end, when a *submit* or *evaluate* operation is executed by T, we implicitly commit the current sub-transaction of T and begin a new sub-transaction.

The following edges are defined for FSG($\mathcal{H}$):

- An edge $V1 \longrightarrow V2$ for each pair $V1$, $V2$ of vertexes in FSG such that $V1$ and $V2$ are executed by the same thread $Th_i \in \mathcal{TH}$ and Th_i executes $V1$ before $V2$ — which captures the sequential order of execution by a single thread.
- An edge $V_T^{spawn} \longrightarrow V_T^{begin}$ for each transactional future $T \in \mathcal{T}_{fut}$, where we have denoted with V_T^{spawn} the vertex associated with the operation *submit*(T), i.e., transactional futures cannot be serialized before their submission.
- An edge $V_T^{end} \longrightarrow V_T^{eval}$ for each transactional future $T \in \mathcal{T}_{fut}$, where we have denoted with V_T^{end} the vertex associated with the operation *commit*(T), i.e., transactional futures cannot be serialized after their evaluation.

Figure 5a shows the FSG of the example reported in Figure 1a and the operations associated with each vertex (within brackets). Note that in this example V_T^{begin} coincides with $V_{T_F}^{spawn}$ and $V_{T_F}^{begin}$ coincides with $V_{T_F}^{end}$. Figure 5b shows the FSG of the history in Figure 1c and in the supplemental material we provide also the FSG of the history in Figure 1d.

Weakly and strongly ordered futures. We now extend the FSG with additional edges aimed at imposing restrictions on the serialization order of a future with respect to its continuation, depending on the (strong vs weak) ordering semantics.

Recall that the SO semantics requires a future to be serialized upon submission. This semantics can be easily encoded in the FSG by adding an edge $V_T^{end} \longrightarrow V_T^{C-begin}$ for each SO transactional future $T \in \mathcal{T}_{fut}$, where we have again noted with V_T^{end} the vertex associated with the *commit* operation of T. Intuitively, this edge ensures that SO transactional futures are serialized before their respective continuations. This is illustrated in Figure 5c, where the edge from $V_{T_F}^{begin}$ (which coincides with $V_{T_F}^{end}$) to $V_{T_F}^{C-begin}$ is used to serialize T_F before its continuation.

WO, on the other hand, allows the serialization point of a transactional future to be either upon its submission or its evaluation. To enforce this constraint, we introduce a special type of edge, called *bipath* [29]. The notion of bipath was introduced by Papadimitrou in his seminal paper on the complexity of (view) serializability. A bipath is defined by a pair of edges $(V_i \longrightarrow V_j)$,$(V_k \longrightarrow V_l)$ and the inclusion of a bipath in a graph serves to express that either the first or the second edge of the bipath holds. The inclusion of a bipath in a directed graph, such as the FSG, turns the graph into a *polygraph*. A polygraph encodes a family of directed graphs, where each directed graph is obtained by including, for each bipath in the original polygraph, either one of its edges. As such, polygraphs allow for representing in a compact way a large number of directed graphs. More precisely, a polygraph with n bipaths encodes 2^n different directed graphs [29].

We capture the two alternative serialization orders of a WO transactional future by introducing for each $T_F \in \mathcal{T}_{fut}$ a bipath:

$$(V_{T_F{}^{C-end}} \longrightarrow V_{T_F{}^{begin}}),(V_{T_F{}^{end}} \longrightarrow V_{T_F{}^{C-begin}})$$

where $V_{T_F{}^{C-end}}$ represents the final vertex of the continuation of T_F, or more formally, the vertex associated with the operation that immediately precedes the evaluation of T_F in the (totally ordered) history of operations of the thread that evaluates T_F. With the above definition, the first edge of the bipath orders the continuation before the future (i.e., serialization upon evaluation) and the second edge orders the future before the continuation (i.e., serialization upon submission).

Figure 5d illustrates how the FSG for the history in Figure 1c is extended with a bipath that connects: (i) the end vertex of T_F's continuation (i.e., $V_{T_2}^{begin}$) to the beginning vertex of T_F (i.e., $V_{T_F}^{begin}$) and (ii) the end vertex of T_F (which coincides with $V_{T_F}^{begin}$) to the beginning vertex of $C(T_F)$ (i.e., $V_{T_F}^{C-begin}$).

Inclusion of operations in transactions. Before finalizing the formalization of the proposed semantics, we need to introduce the notion of inclusion of operations into a transaction. The intuition is to include in a transaction T not only the operations that T directly executes, but also the operations executed by transactional futures that are serialized within T possibly indirectly, i.e., via a chain of futures that can be rooted to a submit/evaluate operation by T. A key subtlety here is that our model supports escaping transactional futures, which, with WO semantics, are not a priori bound to be serialized within the spawning or the evaluating (top-level) transaction.

An operation op is included in a transaction T if there exists a path in the FSG from the vertex associated with the begin of T to the vertex associated with the commit/abort of T (or with the last operation issued by T if T is still active) that passes via the vertex associated with op. Based on this definition, we include in a transaction T all the operations issued by T and by any non-escaping future submitted by T and, recursively, by T's non-escaping futures. With WO semantics, an escaping future is given the flexibility to be either included in its spawning or evaluating transaction (depending on which edge of its bipath is considered in the FSG).

Extending the FSG with conflict relations. We now extend the FSG to capture conflict relations. Whenever a (sub-)transaction T executes an operation op that conflicts with an operation op' executed by (sub-)transaction T', the following edges are added to the FSG:

• *Atomicity within the same top-level transaction.* If T and T' are two (sub-)transactions included in the same top-level transaction, an edge is added from T to T', or in the opposite direction, depending on op is ordered before or after op'.

• *Atomicity between different top level transactions.* If T and T' are included in different top-level transactions, T_{top} and T'_{top}, an edge is added from *all* the vertexes associated with T_{top} to *all* the vertexes associated with T'_{top}, or in the opposite direction, depending on whether op is ordered before or after op'.

The rationale is that our FSG associates vertexes to individual (sub-)transactions and not to top-level transactions, as in classic transaction serialization graphs [29]. By adding edges among *all* the vertexes of two conflicting top-level transactions, we guarantee that the ordering relation induced by their conflicting operations is not only reflected at the level of the (sub-)transactions that issued those operations, but also among the corresponding top-level transactions.

Next, analogously to classical serializability theory [29], we impose the constraint that an operation can be accepted by a TM iff its execution does not generate cycles in the FSG. Else, T has to be aborted. Intuitively, this constraint guarantees the equivalence of the history $\mathcal{H}(\mathcal{T},\mathcal{S})$ produced by a TM to a sequential history that respects not only the partial order $\mathcal{S}$ of the operations in $\mathcal{H}$ but also of every additional constraint imposed by the semantics of weakly vs strongly ordered transactional futures and of globally vs locally atomic continuations.

Note that the proposed formalization requires that the absence of cycles in the FSG is ensured before allowing any operation issued by a transaction to return, and not only when a transaction is committed. In this sense, the proposed semantics for transactional futures are similar in spirit to existing safety criteria for TM systems that do not support futures, such as opacity [22], which aim at preventing active transactions, even those that eventually abort, from observing arbitrary snapshots not producible by a sequential execution of some subset of the committed transaction.

It should be noted that, if WO semantics are used, then the FSG is not a plain directed graph, but a polygraph that encodes a family of directed graphs. In this case, a history can be accepted iff there exists at least one directed graph encoded by the polygraph that contains no cycles. This definition of acyclicity of polygraphs, which coincides with the one used by Papadimitrou [29], captures the fact that for at least one of the set of viable serialization orders of the WO futures in the history (each encoded by a different directed graph), it is possible to prove the existence of an equivalent sequential history.

4 Overview of WTF-TM

In this section we introduce WTF-TM (Weakly ordered Transactional Futures), a STM that implements the weakly ordered semantics presented in Section 3. WTF-TM has been developed by extending JVSTM [7, 16], a multi-versioned JAVA STM The techniques used by WTF-TM to orchestrate the execution of transactional futures are largely orthogonal to the mechanisms used to regulate concurrency among top-level transactions. Thus, we abstract the mechanisms used to regulate concurrency among top-level transactions, and assume a plain multi-versioned STM that supports no intra-transaction parallelism.

We present WTF-TM assuming that no escaping futures exist and discuss how to avoid this assumption in Section 4.2.

4.1 Base algorithm

WTF-TM maintains, for each top-level transaction T, a graph, noted $\mathcal{G}$, that is used to track the logical dependencies among the transactional futures spawned or evaluated by T, either directly or indirectly, i.e., via other transactional futures spawned by T. Unlike the FSG (see Section 3.4), $\mathcal{G}$ does not use bi-paths to encode *all* possible serialization orders of futures. Conversely, $\mathcal{G}$ uses only simple directed edges and, as such, is a plain directed graph and not a polygraph. This is done for efficiency reasons, given the inherent cost of managing polygraphs at run-time. As a consequence of this design choice, WTF-TM may reject schedules that are admissible according to the formalized semantics. This represents a classic trade-off in the design of concurrency control schemes [20, 29].

$\mathcal{G}$ is consulted in two main occasions. First, when one of the sub-transactions of T, say T_{sub}, reads a shared variable V, in order to establish which version of V should be observed. To determine the visibility of versions, the $\mathcal{G}$ of T is used to retrieve the "ancestors" of T_{sub}, i.e., the sub-transactions of T that are serialized before T_{sub}. The version observed by T_{sub} is the one included in the write-set of the closest ancestor of T_{sub}, if any, or the version visible by the top-level T according to the underlying STM, if no ancestor wrote to V.

Second, when one of the sub-transactions of T, say T_{sub}, requests to commit. In this case T_{sub} executes a validation scheme aimed at enforcing that T_{sub} can be serialized, either at submission or at evaluation. We will discuss the validation mechanism more in detail shortly.

Each vertex of $\mathcal{G}$ is associated with a sub-transaction and has a corresponding *status*, which is initialized to *active*, when the vertex is created, and is updated to *iCommit* (internally committed), when the sub-transaction issues COMMIT().

During the execution of a sub-transaction, i.e., in its *active* state, all the updates it produced are buffered privately, i.e., they are not visible to any other sub-transaction. When a sub-transaction T' enters the *iCommit* state, its updates are made visible to the other sub-transactions of the same top-level transaction, say T. The updates of all the sub-transactions of T will become atomically visible to other top-level transactions (and to their sub-transactions) only when T is committed.

$\mathcal{G}$ is initialized when a top-level transaction T starts with a single "root" vertex. Every time a *submit*(T') operation is issued to spawn a transactional future T' two new vertexes are created, one corresponding to the future and the other to its continuation. These two vertexes are connected through two edges that depart from the vertex of the (sub-)transaction that spawned T'. To ensure that the write-set of a sub-transaction, T_s, that spawns a transactional future, T_f, is visible to T_f (as T_f is serialized after T_s in $\mathcal{G}$), sub-transactions are automatically *iCommitted* whenever they issue *submit*().

When an *evaluate*(T') operation is issued, a vertex V is created and linked via two edges originated, respectively, on the vertexes associated with the future T' and its continuation.

Commit logic. When a sub-transaction T requests to commit, WTF-TM attempts first to serialize T at submission time, which implies accessing $\mathcal{G}$ in order to: i) merge the write-set of T with the write-set of the sub-transaction that spawned T; ii) remove T's vertex from $\mathcal{G}$.

If T cannot be serialized at submission, T is not aborted. Its vertex in $\mathcal{G}$ is marked as *completed* (but not *iCommitted*, so its updates are invisible) until some sub-transaction T' issues EVAL(T). At that point, T' will attempt to serialize T at its evaluation point, which implies an analogous manipulation of $\mathcal{G}$: i) adding the write-set of T to the write-set of the sub-transaction that evaluated T; ii) remove T' vertex.

WTF-TM employs two validation mechanisms to determine if a sub-transaction can be serialized upon submission or evaluation, which we call *forward* and *backward* validation.

Forward validation. This mechanism is used to determine if a future can be serialized at submission time. Recall that whenever a sub-transaction T_s spawns a future T_f, T_s is automatically *iCommitted*. This guarantees that T_f always observe a snapshot that reflects the updates of its ancestors in $\mathcal{G}$. Serializing T_f at submission time, though, corresponds to moving the position of T_f's vertex in $\mathcal{G}$ and ordering it before the vertex of T_f's continuation, say T_c. A sufficient condition to ensure that this reordering neither affects T_c, nor the sub-transactions serialized after T_c , is that none of these sub-transactions has read any of the variables updated by T_f. To test this condition, WTF-TM navigates $\mathcal{G}$ moving "forward" from T_c and checking, for any reachable sub-transaction T_{sub}, if the write-set of T_f intersects with the read-set of T_{sub}.

Backward validation. This mechanism is used to determine if a future can be serialized at evaluation time. In this case, $\mathcal{G}$ is navigated backwards, starting from the vertex associated with the evaluating sub-transaction until reaching the sub-transaction that spawned T_f. All the sub-transactions along this path[1] have executed concurrently with T_f and their writes were not visible to T_f (since they were not among the ancestors of T_f). As such, T_f can only be reordered after these sub-transactions if they did not update any variable that T_f read.

[1]A single backward path exists since whenever a future T_F is serialized (upon evaluation or submission), T_F is removed from $\mathcal{G}$. This guarantees that $\mathcal{G}$ has no backward bifurcations.

Synchronizing the access to $\mathcal{G}$. The graph $\mathcal{G}$ is manipulated concurrently by the sub-transactions of the same top-level transaction. WTF-TM regulates these concurrent accesses via a mix of lock-based and lock-free techniques:

• *Updates of $\mathcal{G}$* that occur when sub-transactions start/commit (and are relatively infrequent) are regulated a read-write lock, acquired in write mode. $\mathcal{G}$ is also associated with a timestamp that is increased whenever $\mathcal{G}$ is updated and serves as a version counter to ensure the atomicity of traversals of $\mathcal{G}$.

• *Validation operations* synchronize with concurrent committing transactions by acquiring the $\mathcal{G}$ lock in read-mode.

• *Read operations* require establishing the set of ancestors of a sub-transaction in $\mathcal{G}$ and need to synchronize with concurrent updates of $\mathcal{G}$. Since reads are typically more frequent that commit operations, we avoid acquiring the read-write lock and use a lock-free synchronization approach: $\mathcal{G}$'s timestamp is read before and after traversing the $\mathcal{G}$ to extract the ancestors' list, repeating the traversal if the timestamp changes due to a concurrent update.

4.2 Escaping Transactional Futures

Let us discuss how to extend the base algorithm to manage escaping futures with both LAC and GAC semantics.

Locally Atomic Continuations. Ensuring these semantics implies guaranteeing that an escaping future can be serialized either at submission time or after any operation issued by its spawning top-level transaction. In order to ensure this property, whenever a top-level transaction T requests to commit, it needs to verify whether it spawned any future that is still active and has not been evaluated. In the latter case, T has to block until all such futures are committed.

Globally Atomic Continuations. With these semantics, an escaping future is not bound to be serialized within its spawning top-level transaction. This spares top-level transactions from blocking if, at commit time, there is any uncommitted escaping future. On the downside, if an escaping future T_f requests to commit after its spawning top-level transaction T_s commits, T_f looses the opportunity to serialize upon submission (as T_s's updates may have been in the meanwhile observed by other committed top-level transactions) and is bound to be serialized upon evaluation, within the $\mathcal{G}$ of a different top-level transaction T_e. In order to determine if the execution of T_f is compatible with this new serialization order, it is necessary to guarantee that the state observed by T_f during its execution is consistent at evaluation time. This can be ensured by validating the read-set of T_f and checking if it is still up to date at evaluation time, i.e., considering the updates produced by all the top-level transactions serialized before T_e and by all the sub-transactions of T_e serialized before the vertex associated to the evaluation of T_f in T_e's $\mathcal{G}$.

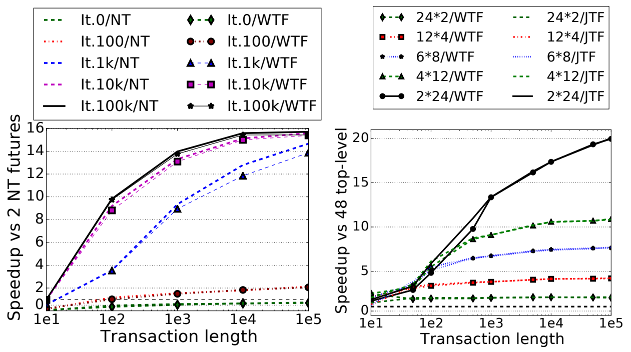


Figure 6. Left: speedup of WTF-TM (16 futures/2 top-level) vs 2 non-transactional (NT) futures in a read-only workload. Right: speedup of WTF-TM/JTF with different top-level*futures w.r.t. 48 top-level in a contended workload.

5 Evaluation

This section seeks answers to three key questions: i) what is the minimum transaction granularity for which it is using (WO) futures can lead to speedups? (Section 5.1); ii) how large is the overhead of WO w.r.t. SO and non-transactional futures? (Section 5.2); iii) what speedups does WO enable with respect to SO and to applications that do not use futures? (Section 5.3).

We focus the experimental study on the case of no escaping futures. Thus, we do not quantitatively evaluate the performance of the GAC vs LAC semantics, whose trade-offs were qualitatively discussed and illustrated in Section 3.3.

We compared the performance of WTF-TM with: i) The original implementation of JVSTM, which serves as a baseline not supporting futures; ii) JTF [44], which (see Section 3) supports TFs with SO semantics. The results are the average of five runs, executed on an Intel Xeon CPU E5-2660 v4 @ 2.00GHz, with 56 cores, 64GB of RAM, Ubuntu 16.04.6 LTS and OpenJDK 64-bit Server 1.7.0(build 19.0-b03).

5.1 When to use (WO) transactional futures?

To control the workload's characteristics in a predictable way, we use a synthetic benchmark that generates a configurable number of reads and writes to an array of 1M elements from within each transaction. Additionally, in between two memory accesses, CPU-bound computations are emulated by spinning for a configurable amount of iterations, indicated as *iter*.

We start by considering a read-only workload. In Fig. 6 (left), we plot the throughput using 2 top-level transactions, each parallelized with 16 futures, normalized with regard to the throughput running 2 top-level transactions without parallelization. We vary the number of read accesses from 10 to 100K on the x-axis. At the same time, we vary *iter* from 0 to 100K. The read location is selected uniformly at random across the whole array. It is worth noting that it is of no-use to employ any concurrency control, since the workload is read-only. Thus, by comparing the performance of WTF-TM and non-transactional futures, one can quantify the overhead due to enforcing WO semantics and the inherent costs of using futures (e.g., inter-thread communication for future's activation).

WTF-TM achieves close to ideal speedups when transactions are sufficiently long and contain enough CPU-bound computations (*iter*>1000). When we use a fully memory-bound workload (*iter*=0), increasing the degree of intra-transaction parallelism leads to negligible speedup even for transactions that execute 100K memory accesses.

Figure 6 (right) also shows that the performances of WTF-TM and of a non-transactional future implementation (Java's standard one) are quite close. This indicates that most of the overhead incurred by WTF-TM are inherent to the usage of futures and that scalability is not being limited by WTF-TM.

5.2 Overhead of WTF-TM with respect to JTF

To quantify the overhead of WTF-TM w.r.t. to JTF, we used a conflict-prone workload where WTF-TM can neither avoid aborts by serializing futures at evaluation nor benefit from avoiding stragglers. Specifically, futures execute a sequence of uniformly distributed read accesses over an array of 1 million elements, followed by ten update operations chosen uniformly at random from a (different) set of 20 "hot spot" items. Based on the findings of the last section, we set *iter* to 1k, to generate a workload where future-based parallelization can be beneficial.

The right plot in Fig. 6 shows the normalized throughput of WTF-TM and JTF w.r.t. to JVSTM. We fix the total number of threads to 48, which are allocated either to execute futures or top-level transactions. The notation $i * j$ means that we execute i top-level transactions, each parallelized via j threads.

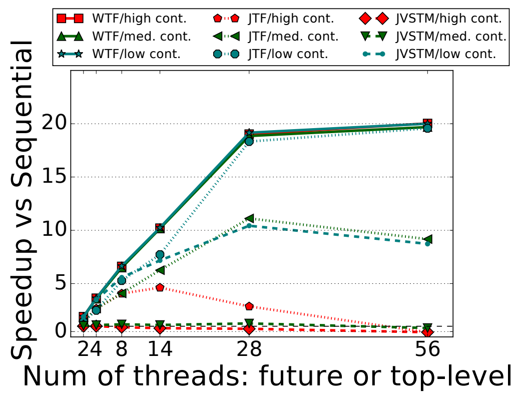
In this conflict-prone workload, the use of transactional futures, either with WO or SO semantics, reduces significantly the likelihood of conflicts between transactions as well as the cost of restarts — hence their speedups w.r.t. JVSTM.

Further, the performances of WTF-TM and JTF are almost indistinguishable, confirming that WTF-TM introduces very limited overhead with respect to JTF. The only exception is represented by the scenario of 2 top-level threads and 24 futures, where WTF-TM exhibits 10%-20% overhead when the length of read is below 500. We argue that this is due to the cost of synchronizing the manipulations of the graph structure used by WTF-TM to support WO semantics.

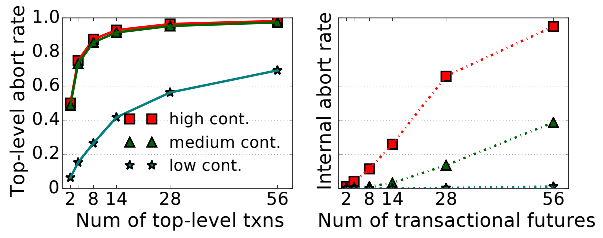
5.3 Gains of WTF-TM with respect to JTF

After showing that WTF-TM does not incur significant overhead w.r.t. to JTF even in unfavourable workloads, we consider two workloads where WTF-TM can outperform JTF by either avoiding aborts and/or avoiding stragglers.

Synthetic benchmark. Let us consider a workload where futures can conflict with their continuation. This causes the continuation to be aborted with SO semantics, whereas with WO the continuation's abort can be avoided by serializing its



(a) Speedup using multiple futures (WTF-TM, JTF) or top-levels (JVSTM) w.r.t. 1 top-level (JVSTM).



(b) Abort rate using JVSTM (left) or JTF (right).

Figure 7. Speedup (a) and abort rate (b) with different intra- and inter-transaction parallelism levels.

future upon evaluation. Each future performs 10K reads from an array of 1M elements. Then, it writes once to a number of randomly selected hot spots. Each continuation reads a random element from the hot spots and spawns a new future, until a given number of concurrent futures is reached. Finally, the top-level transaction evaluates all the futures it spawned (in spawning order) and commits. We set *iter* to 1k and vary contention by varying the hot spots' size: 100, 1K and 50K items.

In Fig. 7a we report, on the y-axis, throughput normalized w.r.t. executing top-level transactions (no futures) sequentially. We use the number of threads indicated on the x-axis as the number of concurrently spawned futures for JTF and WTF-TM, and as the number of concurrent top-level transactions for JVSTM. The worst performing baseline is the one that does not use futures, which incurs in the largest abort rates, see Figure 7b (left). In fact, when futures are not exploited to parallelize top-level transactions, these last longer and are, as such, more prone to conflict.

The throughput of JTF is strongly affected by the degree of contention. In the high contention scenario, JTF performance drops beyond 14 threads. This is explicable by analyzing the abort rate of futures and continuations, shown in Fig. 7b (right). Conversely, the performance of WTF-TM is unaffected by the contention level since the WO semantics allows for serializing futures upon evaluation, sparing them from *any* abort. As a result, WTF-TM achieves peak gains of up to 20× (high contention, 56 threads) w.r.t. JTF.

Bank benchmark. We evaluated the performance of WTF-TM using a benchmark, which we call *Bank*, that emulates replaying a log of operations from the daily records of a bank agency for backup or verification purposes. A similar benchmark has been frequently used to evaluate TM systems, e.g., [8, 18, 31, 37].

This workload consists of two operations *transfer* and *getTotalAmount*. As the name suggests, *transfer* moves money from a list of sending accounts to a list of receiving accounts, while *getTotalAmount* returns the total balance of all accounts within the bank. As all transfers are between accounts belonging to the same bank, *getTotalAmount* is expected to always return the same quantity and serves as a sanity check to detect errors during the backup/verification process.

We set the total number of accounts of the emulated bank to 100K and fix the number of (pairs of) accounts involved by *transfer* operations to 100, selected uniformly at random. Also in this case, we set *iter* to 1000. Note that, in these settings, *transfer* operations are significantly shorter than *getTotalAmount* operations — thus the latter operations are prone to straggle the former ones.

To parallelize this workload using transactions (without futures), we divide the log of operations to be replayed into fixed chunks. Each chunk is executed sequentially using a top-level transaction, and we vary the number of top-level transactions that execute concurrently.

When using futures, instead of executing a chunk sequentially, each operation in the log is delegated to a future spawned from within the same top-level transaction. We compare two variants of WTF-TM: *(i)* one that evaluates futures in the order they are spawned (WTF-TM-InOrder); and *(ii)* one that evaluates futures as soon as they complete execution (WTF-TM-OutOfOrder). The latter variant allows us to quantify the benefits of WTF-TM from avoiding straggling futures.

Figure 8 shows the speedup with respect to a sequential version and the internal abort rates for three workloads encompassing 10%, 50%, and 90% *transfer* operations (the remaining ones being *getTotalAmount*). From the plots, we can see that both WTF-TM variants outperform JTF in all workloads, achieving up to ~9× higher throughput. The abort plots show that WTF-TM variants incur significantly lower abort rates. In this workload, both *transfer* and *getTotalAmount* commute, therefore WTF-TM benefits from its ability to order futures at different serialization points, reducing abort rates. However, as the percentage of update operations increase, the aborts incurred by WTF-TM increase, lowering its speedup.

Finally, when comparing both variants of WTF-TM, we notice that evaluating futures out of order (to reduce the effect of stragglers) is always beneficial. As expected, the largest gains stemming from straggling avoidance (larger than 2×) are achieved in the 50% and 90% update scenarios, as in the 10% update scenario the slowest operations (i.e., *getTotalAmount*) are by far the most common operation executed by futures — thus limiting the actual straggling effect.

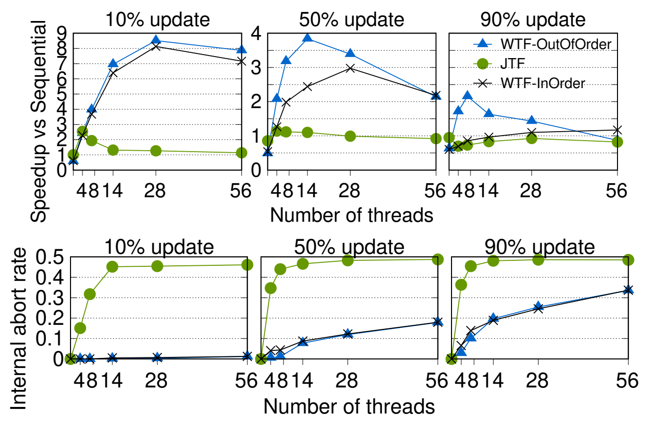


Figure 8. Throughput (top) and internal abort rate (bottom) for the Bank benchmark with 10%, 50% ,and 90% updates.

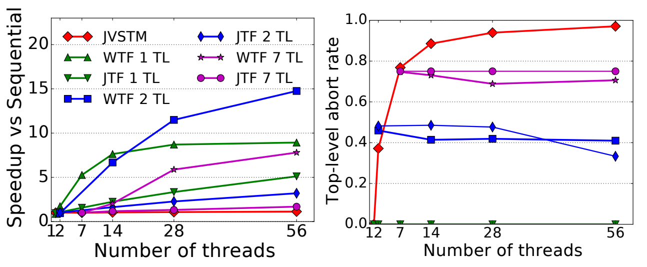


Figure 9. Vacation Benchmark: Speedup w.r.t. 1 top-level (left) and top-level abort rate (right) using different levels of intra- and inter-transaction parallelism.

Vacation benchmark. We adapted the Vacation benchmark (of the STAMP suite [9]), which emulates a travel agency to make use of transactional futures. We parallelized its *MakeReservation* transaction using futures, similarly to what was done in previous work [14, 17, 44]: each reservation consists of a fixed number of search operations within a database of flights, cars, and hotels; we divide these search operations among a fixed number of futures and emulate the scenario in which some search operations may have to access a remote database by injecting with 10% probability a delay of 100ms right after beginning a future.

Figure 9 (left plot) shows speedups w.r.t. executing top-level transactions (no futures) sequentially. The x-axis reports the total degree of parallelism, i.e., the number of active futures (1 for JVSTM) × the number of concurrent top-level transactions. For JTF, where futures are serialized in their spawning order, a new future is activated when the oldest spawned future completes; since WTF-TM supports out-of-order evaluation, a new future can be spawned as soon as *any* future completes.

WTF-TM outperforms all baselines achieving up to ~12× and ~4.5× speedups compared to JVSTM and JTF, respectively. The key reason for the gains of WTF-TM (and JTF) with respect to JVSTM (see Fig. 9, right plot) is the high probability of contention between top-level transactions and the high toll (wasted work) imposed by the abort of top-level transactions (recall that JVSTM does not exploit intra-transaction parallelism). Since this workload does not generate conflicts between futures, WTF-TM and JTF achieve a similar abort rate, which is significantly lower than with JVSTM (as futures reduce the duration of each top-level transaction, reducing both the cost and chance of aborting). However, WTF-TM scales much better than JTF, thanks to its ability to evaluate futures out of order and mitigate the bottlenecks due to straggling futures.

6 Conclusions and future work

This paper addressed the problem of combining futures and transactional memory by proposing a set of semantics that explore different trade-offs between simplicity and efficiency.

We also introduced WTF-TM, a STM that combines multi-versioning and graph-based concurrency control techniques. We evaluated WTF-TM across a number of diverse workloads. Our experimental results allowed us to shed lights on the scenarios in which transactional futures bring benefits and to quantify the performance trade-offs of the different semantics we proposed for transactional futures.

Futures represent a natural abstraction for asynchronous event-driven programming, which nowadays are increasingly popular in a range of domains, e.g., from web applications to stream processing. We hope that this work will pave the way to the development of a novel class of event-drivent applications, which will transparently support concurrent manipulations of shared state via the abstraction of transactional futures. The development of complex, real-life applications based on transactional futures would be greatly beneficial to research community for two main reasons. On the one hand, it would allow to quantify the reduction in complexity (e.g., development costs) stemming from the use of transctional futures with respect to conventional (e.g., lock-based) synchronization primitive — analogously to the studies that have demonstrated increased programmer's productivity thanks the use of classic TM [34]. On the other hand, it would provide a broader set of benchmarks directly inspired from real use case to evaluate the performance of future TMs with support for transactional futures — whereas in this work we had to resort to parallelize existing benchmarks that were not originally designed to use futures (e.g., STAMP's Vacation), or to develop rather simplistic synthetic benchmarks that did not fully exploit the richness of the proposed semantics (e.g., escaping futures).

Acknowledgments

We would like to thank our shepherd Kostis Sagonas, and the anonymous referees for their valuable comments. This work was partially supported by FCT, under project UIDB/50021/2020 and project Lisboa-01-0145- FEDER-031456 (Angainor), and KTH faculty funding.

References

[1] Atul Adya and Barbara H Liskov. 1999. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions.* Ph.D. Dissertation. Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science.

[2] Kunal Agrawal, Jeremy T. Fineman, and Jim Sukha. 2008. Nested Parallelism in Transactional Memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Salt Lake City, UT, USA) *(PPoPP '08)*. Association for Computing Machinery, New York, NY, USA, 163–174. https://doi.org/10.1145/1345206.1345232

[3] Utku Aydonat and Tarek S. Abdelrahman. 2008. Serializability of Transactions in Software Transactional Memory.

[4] Woongki Baek, Nathan Bronson, Christos Kozyrakis, and Kunle Olukotun. 2010. Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory. In *Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Thira, Santorini, Greece) *(SPAA '10)*. Association for Computing Machinery, New York, NY, USA, 253–262. https://doi.org/10.1145/1810479.1810528

[5] João Barreto, Aleksandar Dragojević, Paulo Ferreira, Rachid Guerraoui, and Michal Kapalka. 2010. Leveraging Parallel Nesting in Transactional Memory. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 91–100. https://doi.org/10.1145/1693453.1693466

[6] A. J. Bernstein. 1966. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers* EC-15, 5 (1966), 757–763. https://doi.org/10.1109/PGEC.1966.264565

[7] João Cachopo and António Rito-Silva. 2006. Versioned boxes as the basis for memory transactions. *Science of Comp. Prog.* 63, 2 (2006), 172–185.

[8] Daniel Castro, Paolo Romano, and João Barreto. 2019. Hardware Transactional Memory meets memory persistency. *J. Parallel and Distrib. Comput.* 130 (2019), 63 – 79. https://doi.org/10.1016/j.jpdc.2019.03.009

[9] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*. IEEE, 35–46. https://doi.org/10.1109/IISWC.2008.4636089

[10] clojure.org. 2019. Concurrent Programming in Clojure. https://clojure.org/about/concurrent_programming.

[11] Edward G. Coffman and Peter J. Denning. 1973. *Operating Systems Theory.* Prentice Hall Professional Technical Reference.

[12] Dave Dice, Maurice Herlihy, and Alex Kogan. 2018. Improving Parallelism in Hardware Transactional Memory. *ACM Trans. Archit. Code Optim.* 15, 1, Article 9 (March 2018), 24 pages. https://doi.org/10.1145/3177962

[13] Dave Dice, Yossi Lev, Yujie Liu, Victor Luchangco, and Mark Moir. 2013. Using Hardware Transactional Memory to Correct and Simplify and Readers-Writer Lock Algorithm. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Shenzhen, China) *(PPoPP '13)*. Association for Computing Machinery, New York, NY, USA, 261–270. https://doi.org/10.1145/2442516.2442542

[14] Nuno Diegues and João Cachopo. 2013. Practical Parallel Nesting for Software Transactional Memory. In *Proceedings of the 27th International Symposium on Distributed Computing - Volume 8205* (Jerusalem, Israel) *(DISC 2013)*. Springer-Verlag, Berlin, Heidelberg, 149–163. https://doi.org/10.1007/978-3-642-41527-2_11

[15] Nuno Diegues, Paolo Romano, and Luís Rodrigues. 2014. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT '14)*. Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/2628071.2628080

[16] Sérgio Miguel Fernandes and João Cachopo. 2011. Lock-Free and Scalable Multi-Version Software Transactional Memory. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (San Antonio, TX, USA) *(PPoPP '11)*. Association for Computing Machinery, New York, NY, USA, 179–188. https://doi.org/10.1145/1941553.1941579

[17] Ricardo Filipe and Joao Barreto. 2015. Nested Parallelism in Transactional Memory. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications.* Springer, 192–209.

[18] Wilson W. L. Fung, Inderpreet Singh, Andrew Brownsword, and Tor M. Aamodt. 2011. Hardware Transactional Memory for GPU Architectures. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (Porto Alegre, Brazil) *(MICRO-44)*. Association for Computing Machinery, New York, NY, USA, 296–307. https://doi.org/10.1145/2155620.2155655

[19] Vincent Gramoli and Rachid Guerraoui. 2014. Democratizing Transactional Programming. *Communications of ACM* 57, 1 (Jan. 2014), 86–93. https://doi.org/10.1145/2541883.2541900

[20] Vincent Gramoli, Derin Harmanci, and Pascal Felber. 2010. On the Input Acceptance of Transactional Memory. *Parallel Processing Letters* 20 (03 2010). https://doi.org/10.1142/S0129626410000041

[21] Rachid Guerraoui, Thomas A Henzinger, and Vasu Singh. 2008. Permissiveness in transactional memories. In *International Symposium on Distributed Computing*. Springer, 305–319.

[22] Rachid Guerraoui and Michal Kapalka. 2007. *Opacity: A correctness condition for transactional memory.* Technical Report. EPFL.

[23] Robert H. Halstead. 1985. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems* 7 (1985), 501–538.

[24] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '93)*. Association for Computing Machinery, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[25] Alex Kogan and Maurice Herlihy. 2014. The Future(s) of Shared Data Structures. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing* (Paris, France) *(PODC '14)*. Association for Computing Machinery, New York, NY, USA, 30–39. https://doi.org/10.1145/2611462.2611496

[26] Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased transactional memory.

[27] Oracle. 2019. Javadoc of the Future Interface in JAVA (2019). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html.

[28] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. A Study of Transactional Memory vs. Locks in Practice. In *Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures* (San Jose, California, USA) *(SPAA '11)*. Association for Computing Machinery, New York, NY, USA, 43–52. https://doi.org/10.1145/1989493.1989500

[29] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. https://doi.org/10.1145/322154.322158

[30] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. 2008. Dependence-Aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, USA, 246–257.

[31] Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. 2007. From Causal to Z-Linearizable Transactional Memory. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '07)*. Association for Computing Machinery, New York, NY, USA, 340–341. https://doi.org/10.1145/1281100.1281162

[32] Paolo Romano, Roberto Palmieri, Francesco Quaglia, Nuno Carvalho, and Luis Rodrigues. 2014. On speculative replication of transactional systems. *J. Comput. System Sci.* 80, 1 (2014), 257–276.

[33] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. 2010. Is Transactional Programming Actually Easier?. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Bangalore, India) *(PPoPP '10)*. Association for Computing Machinery, New York, NY, USA, 47–56. https://doi.org/10.1145/1693453.1693462

[34] Christopher J Rossbach, Owen S Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier?. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 47–56.

[35] rust lang.org. 2019. Zero-cost Futures in Rust. https://docs.rs/futures/0.1.27/futures/.

[36] Nir Shavit and Dan Touitou. 1997. Software transactional memory. *Distributed Computing* 10, 2 (1997), 99–116.

[37] Konrad Siek and Paweł Wojciechowski. 2015. Atomic RMI: A Distributed Transactional Memory Framework. *International Journal of Parallel Programming* 44 (04 2015). https://doi.org/10.1007/s10766-015-0361-x

[38] Michael F. Spear, Wenjia Ruan, Yujie Liu, and Trilok Vyas. 2015. Case Study: Using Transactions in Memcached. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications - COST Action Euro-TM IC1001*. Springer, 449–467. https://doi.org/10.1007/978-3-319-14720-8_20

[39] Janwillem Swalens, Joeri De Koster, and Wolfgang De Meuter. 2018. Chocola: Integrating Futures, Actors, and Transactions. In *Proceedings of the 8th ACM SIGPLAN International Workshop on Programming Based on Actors, Agents, and Decentralized Control* (Boston, MA, USA) *(AGERE 2018)*. Association for Computing Machinery, New York, NY, USA, 33–43. https://doi.org/10.1145/3281366.3281373

[40] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. 2009. NePalTM: design and implementation of nested parallelism for transactional memory systems. In *ECOOP*. Springer, 123–147.

[41] Adam Welc, Suresh Jagannathan, and Antony Hosking. 2005. Safe Futures for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 439–453. https://doi.org/10.1145/1094811.1094845

[42] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-Performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '13)*. Association for Computing Machinery, New York, NY, USA, Article 19, 11 pages. https://doi.org/10.1145/2503210.2503232

[43] Pantea Zardoshti, Tingzhe Zhou, Pavithra Balaji, Michael L. Scott, and Michael F. Spear. 2019. Simplifying Transactional Memory Support in C++. *TACO* 16, 3 (2019), 25:1–25:24.

[44] J. Zeng, J. Barreto, S. Haridi, L. Rodrigues, and P. Romano. 2016. The Future(s) of Transactional Memory. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 442–451. https://doi.org/10.1109/ICPP.2016.57

[45] Zeng, Kun. 2010. Conflict graph based hardware transactional memory. In *2010 3rd International Conference on Computer Science and Information Technology*, Vol. 5. IEEE, 496–501. https://doi.org/10.1109/ICCSIT.2010.5563895

[46] T. Zhou, P. A. Zardoshti, and M. Spear. 2017. Practical Experience with Transactional Lock Elision. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, 81–90. https://doi.org/10.1109/ICPP.2017.17

A Artifact Appendix

A.1 Abstract

Our artifact includes the source code for Weakly ordered Transactional Futures (WTF), together with source code of two baselines, namely Strongly ordered Transactional Futures (JTF) [44] and the underlying transactional memory, JVSTM [7]. We have included in the artifact a Java Virtual Machine (JVM) with First Class Continuation support provided, which is needed to run JTF and JDK 1.7. JDK 1.7 is required for compilation. To ensure a fair comparison, we use the same JDK and JVM for compiling and running our source code also. The only software required to run this artifact is Apache Ant on top of Linux x86-64 OS.

A.2 Artifact check-list (meta-information)

The artifact consists of the following directories:

- **Compilation: Apache Ant build tool, JDK 1.7 (available in the artifact)**
- **Run-time environment: Linux x86-64**
- **Metrics: Throughput, Abort Rate**
- **How much disk space required (approximately)?: 100 MBytes**
- **How much time is needed to prepare workflow (approximately)?: 60 minutes**
- **How much time is needed to complete experiments (approximately)?: depends on contention level, subsecond when there is no contention, 30 minutes for one experiment under high contention**
- **Archived: The artifact is available at the following URL: https://doi.org/10.5281/zenodo.4323407**

A.3 Description

- *src*
  - *WTF: source code for Weakly ordered futures (the main contribution of this paper).*
  - *JTF: source code for Strongly ordered futures (JTF[44]).*
- *jdk1.7.0_80: java 1.7 (needed for the compilation of JTF [44])*
- *openjdk-continuation-vm2013-linux-amd64: JVM with continuation support (First Class Continuation support provided by the OpenJDK Hotspot VM).*
- *raw-results: raw results for the vacation experiment (fig. 9 in the paper).*
- *scripts:*
  - *run-vacation.sh: script that launches the experiment for Vacation benchmark (Fig. 9 in the paper).*
  - *plot-normalized-throughput.py: script that plots the normalized throughput for the data inside raw-results (Fig. 9 in the paper, right).*
  - *plot-abort-rate.py: script that plots the abort rate for the Vacation benchmark (Fig. 9 in the paper, right).*

A.3.1 How to access. *Refer to subsection A.5*

A.3.2 Hardware dependencies. Any multi-core machine which runs JDK 1.7.

A.3.3 Software dependencies. Linux x86-64, JDK 1.7, Apache Ant

A.4 Installation

Compile the source code:

```
cd ROOT_FOLDER
cd src/JTF
ant compile
cd ROOT_FOLDER
cd src/WTF
ant compile
```

A.5 Experiment workflow

To run a simple example:

```
cd ROOT_FOLDER
./openjdk-continuation-vm2013-linux-amd64/bin/java \
  -cp src/WTF/build/classes/ benchmark.vacation.Vacation \
  -c 1 -n 2016 -q 1 -u 98 -r 105 -t 4 -nest true -sib 4 -updatePar false\
  -readOnly false -unsafe false -max\_num\_core 560 \
  -streamingEnabled 3 -stragglerProbability 100 \
  -delayDuration 100
```

An output similar to the following one should be observed:

```
Nov 29, 2020 12:29:37 AM jvstm.ActiveTransactionsRecord <clinit>
INFO: ***** AOM reversion = false (disable/enable it in property jvstm.aom.reversion)
Execution\_Time\_Millis: 6916
throughput: 0.578368999421631
abort\_rate: 0.0
internal\_abort\_rate: 0.0
6916 4 0 0 0;
```

A.6 Evaluation and expected results

To reproduce the vacation experiment (Fig. 9 in the paper):

```
cd ROOT_FOLDER
./scripts/run-vacation.sh
```

Parameters that we varied in this experiemtn:

- -c $C: number of [c]lients, i.e., possible concurrent top-level transactions (1, 2, 7 for WTF/JTF(corresponding to different lines/legend in the plot), 1,2,7,14,28,56 for JVSTM)
- -sib $SIB: Number of thread divided by $C.Note number of thread corresponding to the numbers in the x-axis in Figure 7.
- -nest $NEST: true (WTF and JTF) to use futures or false (JVSTM) for top-level transactions only.

Other parameters:

- -n: 20160 [n]umber of user queries per transaction, it can be evenly divided among 560 futures.
- -q: 1% Percentage of relations [q]ueried, and 1% indicates high conflict rate between top-level transactions.
- -u: 98% for Percentage of [u]ser transactions to be MakeReservation Transaction.
- -r: Number of possible [r]elations
- -t: 10485 for Number of [t]ransactions.
- -max_num_core: 560
- - streamingEnabled: 3 coding for Out-of-order streaming strategy which let futures be evaluated as soon as it finished executing.
- - stragglerProbability: 100 (which corresponds to 100/1000, i.e., 10%)
- - delayDuration: 100ms (each spawned future is injected with 10% probability a delay of 100ms right after beginning a future)

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html