Brief Announcement: Giving Future(s) to Transactional Memory

Jingna Zeng
KTH Royal Institute of Technology
University of Lisbon

Seif Haridi
KTH Royal Institute of Technology

Shady Issa, Paolo Romano, Luís Rodrigues
INESC-ID/University of Lisbon

ABSTRACT

This paper extends the Transactional Memory (TM) paradigm by proposing a new powerful abstraction, the *transactional future*. Transactional futures, as the name suggests, combine TM with futures, by 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 main contribution of this paper is the definition of a set of semantics for transactional futures that explore different trade-offs between simplicity and efficiency.

CCS CONCEPTS

• **Theory of computation → Abstraction**; • **Computing methodologies → Parallel algorithms**.

KEYWORDS

Transactional Memory; Parallel Programming; Futures; Synchronization; Concurrency Control

ACM Reference Format:
Jingna Zeng, Seif Haridi, and Shady Issa, Paolo Romano, Luís Rodrigues. 2020. Brief Announcement: Giving Future(s) to Transactional Memory. In *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '20), July 15–17, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3350755.3400220

1 INTRODUCTION

Transactional Memory (TM) [9, 14] is an attractive paradigm for parallel programming that relies on the familiar abstraction of transaction to shift the burden of coordinating concurrent access to shared memory from programmers to the run-time environment, possibly leveraging ad hoc hardware supports [3–5, 7, 10–13, 16, 17, 20].

This work aims at filling an important gap in the literature on TM by investigating how to extend the TM model to support a popular abstraction for parallel programming, namely futures (a.k.a., promises [8]). In fact, despite the wide body of research that addressed various aspects of the TM domain (e.g., supporting intra-transaction using parallel nesting [1][6][15][2]), the problem of how to integrate the abstractions of futures and TM remains largely unaddressed at current date, to the best of our knowledge.

We pursue this goal by introducing the abstraction of *transactional futures*, i.e., transactions that execute wrapped within futures and that are spawned and evaluated by other 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. Based on these considerations, we propose four different semantics that regulate two different dimensions: the degree of atomicity between futures and continuations, and their admitted serialization orders. These alternative semantics 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 different constraints on the serialization order of transactional futures).

The semantics proposed in this paper extend the ones proposed in our previous work [18], where we proposed a simplistic model for what concerns the serialization order of transactional futures (called *strongly ordered* in this paper) that imposes the serialization of futures at their submission point. In this paper, we investigate multiple definitions of the isolation and atomicity semantics of transactional futures, which raise interesting opportunities as well as subtle issues. In the full version of this paper [19], we also present a graph-based formalization and software-based implementation of the semantics herein proposed.

2 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 $\mathcal{TH} = \{Th_1, \ldots, Th_n\}$ 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. To integrate futures and transactions, we allow the latter to return values: in fact, 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.

Further, we allow transactions to issue, besides reads and writes, two additional operations: *submit* and *evaluate*. These two primitives allow, respectively, for submitting and evaluating a transactional future, i.e., a transaction encapsulated in a future that can run in parallel with the thread that submitted it. We assume, for simplicity, that future submission and evaluation can only be done within the context of a transaction.

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 object f can then be passed

SPAA '20, July 15–17, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-6935-0/20/07.
https://doi.org/10.1145/3350755.3400220

as an input parameter to the *evaluate* operation, which eventually returns the value generated by the transaction. 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. It is easy to see that this transaction execution model supports an arbitrary deep nesting of calls to transactional futures in a top-level transaction. Also, a transactional future T_F can be uniquely associated with one top-level transaction T_s within whose context T_F is submitted, and with one or more top-level transaction T_e within whose context T_F is evaluated. Importantly, note that our model does not require transactional futures to be evaluated by the same transaction/thread that submit them, i.e. possibly $T_s \neq T_e$.

2.1 A Basic Example

Figure 1a illustrates a simple example that allows us to set the ground in our search for plausible semantics of execution 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 set the value of X to 10. 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 $C(T_F)$.

We consider two different semantics regarding the plausible serialization order of transactional futures and their continuations

- *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 Figure 1a also 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 execution, the future requests to commit in real time before the future is evaluated; the return of the future's *commit* operation is placed after the call to *evaluate*, and before the latter operation returns. In general, though, the timing of the call/return events of the *commit* and *evaluate* operations of future vary.

Note that, if a future is evaluated before it completes its execution or requests to commit, its evaluation will block until the future has been successfully committed. This is analogous to what happens when one attempts to evaluate a plain/non-transactional future that has not completed executing. A related observation is that the choice of WO vs SO semantics has important implications on the definition of the upper bound of the execution interval of the *commit* operation for a future. With SO semantics, in fact, the serialization order of a future is defined *prior* to the future's activation. As such, whenever a SO future requests to commit (and there are no previously spawned futures), it is immediately possible to determine the outcome of the future, i.e., whether this can be serialized upon its submission. As such, if the future in Figure 1a were to abide by SO semantics, the return of its commit request could be also placed before the call of its *evaluate* operation.

This is not always the case, though, with WO semantics. In fact, whenever a WO future is serialized upon evaluation, the return call of its *commit* operation must necessarily follow, in real time, the call of its *evaluate* operation. Consider a case, such as the one illustrated in Figure 1a, in which a WO future requests to commit before being evaluated. Assume further that the underlying TM implementation opts for serializing the future upon evaluation — e.g., because the future wrote some data item that its continuation read without observing the future's write, thus, the continuation would have to abort if the future had to be serialized upon submission. In such a scenario, given that the future has not been evaluated yet, it would be impossible at that point in time for *any* TM implementation to determine whether the future can be committed or not.

Thus, if on the one hand WO semantics provide more flexibility in defining the serialization points of a future, this flexibility comes at a cost: in order for the future to be serialized upon evaluation, the future's commit request may have to be blocked for an arbitrarily long time, i.e., up until the future is actually evaluated.

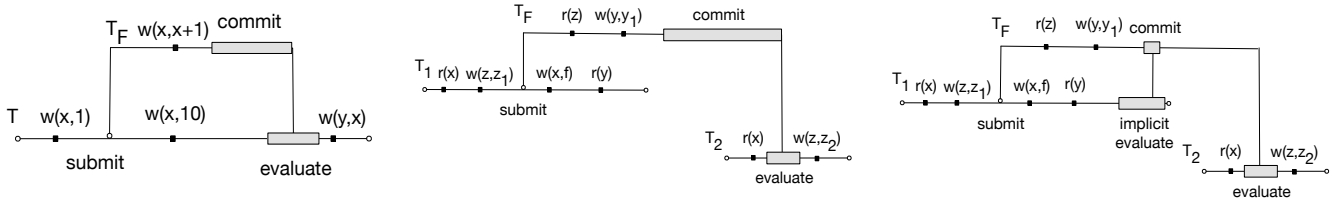
2.2 Escaping futures

We now focus on a specific type of executions that are allowed by the assumed execution model of transactional futures, but that cannot be supported with parallel nesting: *escaping transactional futures*, i.e., transactional futures that are not evaluated by the same transaction in which they are submitted.

We show an example of escaping futures in Figure 1b. Here, an escaping transactional future is used as a complementary communication means (in addition to shared variables) 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 the reference of the future returned by *submit*(T_F) to variable X, reads Y not observing T_F's write and requests to commit. With WO semantic, it is possible to serialize T_F after T_1. This allows the top-level transaction T_1 to commit without having to block waiting for T_F to commit first[1], making the reference to T_F available to other top-level transactions possibly executing on different threads, e.g., T_2 in Figure 1b.

Globally Atomic Continuation. One may argue that by communicating the reference of T_F via X, a logical causality has been established between the write operations to X by T_1 and the read operation for X issued by T_2 before evaluating T_F. Despite spanning two top-level transactions, these operations represent a chain of causally related events that led to the evaluation of T_F. If they were, in the light of this reasoning, considered as the continuation of T_F,

[1]This would be necessary if we assumed SO semantic for T_F. In this case, it cannot be otherwise guaranteed that T_1 observes all the writes possibly issued by T_F unless T_F completes its execution.



(a) A Simple Example of Transactional Futures

(b) Escaping Transactional Future with GAC semantic

(c) Escaping TF across with LAC semantic

Figure 1: Example executions of Transactional Futures.

then they should intuitively appear as an atomic block to T_F. We term this atomicity model *Globally Atomic Continuation* (GAC).

Locally Atomic Continuation. The above example could be easily generalized 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 already discussed in Section 2.1, though, this may lead to stretching the execution interval of T_F arbitrarily, which may be undesirable for efficiency reasons.

This leads us to consider an alternative semantic, called *Locally Atomic Continuation* (LAC), which limits the boundaries of a continuation to its spawning top-level transaction. With LAC semantic any top-level transaction T is requested, during its commit phase to *implicitly* evaluate any escaping futures F it 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 explicitly by programmers but is rather imposed by the considered atomicity semantic.

The LAC semantic ensures that a future is serialized within its spawning top-level transaction. In fact, with LAC, a future that escapes from its top-level transaction T is serialized either upon submission or (if WO semantic is used) upon its "implicit" evaluation, i.e., as the last (sub-)transaction of T right before T's commit.

Figure 1c illustrates the LAC semantics for the same history of Figure 1b: 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 much earlier than if GAC semantics were considered. Note, though, that this come at a cost for T_1, which is now forced to wait for T_F's completion before being able to commit.

Figure 1c 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. The semantics we propose for repeated evaluations of a future is that only the first *evaluate* determines the serialization point of transactional futures (assuming WO). Further evaluations just return the results the future produced during its first evaluation. This definition follows the common sense that a transaction is only committed (and, hence, serialized) once.

3 CONCLUSIONS

This work shed lights on the challenges of defining semantic models capable of reconciling the future's abstraction with the transactional memory paradigm. We showed that the key complexity lies in that futures allow for defining complex concurrency patterns, for which it is not how to define adequate isolation and atomicity guarantees.

We analyzed a set of concurrent programming patterns that can be enabled via the future's abstraction, and used these examples to motivate a set of possible semantics for transactional futures that explore different trade-offs between simplicity and efficiency.

REFERENCES

[1] Kunal Agrawal, Jeremy T Fineman, and Jim Sukha. 2008. Nested parallelism in transactional memory. In *PPoPP*. 163–174.

[2] Woongki Baek and Christos Kozyrakis. 2009. NesTM: Implementing and Evaluating Nested Parallelism in Software Transactional Memory. In *PACT* (Raleigh, NC, USA).

[3] Dave Dice, Maurice Herlihy, and Alex Kogan. 2018. Improving Parallelism in Hardware Transactional Memory. *TACO* 15, 1 (2018), 9:1–9:24. https://doi.org/10.1145/3177962

[4] 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 *PPoPP*. 261–270.

[5] Nuno Diegues, Paolo Romano, and Luís E. T. Rodrigues. 2014. Virtues and limitations of commodity hardware transactional memory. In *PACT*. 3–14.

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

[7] Vincent Gramoli and Rachid Guerraoui. 2011. Democratizing transactional programming. In *Middleware 2011*. Springer, 1–19.

[8] Robert H Halstead Jr. 1985. Multilisp: A language for concurrent symbolic computation. *ACM TOPLAS* 7, 4 (1985), 501–538.

[9] Maurice Herlihy and J Eliot B Moss. 1993. *Transactional memory: Architectural support for lock-free data structures*. Vol. 21. ACM.

[10] Shady Issa, Pascal Felber, Alexander Matveev, and Paolo Romano. 2019. Extending hardware transactional memory capacity via rollback-only transactions and suspend/resume. *Distributed Computing* (2019), 1–22.

[11] Yossi Lev, Mark Moir, and Dan Nussbaum. 2007. PhTM: Phased transactional memory. In *Workshop on Transactional Computing (Transact)*.

[12] Victor Pankratius and Ali-Reza Adl-Tabatabai. 2011. A study of transactional memory vs. locks in practice. In *SPAA*. 43–52.

[13] Christopher J Rossbach, Owen S Hofmann, and Emmett Witchel. 2010. Is transactional programming actually easier?. In *PPoPP*. 47–56.

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

[15] 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.

[16] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *SC*. IEEE, 1–11.

[17] 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.

[18] Jingna Zeng, Joao Barreto, Seif Haridi, Luís Rodrigues, and Paolo Romano. 2016. The Future (s) of Transactional Memory. In *ICPP*. 442–451.

[19] Jingna Zeng, Shady Alaaeldin Issa, Seif Haridi, Luis Rodrigues, and Paolo Romano. 2019. *Investigating the Semantics of Futures in Transactional Memory Systems*. Technical Report. INESC-ID, Lisbon, Portugal.

[20] Tingzhe Zhou, Pantea Zardoshti, and Michael F. Spear. 2017. Practical Experience with Transactional Lock Elision. In *ICPP*. 81–90.