1. INTRODUCTION

Transactional memory (TM) [Herlihy and Moss 1993; Shavit and Touitou 1997] is one of the most promising paradigms to synchronize concurrent access to shared state. An extensive amount of work in the area of TM has focused on transaction execution models that assume that the operations of a transaction are issued sequentially. However, there are many cases where it may be beneficial to exploit concurrency within a single transaction. This has motivated the study of abstractions that allow programmers to express parallelism in the context of transactional systems.

There are two abstractions that are extremely relevant in this context. One is the abstraction of parallel nesting, where multiple sub-transactions, that are part of an larger outer (also called top-level) transaction, can be launched in parallel, using language constructs such as cobegin/coend, parallelfor, etc. The other is the future [Halstead Jr 1985] abstraction, which lets programmers activate a parallel task and return a future contract, i.e., a promise to deliver the result of the task when the contract is evaluated.

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) and when this task needs to be completed (i.e., when the future is evaluated). Differently from the parallel nesting model, however, the future abstraction does not block the execution of the main thread, also called continuation in the context of futures, till the completion of nested parallel transactions. Further, futures are not necessarily evaluated by the thread that created them. Also, a thread may create a future f and, in the continuation of f evaluate a different future f' (without having first evaluated f). Overall, futures are able to generate a wider range of concurrent computations and enable programming patterns not supported by parallel nesting.

The abstraction of Futures has been studied for type-specific optimization on long-lived shared data structures [Kogan and Herlihy 2014] and language integration [Welc et al. 2005]. Interestingly, while the combination of parallel nesting and and transactions has been extensively studied in the literature [Barreto et al. 2010; Barreto et al. 2012; Diegues and Cachopo 2013; Agrawal et al. 2008; Baek et al. 2010], we are not aware of works on the combination of futures and TMs — an abstraction which we call Transactional Futures.

In this work, we focus on the problem of defining desirable atomicity and isolation semantics for systems in which futures are used to coordinate the execution of parallel tasks, and whose accesses to shared data are synchronized via transactions. We start by illustrating, via a set of examples of concurrent computations with increasing complexity, 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 dependency graphs, there are different alternatives to define the notion of atomicity between a future and its continuation.

On the basis of these considerations, we formalize four different semantics over two dimensions: the degree of atomicity between futures and continuations, and their admitted serialization orders. The proposed semantics are defined on the basis of a graph-based characterization of the dependencies developed by transactions, both by interacting via reads or writes to shared variables and via the creation and evaluation of futures.

2. WHICH SEMANTICS FOR 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 $T = \{Th_1, \ldots, Th_n\}$ of threads which can access (i.e., read and write) a set of shared variables $V$.

In the conventional transactional model, transactions start by issuing a begin operation, can be followed by a sequence of read and write operations, and be finally completed by either a commit or abort operation. 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. In our model we do not
formulate any assumption on the domain of the values returned by transactions nor on the logic used to determine them (e.g., it may be non-deterministic).

Second, we allow transactions to issue, besides reads and writes, two additional operations: submit and evaluate. These two primitives allow, respectively, for submitting and evaluating transactional futures, i.e., transactions encapsulated in a future that can run in parallel with the thread that submitted them. We consider in our model, future submitting and evaluating can only be done in a transaction. The submit operation takes as input a transaction \( T \), activates a parallel thread in which it runs \( T \), and returns a future object \( f \in F \). The returned future object \( f \) can be passed as input parameter to the operation evaluate. This primitive blocks until the transaction associated with the future \( f \) has completed its execution, and returns the value generated by the transaction. As typical TM environments, we assume that if a transaction fails due to conflict, 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 failures 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 which do not run in the context of a future are denoted top-level transactions. It is easy to see that our 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 \( T \), within whose context \( T_F \) is submitted, and with one top-level transaction \( T_F \) 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 1 illustrates a simple example usage of transactional futures, which allows us to set the ground in our search for the expectable semantics when integrating futures and transactions. 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 \( T \) also 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, i.e., the read and write operations by \( T_F \) and \( C(T_F) \) commute. Clearly this may not the case in general, e.g., if \( T_F \) had to write \( X = 10 \cdot X \) instead of \( X + 1 \), different serializations would provide different outcomes. In cases where the serialization order of \( T_F \) and \( C(T_F) \) is relevant, it is desirable to allow programmers to specify additional restrictions on the serialization order of transactional futures and of their continuations. In this sense, in order to ensure equivalence with a sequential version of the program (not using futures), a simple solution is to impose the serialization of the operations issued by \( T_F \) before the ones by \( C(T_F) \). However, in some cases other serializations may also be meaningful.
These considerations lead us to consider two different semantics regarding the perceived ordering of operations within transactional futures:

- **Weakly Ordered Transactional Futures**: 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**: A future and its continuation should appear as executed atomically with the future serialized before its continuation.

Finally, in the example of Figure 1, it is natural to expect that the presence of future $T_F$ is hidden to other top-level transactions, which should globally appear as mutually atomic.

### 2.2 Beyond parallel nesting

The simple example considered above could also have been implemented using parallel nesting [Agrawal et al. 2008]. However, as already mentioned, futures support a broader class of concurrent computations than parallel nesting. Futures, in fact, do not force blocking the “main” thread while the nested transactions execute in parallel, but rather allow for the arbitrary interleaving of submissions and evaluations of different futures. In the following we provide a set of examples that illustrate this increased flexibility of the future abstraction, and analyze the issue of defining adequate atomicity semantics when coping with some of the complex concurrency patterns supported by futures.

**Submitting futures from a continuation.** Figure 2 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 $w(x,x_0) \rightarrow w(y,y_0) \rightarrow r(x) \rightarrow r(y)$, or $r(x) \rightarrow r(y) \rightarrow w(x,x_0) \rightarrow w(y,y_0)$. Equivalently, $T_{F2}$ should not be serialized between the write to $Y$ and the one to $Z$ by $T_0$, i.e., either $w(y,y_0) \rightarrow w(z,z_0) \rightarrow r(y) \rightarrow r(z)$, or $r(y) \rightarrow r(z) \rightarrow w(y,y_0) \rightarrow w(z,z_0)$. 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.

**Escaping transactional futures.** Interestingly, futures may escape the transactional context in which they were activated. We show a first example of escaping futures in Figure 3, in which the transactional future $T_{F2}$ is activated by the transactional future $T_{F1}$. The latter issues a write on $X$, 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. For this example, we argue that 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. it spans two transactional futures (associated with the same top-level transaction). Hence, the reads issued by $T_{F2}$ should observe either both writes on $X$ and $Y$ (by, resp., $T_{F1}$ and $T_0$) or none of them.

Figure 4 depicts another interesting programming pattern in which escaping transactional futures are 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 transactional future $T_F$. In $T_F$’s continuation, $T_1$ writes a reference of the future returned by submit($T_F$) to variable $X$, writes to $Y$ and commits. If a weak ordering semantics is specified for $T_F$, it is possible to serialize $T_F$ after these writes by $T_1$. This allows $T_1$ to commit without having to block waiting for $T_F$ to commit first, making the reference to $T_F$ available to other top-level transactions possibly executing on different threads, e.g., $T_2$ in Figure 4.

In fact, the example of Figure 4 spotlights another interesting scenario for which it is even less obvious to determine which operations should be considered as part of the continuation of a transactional future.

On the one hand, following the same rationale used when analyzing the example of Figure 3, one may argue that by communicating the reference to $T_F$ via $X$, a logical causality has been established between the write operations to $X$ and $Y$ by $T_1$ and the operations 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 continuation of $T_F$, then they would have to appear as an atomic block to $T_F$. We refer to this atomicity model, which we will define more rigorously in the next section, as **strongly atomic continuation**.

On the other hand, 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 sequential transactions propagating the reference to $T_F$ to each other. One may hence argue that requiring futures to observe the execution of arbitrary large sets of top-level transactions as atomic may be as an overly

---

1This would be necessary is a strong ordering semantic were specified for $T_F$, since, in this case, it could not be otherwise guaranteed that $T_1$ observes all the writes possibly issued by $T_F$ unless $T_F$ completes its execution.
restrictive, and arguably impractical, constraint. This leads us to consider more relaxed atomicity semantics, which limit the boundaries of a continuation to its encompassing top-level transaction. We term these semantics weakly atomic continuation.

3. FORMALIZING THE SEMANTICS OF TRANSACTIONAL FUTURES

The first step to formalize the concept of atomicity between transactional futures is to introduce a framework to specify rigorously what is a continuation, and to establish the set of feasible serialization orders among transactional futures and continuations. To this end we introduce a graph-based characterization, called Future Serialization Graph (FSG). Let $H(T, S)$ be a history defined over: i) a set of transactions $T = T_{top} \cup T_{fut}$, where $T_{top}$ denotes the set of top-level transactions and $T_{fut}$ denotes the set of transactional futures; ii) a partial order $S$ over the operations of the transactions in $T$. $FSG(H)$ is defined as a directed graph having the following set of vertexes:

1. a vertex $V^B_{T}$ for each transaction $T \in T$, which is associated with all the operations executed by $T$ since its begin until the first occurrence of the first of the following operations $\{\text{submit, evaluate, abort, commit}\}$

2. for each transactional future $T \in T_{fut}$, two additional vertexes are defined:
   a. $V^F_{T}$, which is associated with the sequence of read/write operations executed by the thread that submitted the future $T$, after $T$'s submission and until the first occurrence of $\{\text{submit, evaluate, abort, commit}\}$;
   b. $V^E_{T}$, which is associated with the sequence of operations issued by some thread that starts with the evaluation of Future $T$ and ends with the first occurrence of $\{\text{submit, evaluate, abort, commit}\}$;

and the following edges:

1. An edge $V^1 \rightarrow V^2$ for each pair $V^1, V^2$ of vertexes in FSG s.t. $V^1$ and $V^2$ are executed by the same thread $t$ and $t$ executes $V^1$ before $V^2$.

2. An edge $V^F_{T} \rightarrow V^B_{T}$ for each transactional future $T \in T_{fut}$, where we have denoted with $V^S_{T}$ the vertex associated with the operation $\text{submit}(T)$, i.e., transactional futures have to be serialized after their submission. Note that, if we denote with $T_{sub}$ the transaction that submitted the transactional future $T$, then $V^S_{T}$ coincides either with i) $V^B_{T_{sub}}$ if $T$ is the first transactional future submitted by $T_{sub}$, or ii) with some vertex $V^S_{T'}$ associated with the continuation of some transactional future $T'$ that $T_{sub}$ submitted before $T$.

3. An edge $V^F_{T} \rightarrow V^E_{T}$ for each transactional future $T \in T_{fut}$, i.e., transactional futures have to be serialized before their evaluation.

4. An edge $T:V^F_{T} \rightarrow V^C_{T}$ for each strongly ordered transactional future $T \in T_{fut}$, i.e., strongly ordered transactional futures must serialize before their continuations.

For the sake of clarity, Figure 5 shows the FSG of the example reported in Figure 1.

Inclusion of operations in transactions. Based on the information encoded in the FSG we can start defining the notion of inclusion of operations into a transaction. The key subtlety here is that, since our model supports escaping transactional futures, we need to treat their operations differently from the ones of non-escaping transactional futures. We say that 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 $T$ that passes via the vertex associated with $op$. In other words, we include in a transaction $T$ all and only the operations issued by $T$ and by any non-escaping future submitted by $T$ and, recursively, by $T$'s futures.

Strongly and Weakly Atomic Continuations. Let us now identify, given a transactional future $T$, which operations should be associated with its continuation $C(T)$. Recall that, intuitively, we aim at defining the continuation of a transactional future $T$, submitted by a thread $T_h$, as the sequence of causally related operations that lead to the evaluation of $T$, starting from the first operation issued by $T_h$ after having submitted the future.

The edges so far included in the FSG allows to capture the partial ordering relations established among transactions due to the use of $\text{submit}$ and $\text{evaluate}$ primitives on transactional futures. As already discussed, however, in the transactional future model, transactions can communicate references to futures by writing to shared memory. To tackle this issue, and capture also this form of causal dependencies, it is necessary to include in the FSG also the set of read-after-write data dependencies developed by transactions. We call this extension of the FSG, $FSG^+$.

We can now define the notion of strongly and weakly atomic continuations, which we had informally introduced in the previous section:

- **Strongly Atomic Continuation.** The strongly atomic continuation of a transactional future $T$, $C^{Strong}(T)$, is the set of vertexes contained by the paths of the FSG+ that connect $V^C_{T}$ to $V^F_{T}$.

- **Weakly Atomic Continuation.** The weakly atomic continuation of a transactional future $T$, $C^{Weak}(T)$, is the set of vertexes...
of the FSG+ that are: i) associated with the same top-level transaction of \( T \), and ii) contained in the paths that connect \( V^C_i \) to \( V^E_i \).

We illustrate the concept of strongly and weakly atomic continuations with the help of Figure 6. When using the strongly atomic continuation semantics, the operations associated with the vertexes \( V^C_i \) and \( V^B_i \) are mandated to appear as a single atomic block to the transactional future \( T_F \) represented by vertex \( V^B_i \). With the weakly atomic continuation model, \( T_F \) is only required to execute atomically with respect to \( V^C_i \), but \( T_F \) may serialize after \( V^C_i \) and before \( V^B_i \).

To conclude the specification of the proposed execution semantics for transactional futures, we describe how the information in the FSG, together with conventional transactional data dependency graph, can be exploited by a graph-based concurrency control algorithm to enforce the propose semantics. We assume that the concurrency control algorithm works as an on-line scheduler that receives the next operation \( op \) by a transaction \( T \), associated with a top-level transaction \( T_{top} \) and determines the result of the operation according to the following rules:

1. if \( op \) develops a data dependency to/from an operation \( op' \) included in a different top-level transaction \( T'_{top} \), a data dependency edge is added to the FSG from all the vertexes associated with \( T_{top} \), resp. \( T'_{top} \), to all the vertexes of the FSG associated with \( T_{top} \), resp. \( T_{top} \). (Atomicity between different top level transactions.)

2. if \( op \) develops a data dependency to/from an operation \( op' \) included with the continuation of \( T \), \( C(T) \), a data dependency edge is added to the FSG from all the vertexes associated with \( T \), resp. \( C(T) \), to all the vertexes of the FSG associated with \( C(T) \), resp. \( T \). (Atomicity between futures and continuations.)

3. if a cycle is detected in the resulting graph, break the cycle by aborting \( T \) or some other transaction involved in the cycle.

4. CONCLUSIONS AND FUTURE WORK

This work was aimed at shedding 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 may not be obvious how to define atomicity guarantees between futures and their corresponding continuations. We analyzed of a set of concurrent programming patterns, which can enabled via the future’s abstraction, and used these examples to identify and motivate a set of possible semantics for transactional futures. We formalized the proposed semantics by relying on a graph-based characterization that aims to identify the feasible serialization orders among transactional futures and their continuations.

We hope that this work will allow us to gather feedback on the envisioned semantic models for transactional futures, and pave the way for further research addressing the problem of integrating transactions and futures. As part of our work, we plan to develop an implementation of the Transactional Future abstraction in software, and to quantify the performance impacts due to the choice of different semantics models.

REFERENCES


