Transactional Tasks: Parallelism in Software Transactions

Janwillem Swalens[1], Joeri De Koster[2], and Wolfgang De Meuter[3]

1 **Software Languages Lab, Vrije Universiteit Brussel**
jswalens@vub.ac.be
2 **Software Languages Lab, Vrije Universiteit Brussel**
jdekoste@vub.ac.be
3 **Software Languages Lab, Vrije Universiteit Brussel**
wdmeuter@vub.ac.be

Abstract

Many programming languages, such as Clojure, Scala, and Haskell, support different concurrency models. In practice these models are often combined, however the semantics of the combinations are not always well-defined. In this paper, we study the combination of futures and Software Transactional Memory. Currently, futures created within a transaction cannot access the transactional state safely, violating the serializability of the transactions and leading to undesired behavior.

We define *transactional tasks*: a construct that allows futures to be created in transactions. Transactional tasks allow the parallelism in a transaction to be exploited, while providing safe access to the state of their encapsulating transaction. We show that transactional tasks have several useful properties: they are coordinated, they maintain serializability, and they do not introduce non-determinism. As such, transactional tasks combine futures and Software Transactional Memory, allowing the potential parallelism of a program to be fully exploited, while preserving the properties of the separate models where possible.

1998 ACM Subject Classification D.1.3 [Concurrent Programming] Parallel Programming; D.3.2 [Language Classifications] Concurrent, distributed, and parallel languages

Keywords and phrases Concurrency, Parallelism, Futures, Threads, Fork/Join, Software Transactional Memory

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2016.23

Supplementary Material ECOOP Artifact Evaluation approved artifact available at http://dx.doi.org/10.4230/DARTS.2.1.13

1 Introduction

Concurrent programming has become essential to exploit modern hardware, especially since multicore processors have become ubiquitous. At the same time, programming with concurrency is notoriously tricky. Over the years, a plethora of concurrent programming models have been designed with the aim of combining performance with safety guarantees. In this paper, we study futures and Software Transactional Memory.

Futures are used to introduce parallelism in a program: the construct `fork` e starts the concurrent execution of the expression e in a *parallel task*, and returns a future [3, 16]. A *future* is a placeholder that is replaced with the value of e once its task has finished. Futures make it easy to add parallelism to a program [16] and can be implemented efficiently [8, 7, 19].

While futures enable parallelism, Software Transactional Memory (STM) allows access to shared memory between parallel tasks [26]. It introduces *transactions*: blocks in which shared memory locations can be read and modified safely. STM simplifies concurrent access to shared memory, as transactions are executed atomically and are guaranteed to be serializable.

It is often desirable to combine concurrency models and this is frequently done so in practice [27]. Languages such as Clojure, Scala, and Haskell provide support both for parallel tasks and transactions. However, the combination of these two models is not always well defined or supported. Using transactions to share memory between parallel tasks has a clearly defined semantics. Conversely, parallelizing a single transaction by creating new tasks is either not allowed (e.g. Haskell) or leads to unexpected behavior (e.g. Clojure and Scala). For example, in Section 3 a path-finding algorithm is shown in which several search operations are executed in parallel on a grid. In this application, each task starts a transaction that searches for a single path. We would like to further exploit the available parallelism in each individual search operation by starting new subtasks from within each path-finding transaction. This is one application where we show the need for the combination of both *transaction-in-future* and *future-in-transaction* parallelism.

In this paper, we present *transactional tasks*, a novel construct to introduce parallelism within a transaction (Section 4). A transaction can create several transactional tasks, which run concurrently and can safely access and modify their encapsulating transaction's state. We demonstrate several desirable properties of transactional tasks (Section 5). Firstly, transactional tasks are coordinated: the tasks in a transaction either all succeed or all fail. Secondly, the serializability of the transactions in the program is maintained. Thirdly, transactional tasks do not introduce non-determinism in a transaction.

Furthermore, we have implemented transactional tasks on top of Clojure (Section 6), and applied them to several applications from the widely used STAMP benchmark suite [21]. This confirms that they can produce a speed-up with limited developer effort (Section 7). As a result, we believe that transactional tasks successfully combine futures and Software Transactional Memory, allowing the potential parallelism of a program to be fully exploited, while preserving the properties of the separate models where possible.

2 Background: Parallel Tasks, Futures, and Transactions

We start by defining parallel tasks, futures, and transactions separately. We also provide a formalization for each model, list their properties and describe their use cases.[1]

The semantics described here are deliberately very similar to those offered both by Clojure and Haskell. We aim to approximate them closely, to demonstrate how the described problems also apply to these programming languages. A list of differences between our formal model and the implementations of Clojure and Haskell is given in Appendix A.

[1] We use the following notation for sets and sequences. $\uplus$ denotes disjoint union, i.e. if $A = B \uplus C$ then $B = A \backslash C$. We use a short-hand for unions on singletons: $A \cup a$ signifies $A \cup \{a\}$. The notation $S = a \cdot S'$ deconstructs a sequence S into its first element a and the rest S'. The empty sequence is written $[]$. We write $\overline{a}$ for a (possibly empty) sequence of a.

2.1 Parallel Tasks and Futures

2.1.1 Description and Properties

A **parallel task** or thread is a fragment of the program that can be executed in parallel with the rest of the program. A runtime system schedules these tasks over the available processing units. In a typical implementation, a distinction is made between the lightweight tasks created in the program, and more heavyweight OS threads over which they are scheduled. Managing and scheduling the threads is a responsibility of the runtime, and transparent to the program.

A parallel task can be created using `fork` e.[2] This begins the evaluation of the expression e in a new task, and returns a future. A **future** is a placeholder variable that represents the result of a concurrent computation [3, 16]. Initially, the future is *unresolved*. Once the evaluation of e yields a value v, the future is *resolved to* v. The resolution of future f can be synchronized using `join` f: if the future is still unresolved, this call will block until it is resolved and then return its value.

In a functional language, `fork` and `join` are **semantically transparent** [13]: a program in which `fork` e is replaced by e and `join` f is replaced by f is equivalent to the original. As a result, futures can be added to a program to exploit the available parallelism without changing its semantics. Programmers can wrap an expression in `fork` whenever they believe the parallel execution of the expression outweighs the cost of the creation of a task.

A common use case of futures is the parallelization of homogenous operations, such as searching and sorting [16]. For this purpose, Clojure provides a parallel map operation: `(pmap f xs)` will apply `f` to each element of `xs` in parallel. Futures are also used to increase the responsiveness of an application by executing long-running operations concurrently, e.g. in graphical applications expensive computations or HTTP requests often return a future so as not to block the user interface.

2.1.2 Formalization

Figure 1 defines a language with support for parallel tasks and futures. The syntax consists of a standard calculus, supporting conditionals (`if`), local variables (`let`), and blocks (`do`). To support futures, the syntax is augmented with references to futures (f), and the expressions `fork` e and `join` e.

The program state p consists of a set of tasks. Each task contains the expression it is currently evaluating, and the future to which its result will be written. The future f associated with each task is unique, and hence can be considered an identifier for the task. A program e starts with initial state $\{\langle f, e\rangle\}$, i.e. it contains one "root" task that executes e. We use evaluation contexts to define the evaluation order within expressions. The program evaluation context $\mathcal{P}$ can choose an arbitrary task, and use the term evaluation context $\mathcal{E}$ to find the active site in the term. $\mathcal{E}$ is an expression with a "hole $\square$". We note $\mathcal{E}[e]$ for the expression obtained by replacing the hole $\square$ with e in $\mathcal{E}$.

We define the operational semantics using transitions $p \rightarrow_{\text{f}} p'$; the subscript f denotes all reduction rules that apply to futures. The operational semantics is based on [13] and [32]. Rule congruence$|_{\text{f}}$ defines that the base language can be used in each task. Transitions in the base language are written $e \rightarrow_{\text{b}} e'$ and define a standard λ calculus, but are not detailed here. The expression `fork` e creates a task in which e will be evaluated, and reduces to the

[2] In Clojure this is `(future e)`, in Scala `Future { e }`, in Haskell `forkIO e`.

Syntax

$c \in$ Constant	$::= \mathtt{true} \mid \mathtt{false} \mid 0 \mid 1 \mid \ldots$
$x \in$ Variable	
$f \in$ Future	
$v \in$ Value	$::= c \mid x \mid \lambda x.e \mid f$
$e \in$ Expression	$::= v \mid e\ e \mid \mathtt{if}\ e\ e\ e \mid \mathtt{let}\ x = e\ \mathtt{in}\ e$
	$\mid \mathtt{do}\ \overline{e;}\ e \mid \mathtt{fork}\ e \mid \mathtt{join}\ e$

State

Program state	p	$::= \mathrm{T}$
Task	$t \in \mathrm{T}$	$::= \langle f, e \rangle$

Evaluation Contexts

$$\mathcal{P} ::= \mathrm{T} \uplus \langle f, \mathcal{E} \rangle$$

$$\mathcal{E} ::= \Box \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid \mathtt{if}\ \mathcal{E}\ e\ e \mid \mathtt{join}\ \mathcal{E}$$
$$\mid \mathtt{let}\ x = \mathcal{E}\ \mathtt{in}\ e \mid \mathtt{do}\ \overline{v;}\ \mathcal{E}\ \overline{;e}$$

Reduction Rules

$$\text{congruence}|_{\mathrm{f}} \quad \frac{e \rightarrow_{\mathrm{b}} e'}{\mathrm{T} \uplus \langle f, \mathcal{E}[e] \rangle \rightarrow_{\mathrm{f}} \mathrm{T} \cup \langle f, \mathcal{E}[e'] \rangle}$$

$$\text{fork}|_{\mathrm{f}} \quad \frac{f' \text{ fresh}}{\mathrm{T} \uplus \langle f, \mathcal{E}[\mathtt{fork}\ e] \rangle \rightarrow_{\mathrm{f}} \mathrm{T} \cup \langle f, \mathcal{E}[f'] \rangle \cup \langle f', e \rangle}$$

$$\text{join}|_{\mathrm{f}} \quad \mathrm{T} \uplus \left\langle f, \mathcal{E}[\mathtt{join}\ f'] \right\rangle \uplus \left\langle f', v \right\rangle \rightarrow_{\mathrm{f}} \mathrm{T} \cup \langle f, \mathcal{E}[v] \rangle \cup \left\langle f', v \right\rangle$$

Figure 1 Operational semantics of a language with futures.

future f'. After the expression e has been reduced to a value v, `join` f' will also reduce to v. It is possible to join a task multiple times, each join reduces to the same value v. A join can only be reduced by rule join$|_{\mathrm{f}}$ if the corresponding future is resolved to a value, this detail encodes the blocking nature of our futures.

The semantic transparency property of futures is maintained by our semantics: when a task is created, its expression is evaluated and a placeholder f' is returned. When the task is joined, the placeholder is used to look up the value of the expression. This is equivalent to evaluating the future's expression in place: as our base language is purely functional, an expression always evaluates to the same value, regardless of the task it is executed in.

2.2 Software Transactional Memory

2.2.1 Description and Properties

Software Transactional Memory (STM) is a concurrency model that allows multiple threads to access shared variables, grouping the accesses into *transactions* [26]. In our case, these *transactional variables* are memory locations that contain a value, created using `ref` v. In Haskell these are called TVars, in Clojure they are refs. A transaction `atomic` e encapsulates an expression that can contain a number of *primitive operations* to the shared objects, such as reads (`deref` r) and writes (`ref-set` r v).

Transactional systems guarantee **serializability**: transactions appear to execute serially, i.e. the steps of one transaction never appear to be interleaved with the steps of another [18]. The result of a transactional program, which may execute transactions concurrently, must always be equal to the result of *a* serial execution of the program, i.e. one in which no transactions execute concurrently.

Transactions are used to allow safe access to shared memory in programs with multiple parallel tasks. These applications typically contain complex data structures of which pieces are encapsulated in transactional variables. For example, a web browser could encapsulate its Document Object Model in transactional variables, as its plug-ins are interested only in a subtree of the Document Object Model, but expect a consistent view of this part. Other examples include rich text documents, HTML pages, and CAD models [14]; networks, mazes,

Syntax

$r \in \text{Reference}$

$v \in \text{Value} \quad ::= \cdots \mid r$

$e \in \text{Expression} \quad ::= \cdots \mid \texttt{ref}\ e \mid \texttt{deref}\ e \mid \texttt{ref-set}\ e\ e \mid \texttt{atomic}\ e$

State

Program state	p	$::=$	$\langle \mathrm{T}, \theta \rangle$
Task	$t \in \mathrm{T}$	$::=$	$\langle f, e \rangle$
Heap	$\theta \in \text{Reference} \rightharpoonup \text{Value}$		
Local store	$\tau \in \text{Reference} \rightharpoonup \text{Value}$		

Evaluation Contexts

$\mathcal{P} ::= \langle \mathrm{T} \uplus \langle f, \mathcal{E} \rangle, \theta \rangle$

$\mathcal{E} ::= \cdots \mid \texttt{ref}\ \mathcal{E} \mid \texttt{deref}\ \mathcal{E} \mid \texttt{ref-set}\ \mathcal{E}\ e \mid \texttt{ref-set}\ r\ \mathcal{E}$

Reduction Rules

$$\text{congruence}|_{\mathrm{t}} \quad \frac{\mathrm{T} \rightarrow_{\mathrm{f}} \mathrm{T}'}{\langle \mathrm{T}, \theta \rangle \rightarrow_{\mathrm{t}} \langle \mathrm{T}', \theta \rangle}$$

$$\text{atomic}|_{\mathrm{t}} \quad \frac{\langle \theta, \{\}, e \rangle \Rightarrow_{\mathrm{t}}^{*} \langle \theta, \tau', v \rangle}{\langle \mathrm{T} \uplus \langle f, \mathcal{E}[\texttt{atomic}\ e] \rangle, \theta \rangle \rightarrow_{\mathrm{t}} \langle \mathrm{T} \cup \langle f, \mathcal{E}[v] \rangle, \theta :: \tau' \rangle}$$

Transactional Transitions

$$\text{ref}||_{\mathrm{t}} \quad \frac{r \text{ fresh}}{\langle \theta, \tau, \mathcal{E}[\texttt{ref}\ v] \rangle \Rightarrow_{\mathrm{t}} \langle \theta, \tau[r \mapsto v], \mathcal{E}[r] \rangle}$$

$$\text{ref-set}||_{\mathrm{t}} \quad \langle \theta, \tau, \mathcal{E}[\texttt{ref-set}\ r\ v] \rangle \Rightarrow_{\mathrm{t}} \langle \theta, \tau[r \mapsto v], \mathcal{E}[v] \rangle$$

$$\text{deref}||_{\mathrm{t}} \quad \langle \theta, \tau, \mathcal{E}[\texttt{deref}\ r] \rangle \Rightarrow_{\mathrm{t}} \langle \theta, \tau, \mathcal{E}[(\theta :: \tau)(r)] \rangle$$

$$\text{atomic}||_{\mathrm{t}} \quad \langle \theta, \tau, \mathcal{E}[\texttt{atomic}\ e] \rangle \Rightarrow_{\mathrm{t}} \langle \theta, \tau, \mathcal{E}[e] \rangle$$

$$\text{congruence}||_{\mathrm{t}} \quad \frac{e \rightarrow_{\mathrm{b}} e'}{\langle \theta, \tau, \mathcal{E}[e] \rangle \Rightarrow_{\mathrm{t}} \langle \theta, \tau, \mathcal{E}[e'] \rangle}$$

Figure 2 Operational semantics for language with transactions.

graphs, and lists [21]. STM is especially suited if it is not yet known at the start of the atomic block which objects will be accessed: due to optimistic synchronization these are only locked for the duration of the commit.

2.2.2 Formalization

In Figure 2 we extend our language with support for transactions. We extend the syntax of Figure 1 with references to transactional variables (r), and the transactional operations. The program state is extended with a transactional heap θ that contains the values of the transactional variables.

Similar to the semantics for STM in Haskell [17], we make a distinction between two types of transitions. Regular program state transitions are written with a single arrow $p \rightarrow_{\mathrm{t}} p'$ and marked with $|_{\mathrm{t}}$. In contrast, transitions inside a transaction are written with a double arrow $\langle \theta, \tau, e \rangle \Rightarrow_{\mathrm{t}} \langle \theta, \tau', e' \rangle$ and marked with $||_{\mathrm{t}}$. Here, θ is the transactional heap at the start of the transaction, and remains unchanged throughout the transactional transitions as it is only used to look up values. Conversely, τ is the local store that contains the updates made to transactional variables during the transaction.

Transactional operations `ref` v creates a new r and sets its value in the local store to v. `ref-set` r v updates the value of r to v. `deref` r will look for the value of r in $\theta :: \tau$, i.e. first in the local store τ and then the global heap θ.[3]

atomic$||_{\mathrm{t}}$ This rule applies to nested transactions, i.e. one `atomic` nested in another. A nested transaction is reduced in the context of its outer transaction, hence a nested `atomic` e reduces to e.

congruence$||_{\mathrm{t}}$ For now, we only allow reduction rules of the base language ($\rightarrow_{\mathrm{b}}$) to be applied in a transactional context. There is no congruence rule for $\rightarrow_{\mathrm{f}}$ in a transaction, so it cannot contain `fork` or `join` (this is introduced in the rest of this paper).

[3] $(\theta :: \tau)(r) = \begin{cases} \tau(r) & \text{if } r \in \text{dom}(\tau) \\ \theta(r) & \text{otherwise} \end{cases}$

atomic$|_{\mathrm{t}}$ The changes made to transactional variables during a transaction are gathered in τ, and applied to the heap at once, in atomic$|_{\mathrm{t}}$. This encodes exactly the atomicity of a transaction: its effects appear to take place in one indivisible step. This also encodes the serializability of transactions: semantically it is as if each transaction were executed serially.

According to the classification of Moore and Grossman [22], our transactional semantics is high-level but small-step. It is *high-level*, as it relies on non-determinism in the reduction rules (specifically in the definition of $\mathcal{P}$) to find a successful execution. In contrast to an actual implementation, in the semantics there is no notion of conflicting or aborting transactions. Thus, we do not need to write down a commit protocol, allowing us to focus on the transitions inside the transaction. This simplification provides a straightforward model for programmers, and a correctness criterion against which more complex implementations can be verified. At the same time, the semantics is *small-step*: a transaction takes several steps to complete (denoted with $\Rightarrow_{\mathrm{t}}$). This will allow us to describe parallelism in the transaction later.

In an actual implementation, multiple transactions execute in parallel. Clojure and Haskell implement STM using optimistic synchronization instead of (pessimistic) locks: a transaction is started, and if a conflict is detected while the transaction is running or during its commit, the changes are rolled back and the transaction is retried. Our implementation follows a similar approach and is described in detail in Section 6.

Transactions introduce non-determinism: a program with two tasks that each execute a transaction has two serializations, one in which the first transaction precedes the second and vice versa. However, this is a limited amount of non-determinism: the order in which the individual instructions of two transactions are interleaved does not matter, the developer only has to reason about transactions as a whole.

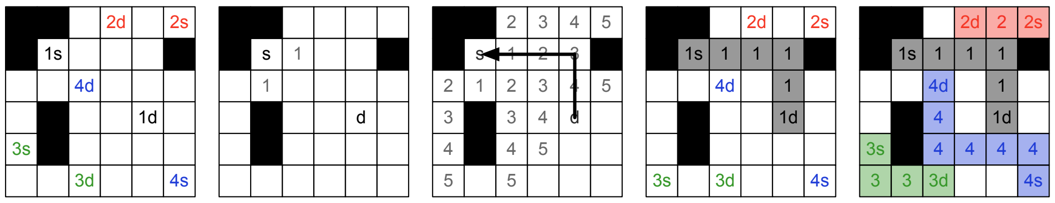
Lastly, by introducing transactions the `fork` and `join` constructs have lost their semantic transparency. For instance, a program that forks two tasks that each execute a transaction has a serialization in which the second transaction precedes the first. However, in the same program with the `fork` and `join` constructs removed the first transaction always precedes the second. This violates the semantic transparency property.

3 Motivation and Example

This section discusses the use of parallelism inside transactions, i.e. `fork` inside `atomic`. Using an example, we illustrate that it is desirable for certain applications to use parallelism in transactions. We discuss how this is realized in contemporary programming languages, and demonstrate that they do not provide satisfying semantics, as several desirable properties are not guaranteed.

As an example, we look at the Labyrinth application of the STAMP benchmark suite [21]. This application implements a transactional version of Lee's algorithm [20, 31]. Lee's algorithm, as it is used in chip design, places electric components on a grid and connects them without introducing overlapping wires.

Listing 1 shows the transactional version of Lee's algorithm implemented in Clojure, based on [31]. Its main data structure is `grid`, a two-dimensional array of transactional variables. The aim of the application is to find non-overlapping paths between given pairs of source and destination points. For example, Figure 3a depicts four source–destination pairs on an 6×6 grid, and Figure 3e shows a possible solution of connecting paths. The transactional variable `work-queue` is initialized to the input list of source–destination pairs.



(a) Global grid: input of four pairs of source and destination points **(b)** Local grid of path 1: first expansion step **(c)** Local grid of path 1: trace back from destination to source **(d)** Global grid updated with path 1 **(e)** Final global grid with four non-overlapping paths

Figure 3 Different steps of the Labyrinth algorithm, illustrated using 4 paths on a two-dimensional 6×6 grid. The black squares are impenetrable 'walls'.

Listing 1 Transactional version of Lee's algorithm in Clojure.

```
1  (def grid (initialize-grid w h))      ; w × h array of cells, each cell is a ref
2  (def work-queue (ref (parse-input))) ; list of source–destination pairs
3
4  (loop [] ; 4
5    (let [work (pop work-queue)] ; atomically take element from work queue
6      (if (nil? work)
7        true ; done
8        (do
9          (atomic
10           (let [local-grid (copy grid)
11                 [src dst]  work ; destructure source–destination pair using pattern matching
12                 reachable? (expand src dst local-grid)] ; ref-sets on local-grid
13             (if reachable?
14               (let [path (traceback local-grid dst)]
15                 (add-path grid path)))))               ; ref-sets on grid
16         (recur)))))
```

As long as the work queue is not empty, a source and destination pair in the input will be processed in a new transaction (lines 9–15). This happens in four steps. First, a local copy `local-grid` is created of the shared `grid` (line 10). Next, a breadth-first search expands from the source point (line 12), recording the distance back to the source in each visited cell of the `local-grid` using `ref-set` (Figure 3b), until the destination point is reached. Afterwards, the traceback function finds a path from the destination back to the source, minimizing the number of bends (line 14; Figure 3c). Finally, the shared `grid` is updated to indicate these cells are now occupied (line 15; Figure 3d). After the transaction has finished, this process is repeated until all work has been processed.[4]

To parallelize this algorithm, several "worker threads" execute the loop simultaneously. Each thread repeatedly takes a source–destination pair from the work queue and attempts to find a connecting path in a transaction. If two threads result in overlapping paths, a conflict occurs when updating the global `grid` (line 15), as the two threads attempt to write to the same transactional variable (that represents the cell where the paths collide). As a result, one of the two transactions is rolled back and will look for an alternative path.

[4] Clojure's loop `(loop [x 0] (recur 1))` defines and calls an anonymous function, in which `recur` executes a recursive call. It is equivalent to Scheme's named let: `(let n ([x 0]) (n 1))`.

Table 1 Characterization of the STAMP applications, abridged from [21]. These numbers were gathered on a simulated 16-core system. The transaction length and transactional execution time are color-coded high ●, medium ◐, low ○.

Application	Instructions /tx (mean)	Time in tx
labyrinth	219,571 ●	100% ●
bayes	60,584 ●	83% ●
yada	9,795 ●	100% ●
vacation-high	3,223 ◐	86% ●
genome	1,717 ◐	97% ●
intruder	330 ○	33% ◐
kmeans-high	117 ○	7% ○
ssca2	50 ○	17% ○

Minh et al. [21] measure various metrics of the applications in the STAMP benchmark, shown in Table 1. We compare the Labyrinth application with the other applications. Firstly we see that this application spends 100% of its execution time in transactions. Hence, the amount of parallelism in this program is maximally the number of transactions that are created, which is the number of input source–destination pairs, even on a machine with more cores. To allow more fine-grained parallelism to be exploited in this program, it is necessary to allow parallelism inside the transactions. Secondly, we infer that the transactions in this application take a long time to execute: an average transaction of the Labyrinth application contains several orders of magnitude more instructions than the other applications in the STAMP benchmark. This means conflicts will be costly: retrying a transaction incurs a large penalty. Parallelizing the transactions will reduce this cost.

Profiling reveals that, for typical inputs, more than 90% of the execution time of the program is spent in the expansion step, which performs a breadth-first search. Listings 2 and 4 show a simplified version of the relevant code. The `expand` function starts with a queue containing the `src` point (listing 2, line 2). In `expand-point` (listing 2, line 10), the first element of the queue is expanded, which updates the neighboring cells in `local-grid` for which a cheaper path has been found, using `ref-set` (listing 4, line 11), and returns these neighbors. These are then appended to the queue (listing 2, lines 8–10), and the loop is restarted. This continues until either the queue is empty or the destination has been reached.

In Listing 3, additional parallelism is exploited by replacing the breadth-first search algorithm by a parallel version of this algorithm. It uses *layer synchronization*, a technique in which all nodes of one layer of the breadth-first search graph are expanded—in parallel—before the next layer is started [33]. The queue now starts as a set containing only the `src` point (line 2). In each iteration of the loop, `expand-step` will expand all elements in the queue in parallel (lines 11–13), using Clojure's parallel map operation `pmap`. The union of all returned neighbors is then used as the queue for the next iteration of the loop (line 10). As before, this continues until either the queue is empty or the destination has been reached.

However, this code does not work as expected in Clojure! Each iteration of `pmap` is executed in a new thread, executing a call to `expand-point`, in which an `atomic` block appears. As transactions are thread-local in Clojure, it detects no transaction is running in the current thread, and starts a *new* transaction. When the `atomic` block ends, this inner transaction is committed. However, the surrounding transaction may still roll back, while the inner transaction cannot be rolled back anymore.

Listing 2 Expansion step: sequential, breadth-first search of local grid.

```
(defn expand [src dst local-grid]
  (loop [queue (list src)]
    (if (empty? queue)
      false ; no path found
      (if (= (first queue) dst)
        true ; destination reached
        (recur
          (concat
            (rest queue)
            (expand-point
              (first queue)
              local-grid)))))))
```

Listing 3 Parallel breadth-first search of local grid.

```
(defn expand [src dst local-grid]
  (loop [queue (set [src])}]
    (if (empty? queue)
      false ; no path found
      (if (contains? queue dst)
        true ; destination reached
        (recur (expand-step queue local-grid))))))

(defn expand-step [queue local-grid]
  (reduce union (set [])
    (pmap ; parallel map
      (fn [p] (expand-point p local-grid))
      queue)))
```

In general, Clojure allows threads to be created in a transaction, but they are not part of that transaction's context. When an `atomic` block appears in a new thread, a separate transaction is created with its own, possibly inconsistent, snapshot of the shared memory. This transaction will commit separately. Clojure does not consider thread creation as part of the transaction, hence it is not undone when the encapsulating transaction is rolled back. As such, *the serializability of the transactions is broken.* This is not the desired behavior of the presented example.

The same problems occur in most library-based STM implementations, including ScalaSTM, Deuce STM for Java, and GCC's support for transactional memory for C and C++.[5] Haskell, on the other hand, does not allow the code above to be written. The type system prohibits the creation of new threads in a transaction, as transactions are encapsulated in the STM monad while `forkIO` can only appear in the IO monad. As such, the serializability of transactions is guaranteed, but the parallelism is limited.

In conclusion, there are several reasons why current approaches to parallelism inside transactions are unsatisfactory [22]:

- By disallowing the creation of threads in transactions, in effect, **the parallelism of a program is limited**: every time transactions are introduced to isolate some computation from other threads, the potential performance benefits of parallelism *inside* this computa-

[5] `https://nbronson.github.io/scala-stm/`, `https://sites.google.com/site/deucestm/`, `https://gcc.gnu.org/wiki/TransactionalMemory`. GCC has support for transactional memory since version 4.7, although still labeled experimental.

Listing 4 Expand a point. (Code simplified for clarity.)

```
(defn expand-point [current local-grid]
  (atomic
    (let [cheaper-neighbors
            (filter
              (fn [neighbor]
                (< (cost neighbor current) ; cost of path to neighbor, through current
                   (deref neighbor)))       ; cost of previous path to neighbor
              (neighbors local-grid current))] ; neighbors of current
      (doseq [neighbor cheaper-neighbors]           ; for each cheaper neighbor:
        (ref-set neighbor (cost neighbor current))) ; set new cost
      cheaper-neighbors)))
```

tion are forfeited. This problem becomes apparent for programs containing long-running transactions. In the Labyrinth example, the maximal amount of parallelism is equal to the number of input source–destination pairs, even though additional parallelism could be exploited by the breadth-first search algorithm.

- In languages and libraries that do allow the creation of threads in transactions, **threads in transactions do not execute within the context of the encapsulating transaction**. This means they do not have access to the transactional state of the encapsulating transaction; instead a new transaction is started. The modifications of the new transaction are committed separately, and the serializability of transactions is no longer guaranteed.
- Inside a transaction, calling a library or other part of the program that contains `fork` is unsafe: it is either not allowed or can lead to incorrect results. This can severely **hinder re-usability** [15]. For instance, it is impossible to use a library that implements a parallel breadth-first search for the Labyrinth application.
- Last, transactions are used to ensure isolation, e.g. to prevent overlapping paths in the example, but they also form the unit of parallelism, evidenced by the fact that the maximal amount of parallelism is equal to the number of transactions. Moore and Grossman [22] and Haines et al. [15] argue that isolation and parallelism are orthogonal issues, but the **notions of isolation and parallelism are conflated**. Parallelizing the search algorithm should be orthogonal to the isolation between transactions, but it is not.

The ideal solution is one where several tasks can be created in a transaction and execute in parallel, i.e. allowing `fork` inside `atomic` (unlike Haskell's `forkIO`). Furthermore, these tasks should be able to access and modify the transactional variables, using the transactional context of the encapsulating transaction (unlike Clojure's or Scala's threads in transactions). With our approach, we want to allow coordination of different tasks by encapsulating them in a transaction: they either all succeed and all their changes are committed, or they all roll back. Finally, the serializability between all transactions in the program should be preserved.

4 Transactional Tasks

In this section, we define transactional tasks (Section 4.1). *Transactional tasks* are parallel tasks that are spawned in a transaction. A transactional task can access and modify the state of its encapsulating transaction, i.e. it operates within its *transactional context.* To allow multiple tasks to operate on the same transactional context simultaneously, each transactional task works on its own version of the data. This resembles traditional transactions, where each transaction operates on its own local store, and later commits its local updates. When a transactional task is joined by its parent, its updates are merged into its parent's context.

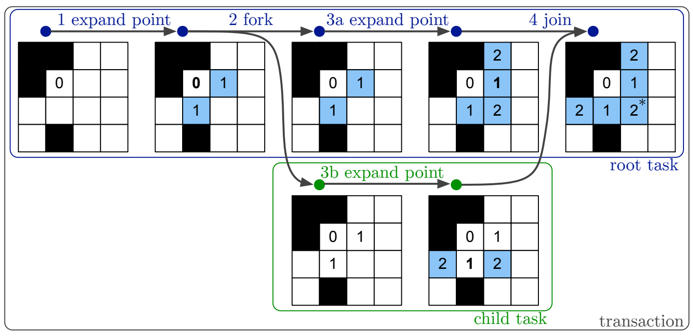


Figure 4 The timeline of the transactional tasks that are forked and joined when expanding the labyrinth grid. At each point in time, we show the grid as it exists in that task. The cells that are stored in the snapshot of the task are white, the modifications stored in the local store are blue.

We describe the joining semantics in Section 4.2, and discuss the properties of transactional tasks in Section 4.3.

4.1 A Transactional Task

In this section, we define what a transactional task is: what data it contains and what operations can be performed on it.

Each transaction starts with one *root* task that will evaluate the transaction's body. In a task, more tasks can be created using the `fork` *e* construct.

In contrast with other approaches, in our model each transactional task created by a fork also operates with respect to a transactional context. Conceptually, each transactional task operates on its own private copy of the transactional heap, and will access and modify that. To this end, a transactional task contains two parts: a *snapshot* containing the values of the transactional variables when the task was created, and a *local store* containing the values the task has written to transactional variables. In Figure 4, a timeline of the `expand` operation of the Labyrinth application is shown. At the start of the transaction, the snapshot of the root task contains the source cell at distance 0 (in white). After step 1, the local store is modified with the expanded cells at distance 1 (in blue).

When a task is created, its snapshot reflects the current state of the transactional variables. Hence, it is the snapshot of its parent task modified with the current local store of the parent. The snapshot of the root task is a copy of the transactional heap. The local store of a newly created task is empty. In Figure 4, step 2 creates a fork, the forked task's snapshot (in white) consists of its parent's snapshot combined with the parent's local store.

While a task executes, it can look up transactional values in its snapshot, and modify them by storing their updated values in the local store. When a task finishes its execution, its future is resolved to its final value. In steps 3a and 3b, the root task and its child both expand a cell and update their local stores (in blue).

When a task is joined for the first time, its local store is merged into the task performing the join, and the value of its future is returned. Step 4 copies the modified cells of the child task (blue cells) into the root task. Subsequent joins of the same task will not repeat this, as their changes are already merged; they will only return the last value of its future.

At the end of the transaction, the modifications of all transactional tasks should have been merged into the root task, and these are committed atomically. If a conflict occurs at commit time, the whole transaction is retried. If a conflict occurs in one of the tasks while the transaction is still running, *all* tasks are aborted and the whole transaction is retried. In other words, the tasks within a transaction are coordinated so that they either all succeed or all fail: they form one atomic block.

4.2 Conflicts and Conflict Resolution Functions

On a join, conflicts are possible: it may happen that the child task has changed a transactional variable that the parent also modified since the creation of the child. In that case, a write–write conflict occurs. An example of such a conflict is marked on Figure 4 with an asterisk (*). Note that in this example both values happened to be the same, but that this is not necessarily the case in general.

For these situations, we allow a *conflict resolution function* to be specified per transactional variable (similar to Concurrent Revisions [9]). To this end, the programmer provides a `resolve` function when the transactional variable is created, i.e. `(ref initial-value resolve)`. If a conflict occurs, the new value of the variable in question is the result of `resolve`$(v_{\text{original}}, v_{\text{parent}}, v_{\text{child}})$, where v_{parent} and v_{child} refer to its value in the parent and child respectively, and v_{original} refers to its value when the child was created (stored in the child's snapshot).

In the Labyrinth example, the new value of a conflicting cell should be the minimum of the joining tasks, i.e. `resolve`$(o, p, c) = \min(p, c)$, as we want to find the cheapest path. Generally, conflict resolution functions are useful when each task performs a part of a calculation. For example, when each task calculates a partial result of a sum, the resolve function is `resolve`$(o, p, c) = p + c - o$, i.e. the total is the value in the parent plus the value that was added in the child since its creation. Similarly, if several tasks generate sets they are combined using `resolve`$(o, p, c) = p \cup c$, or if several tasks generate lists of results they can be combined with `resolve`$(o, p, c) = \text{concat}(o, p - o, c - o)$. If no conflict resolution function is specified, we default to picking the value in the child over the one in the parent, i.e. `resolve`$(o, p, c) = c$. We reason that explicitly joining a task is equivalent to requesting all its changes to be merged. On the other hand, the parent may be preferred by specifying `resolve`$(o, p, c) = p$.

Read–write "conflicts" are not considered to be actual conflicts in our model. If the parent reads a transactional variable while its child wrote to it, the parent still reads the 'old' value from its snapshot. The value will only be updated after an explicit `join` of the child. This prevents non-determinism, as the moment at which changes from one task become visible in another does not depend on how tasks are scheduled.

4.3 Properties

Transactional tasks provide several useful properties. We describe them here and discuss them more formally in Section 5.2.

Serializability of transactions Transactional tasks should always be joined before the end of the transaction they are created in. While a transaction can contain many tasks, each task is fully contained in a single transaction. The changes of the tasks in a transaction have therefore all been applied to the transaction's local store before commit, and on commit they are applied to the transactional heap at once. Consequently, transactions remain atomic and serializable. The introduction of transactional tasks does not change the developer's existing mental model of transactions.

Coordination of transactional tasks The changes of all tasks created in a transaction are committed at once. This means they either all succeed, or all fail. If a conflict occurs in one task during its execution, all other tasks in the transaction are aborted and the transaction, as a whole, restarts. As such, transactions can be used as a mechanism to coordinate tasks. The developer can reason about transactions as one atomic block.

Determinism in transactions Transactional tasks do not introduce non-determinism in transactions. Firstly, it does not matter in which order the instructions of two tasks are interleaved since they both work on their own copies of the data. Secondly, the join operation is deterministic as long as the conflict resolution function is. The changes made in one task only become visible in another after an explicit and deterministic `join` statement, making the behavior in a transaction straightforward to predict. Furthermore, the developer can easily trace back the value of a variable by looking where tasks were joined.

Transactional tasks are nested transactions A transactional task makes a read-only snapshot of the transactional state upon its creation, and stores its modifications in a local store. This mirrors closely how a regular transaction creates a read-only copy of the transactional heap at its creation and stores its modifications in a local store. We could say that, while it is not syntactically shown, a transactional task starts a 'nested transaction'. This similarity should provide a familiar semantics to developers. The differences with nested transactions in the existing literature are discussed in Section 8.

No semantic transparency Transactional tasks are *not* semantically transparent: wrapping `fork` around an expression in a transaction changes the semantics of the program. As explained at the end of Section 2.2, this was already the case for non-transactional tasks that contain a transaction: they also do not necessarily evaluate to the same result every time.

Violating the semantic transparency is a necessary compromise to achieve our goal of executing tasks in parallel. If a transactional task were semantically transparent, its effects on the transactional state would need to be known at the point where it is created, before the parent task can continue. Therefore the child task and its parent would need to be executed sequentially. Instead, we opt to omit semantic transparency as a necessary compromise to accomplish the parallel execution of tasks.

Nonetheless, we argue that we maintain the "easy parallelism" of futures. Firstly, the determinism in transactions ensures that the order in which the instructions in the transactional tasks are interleaved does not affect the result. Secondly, transactional tasks provide a straightforward and consistent semantics of how the transactional effects of tasks are composed. Each task can modify the transactional state, but its effects become visible in a single step only when it is joined, and conflicts are resolved deterministically.

Non-transactional tasks maintain their semantics Tasks that are spawned outside a transaction are unchanged in our model: these *non-transactional tasks* cannot access or modify the transactional state directly. They can still create a transaction internally to modify the transactional state indirectly, as shown earlier in Section 2.

4.4 Summary

Using the concepts introduced in this section, the code in Listings 3 and 4 now behaves as expected. Each newly created task is part of the encapsulating transaction's context, and has access to its state. Transactional tasks can observe the changes that occurred before they were created, and they make their modifications in a private local store. When they are joined, their changes become visible in their parent task.

Eventually, all tasks are joined into the transaction and the transaction commits. All tasks of a transaction are coordinated, and transactions remain serializable, therefore all tasks in one transaction behave as a single atomic block. Transactional tasks provide a straightforward semantics: behavior in a transaction is deterministic, and values can be traced back by looking at the `join` statements. Transactional tasks also provide a familiar semantics: they behave as nested transactions.

5 Semantics and Properties

In this section, we describe the semantics and properties of the transactional tasks model.[6]

5.1 Semantics

Figure 5 defines how transactional tasks are modeled. The program state and non-transactional tasks are as defined in Section 2. On the other hand, a transactional task t_x contains the following components: a future f used to join the task and read its final result, a snapshot σ that contains the values of the transactional variables at its start, a local store τ that stores its changes to transactional variables, a set $\mathrm{F_s}$ of spawned child tasks, a set $\mathrm{F_j}$ of tasks joined into this task, and finally the expression e being evaluated in this task.

Outside a transaction, rules are written $\langle \mathrm{T}, \theta \rangle \rightarrow_{\mathrm{tf}} \langle \mathrm{T}', \theta' \rangle$, similar to Section 2.2. Inside a transaction, rules are written $\mathrm{T_x} \Rightarrow_{\mathrm{tf}} \mathrm{T}'_\mathrm{x}$, i.e. they work on a set of transactional tasks. A task belongs either to the set T if it is not transactional, or to one of the sets $\mathrm{T_x}$ if it is transactional (there is one such set for each transaction). Outside transactions, tasks keep their semantics as previously (rule congruence$|_{\mathrm{tf}}$, as discussed in Section 2.1).

atomic$|_{\mathrm{tf}}$ When a transaction is started, one transactional task is created: the "root" task. Its snapshot σ is the current state of transactional memory, i.e. the transactional heap θ. Its local store τ is initially empty. By applying one or more $\Rightarrow_{\mathrm{tf}}$ rules, the transaction will eventually be reduced to another set of transactional tasks, one of which is the root task. Its local store has been updated to τ', which is merged into the transactional heap on commit. We require that all tasks created in the root task have been joined ($\mathrm{F_s} \subseteq \mathrm{F_j}$). This is discussed in further detail in Section 5.2.
The essence of this rule is the same as for atomic$|_{\mathrm{t}}$ of Section 2.2: the changes τ' of the transaction are applied at once to the heap θ, which has not changed during the transaction. The major difference with rule atomic$|_{\mathrm{t}}$ is that transactions now consist of a set of tasks, instead of one expression and local store.

congruence$||_{\mathrm{tf}}$ Rule congruence$||_{\mathrm{tf}}$ specifies that the transactional transitions from Figure 2 in Section 2.2 apply in transactional tasks as well. In other words, in a transactional task creating, reading, and writing transactional variables, nesting transactions, and using the base language work just like before—although now they operate on the task's local store.

fork$||_{\mathrm{tf}}$ When a task is created, it creates a copy of the current state. Firstly, it creates a copy of the current transactional heap, i.e. its snapshot σ is set to the snapshot of its parent updated with the local modifications of its parent. In the actual implementation this is done differently for efficiency (see Section 6). Secondly, the set of joined tasks $\mathrm{F_j}$ is copied from its parent. This ensures that if any of these tasks are joined again, their transactional state is not merged again.

join$_1||_{\mathrm{tf}}$ and join$_2||_{\mathrm{tf}}$ Two rules describe the join operation of f' into f: join$_1||_{\mathrm{tf}}$ is triggered on the first join, join$_2||_{\mathrm{tf}}$ on subsequent joins.

[6] An executable implementation of the operational semantics built using PLT Redex [12] is available in the artifact, or at `https://github.com/jswalens/transactional-futures-redex`.

State

Program state	p	$::= \langle \mathrm{T}, \theta \rangle$
Non-transactional task	$t \in \mathrm{T}$	$::= \langle f, e \rangle$
Transactional task	$t_\mathrm{x} \in \mathrm{T}_\mathrm{x}$	$::= \langle f, \sigma, \tau, \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, e \rangle$
Snapshot, local store	$\sigma, \tau \in \text{Reference} \rightharpoonup \text{Value}$	
Spawned and joined tasks	$\mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j} \subseteq \text{Future}$	

Reduction Rules

$$\text{congruence}|_\mathrm{tf} \quad \frac{\mathrm{T} \rightarrow_\mathrm{f} \mathrm{T}'}{\langle \mathrm{T}, \theta \rangle \rightarrow_\mathrm{tf} \langle \mathrm{T}', \theta \rangle}$$

$$\text{atomic}|_\mathrm{tf} \quad \frac{f' \text{ fresh} \qquad \{\langle f', \theta, \{\}, \varnothing, \varnothing, e \rangle\} \Rightarrow^*_\mathrm{tf} \{\langle f', \theta, \tau', \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, v \rangle\} \cup \mathrm{T}_\mathrm{x} \qquad \mathrm{F}_\mathrm{s} \subseteq \mathrm{F}_\mathrm{j}}{\langle \mathrm{T} \uplus \langle f, \mathcal{E}[\texttt{atomic } e] \rangle, \theta \rangle \rightarrow_\mathrm{tf} \langle \mathrm{T} \cup \langle f, \mathcal{E}[v] \rangle, \theta :: \tau' \rangle}$$

Transactional Transitions

$$\text{congruence}||_\mathrm{tf} \quad \frac{\langle \sigma, \tau, e \rangle \Rightarrow_\mathrm{t} \langle \sigma, \tau', e' \rangle}{\mathrm{T}_\mathrm{x} \uplus \langle f, \sigma, \tau, \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, \mathcal{E}[e] \rangle \Rightarrow_\mathrm{tf} \mathrm{T}_\mathrm{x} \cup \langle f, \sigma, \tau', \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, \mathcal{E}[e'] \rangle}$$

$$\text{fork}||_\mathrm{tf} \quad \frac{f' \text{ fresh}}{\begin{array}{l}\mathrm{T}_\mathrm{x} \uplus \langle f, \sigma, \tau, \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, \mathcal{E}[\texttt{fork } e] \rangle \Rightarrow_\mathrm{tf} \\ \mathrm{T}_\mathrm{x} \cup \langle f, \sigma, \tau, \mathrm{F}_\mathrm{s} \cup \{f'\}, \mathrm{F}_\mathrm{j}, \mathcal{E}[f'] \rangle \cup \langle f', \sigma :: \tau, \varnothing, \varnothing, \mathrm{F}_\mathrm{j}, e \rangle\end{array}}$$

$$\text{join}_1||_\mathrm{tf} \quad \frac{f' \notin \mathrm{F}_\mathrm{j} \qquad \left\langle f', \sigma', \tau', \mathrm{F}'_\mathrm{s}, \mathrm{F}'_\mathrm{j}, v \right\rangle \in \mathrm{T}_\mathrm{x} \qquad \mathrm{F}'_\mathrm{s} \subseteq \mathrm{F}'_\mathrm{j}}{\mathrm{T}_\mathrm{x} \uplus \langle f, \sigma, \tau, \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, \mathcal{E}[\texttt{join } f'] \rangle \Rightarrow_\mathrm{tf} \mathrm{T}_\mathrm{x} \cup \left\langle f, \sigma, \tau :: \tau', \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j} \cup \mathrm{F}'_\mathrm{j} \cup \{f'\}, \mathcal{E}[v] \right\rangle}$$

$$\text{join}_2||_\mathrm{tf} \quad \frac{f' \in \mathrm{F}_\mathrm{j} \qquad \left\langle f', \sigma', \tau', \mathrm{F}'_\mathrm{s}, \mathrm{F}'_\mathrm{j}, v \right\rangle \in \mathrm{T}_\mathrm{x}}{\mathrm{T}_\mathrm{x} \uplus \langle f, \sigma, \tau, \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, \mathcal{E}[\texttt{join } f'] \rangle \Rightarrow_\mathrm{tf} \mathrm{T}_\mathrm{x} \cup \langle f, \sigma, \tau, \mathrm{F}_\mathrm{s}, \mathrm{F}_\mathrm{j}, \mathcal{E}[v] \rangle}$$

Figure 5 Operational semantics of transactional tasks.

A join can only occur once f' has been resolved, i.e. once the task has been reduced to a single value v. Moreover, we require that the joined task has itself joined all of its children ($\mathrm{F}'_\mathrm{s} \subseteq \mathrm{F}'_\mathrm{j}$). This rule applies recursively: each of the tasks in F'_s should also have joined its children. Thus, this condition ensures that all effects of the task and all its descendants have been applied to its local store τ'.

On the first join of a task, its changes are pulled into the joining task, by merging its τ' into the local τ and adding its set of joined tasks F'_j to the local F_j. On subsequent joins, by any task that has directly or indirectly joined f' before, the changes are not merged again, as these effects have already been merged. In both cases join resolves the future to its value v.

A task does not necessarily need to be joined by its parent: we do not require $f' \in \mathrm{F}_\mathrm{s}$. Any task that is able to obtain a reference to the task's future can join it, using the same rules. This maintains the flexibility of traditional futures. However, a transactional task cannot be joined outside the transaction it was created in, as it depends on the initial state of the heap at the start of the transaction. This is enforced as the joined task should be in T_x, and there is a separate such set for each transaction.

Conflict resolution function For simplicity, we did not describe the functionality provided by the custom conflict resolution function of Section 4.2 in the operational semantics. Instead, the child's value always overwrites the parent's value. To add this function, the operation $\tau :: \tau'$ in rule $\text{join}_1||_\mathrm{tf}$ should be replaced with an operation that calls the

```
(atomic
  (let [a (fork (let [d (fork …)
                      e (fork …)] …))
        b (fork (let [f (fork …)] …))
        c (fork (let [g (fork …)
                      h (fork …)] …))]
    …))
```

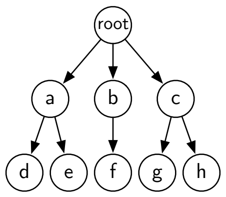


Figure 6 The tasks created in a transaction form a tree. By writing the tree in post-order notation, `d`-`e`-`a`-`f`-`b`-`g`-`h`-`c`-root, we see that task `b` can join tasks `d`, `e`, `a`, and `f`. It may be less obvious that task `f` can also join task `d`: task `a` may return `d`, and `f` can access `a`.

resolve function for conflicting writes in τ and τ'. The operations $\theta :: \tau$ in atomic$||_{\mathrm{tf}}$ and $\sigma :: \tau$ in fork$||_{\mathrm{tf}}$ do not need to be replaced: they represent non-conflicting writes.

5.2 Properties

The following properties are a result of the operational semantics:

Serializability This semantics is trivially serializable: at most one transaction is active at a time, as it is a high-level semantics, and is committed at once in atomic$|_{\mathrm{tf}}$. The semantics thus describes how a correct implementation of transactional tasks should behave for the programmer, and forms a correctness criterion for more complex implementations. An actual implementation that allows multiple transactions to execute concurrently is discussed in Section 6.

Deadlock freedom The tasks created in a transaction form a tree, as illustrated in Figure 6, with the root task at the root of the tree. Each task is identified by a future, which is a reference that can be passed around and supports one operation: `join`. Each task can obtain the future of 1) its child tasks, as it created those; 2) its descendants, if a child returns the future of a descendant (e.g. the root task can access `d` if `a` returns `d`); 3) its earlier siblings (e.g. `b` can access `a`); and 4) any descendants of its earlier siblings, if a sibling returns the future of one of its descendants (e.g. `f` can access `d` if `a` returned `d`). Tasks cannot obtain their own future. Writing the tree in post-order notation defines a strict total order on the tasks, in which each task can access the futures of the tasks that come before it. This is a consequence of the lexical scoping of the language. This means that there cannot be circular dependencies between futures, and therefore deadlocks are impossible.

Coordination: a transaction's tasks are *all* committed, atomically Rule atomic$|_{\mathrm{tf}}$ requires that all tasks created by the root task have been joined before commit ($\mathrm{F_s} \subseteq \mathrm{F_j}$). Furthermore, rule join$_1||_{\mathrm{tf}}$ specifies that each task should join its sub-tasks before it in turn can be joined ($\mathrm{F'_s} \subseteq \mathrm{F'_j}$). As a result, all tasks created in the transaction should have been joined, directly or indirectly, into the root task before it can commit. If this is not the case, the program is incorrect. Consequently, the local store τ' of the root task contains the changes to the transactional state of *all* tasks in the transaction. In rule atomic$|_{\mathrm{tf}}$, these changes are committed atomically.

In-transaction determinacy In a transaction, there is no non-determinism: given the initial state of the transactional heap θ, a transaction will always reduce to the same value v and local store τ', assuming that all conflict resolution functions are determinate. We say that a transaction is determinate. This can be proven using a technique similar to [10].

Listing 5 Overview of implementation of transactions and transactional tasks as an extension of Clojure. Abridged versions of the interfaces of the classes `Transaction` and `TxTask` are listed.

```
public class Transaction {
  // (dosync fn): runs fn in a transaction and returns the return value of fn.
  static public Object runInTx(Callable fn) {
    // If we're already in a transaction, use that one; else create one. In the new tx, fn is called, and an attempt to
    // commit is made. As long as committing fails, we rerun fn and retry.
  }
  void stop(); // Indicate that tx and its tasks should stop by setting stop flag.
}

// Implements Callable so that it can be created in a thread pool; implements Future so we can wait for its final value.
public class TxTask implements java.util.concurrent.Callable, java.util.concurrent.Future {
  static ThreadLocal<TxTask> task; // Transactional task running in current thread (can be null)
  Transaction tx;                  // Associated transaction

  Vals<Ref, Object> snapshot;   // In-tx values of refs on creation of this task (from all ancestors)
  Vals<Ref, Object> values;     // In-tx values of refs (set/commute)
  Set<Ref> sets;                // Set refs
  Map<Ref, List<CFn>> commutes; // Commuted refs
  Set<Ref> ensures;             // Ensured refs
  Set<TxTask> spawned;          // Spawned tasks
  Set<TxTask> joined;           // Directly/Indirectly joined tasks

  // (fork fn): spawns a task. Creates TxTask if a transaction is running, or a regular Future otherwise.
  static public Future spawnFuture(Callable fn) { … }
  // Create a transactional task in tx to execute fn. Snapshot is created from parent (= current thread).
  TxTask(Transaction tx, TxTask parent, Callable fn) { … }

  Object doGet(Ref r) { … }                           // (deref r)
  Object doSet(Ref r, Object v) { … }                 // (ref-set r v)
  Object doCommute(Ref r, IFn fn, ISeq args) { … }  // (commute r fn args)
  void doEnsure(Ref r) { … }                          // (ensure r)
  void join(TxTask child) { … }                       // (join child) (new)
}
```

6 Implementation

We implemented transactional tasks as an extension of Clojure, a Lisp dialect implemented on top of the Java Virtual Machine.[7] In this section, we briefly describe the core concepts of Clojure's STM and our modifications to it. Listing 5 lists the interface of the `Transaction` and `TxTask` (transactional task) classes.

Clojure represents transactional variables as `Ref`s. It uses Multi-Version Concurrency Control (MVCC) [6]: each ref contains a (limited) history of previous values and the time at which they were set. As a result, transactions can read an older version of a ref even after it has been overwritten, preventing conflicts. Only if no recent-enough value is available in the limited history, will the transaction abort and restart.

A transaction contains several data structures to store its modifications (together they correspond to the local store τ of the operational semantics): `sets`, `commutes`, and `ensured` for refs modified using `ref-set`, `commute`, and `ensure`. The latest value of each modified ref is stored in the mapping `values`. In traditional Clojure these data structures are stored in the `Transaction`. To support transactional tasks, we move them to the `TxTask` class, as each task should have its own local store of modifications. Additionally, the sets of `spawned` and `joined` tasks are stored in the current task as well, mirroring the operational semantics.

Clojure translates a transaction `(dosync e)` into `Transaction.runInTx(fn)`, where `fn` corresponds to a function executing `e`. The creation of a task using `(fork e)` is translated to `TxTask.spawnFuture(fn)` and returns a new `TxTask`, which runs in a new thread (using a thread pool). Upon its creation, the new task's `snapshot` and `values` are set to a copy

[7] It is available in the artifact, or at `https://github.com/jswalens/transactional-futures`.

(a) Code example.

```
(atomic
  1(ref-set gray A)
  2(fork 3(ref-set blue B)
         4(fork 5(ref-set green C))
         6(ref-set red D)
         …)
  7(ref-set purple E)
  …)
```

(b) Data after step 3. **(c)** Data after step 7.

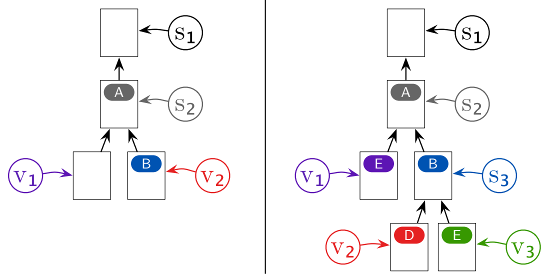


Figure 7 In the code example (a) three tasks are created. Each task contains a snapshot, which is immutable through the task's lifetime, and values, to which updated values for refs are written. (b) and (c) illustrate how the data structures are stored in memory.

of the parent's current `values`, as described in Section 6.1. The other data structures start empty. When reading a ref, `doGet(r)` searches for it first in the task's `values`, next in its `snapshot`, and finally in the ref's history. Modifying a ref calls `doSet(r, v)`, which adds the ref to `sets` and its value to `values`.

When a parent task joins a child, `(join child)` is translated to `parent.join(child)`, which takes several steps. Firstly, it waits until the child task has completed. Next, it checks whether the child task has joined all of its children. Then, the value of each ref in the `sets` of the child is copied from the child's to the parent's `values`, calling the resolution function on conflicts. The child's `sets`, `commutes`, `ensures`, and `joined` are appended to the parent's. Additionally, the child is added to its parent's `joined`. Lastly the child's final value is returned. On subsequent joins of the same child, only the final value is returned.

The transaction also contains a 'stop' flag. If one of the tasks encounters a conflict during its execution, it sets the transaction's stop flag and stops. The transactional operations (`doGet`, `doSet`, `join`...) contain a check point: they check the stop flag when they are called and abort their task if it is set. Consequently, when a task encounters a conflict and sets the stop flag, all other tasks in the same transaction will stop once they reach their next check point, and once all tasks have stopped the transaction is restarted.

When a transaction commits, at the end of `Transaction.runInTx(fn)`, it first checks whether the root task has joined all of its children: if it has we know all tasks have been joined (as explained in Section 5.2), otherwise an exception is thrown. Next, each modified ref is locked and the changes are committed. This may abort other transactions: by setting the stop flag of the other transaction, eventually all its tasks will stop and the conflicting transaction restarts.

6.1 Implementation of Snapshot and Local Store

Even though our implementation is a prototype, we briefly describe how the `snapshot` and `values` are stored, the most frequently used data structures in a transactional task. Even though the snapshot and values of a task are copied from its parent when it is created, we do not actually store duplicates.

Figure 7a lists a program that creates three tasks that each modify some refs. In Figure 7b we illustrate how the data structures are stored in memory after the third statement. We write s_i and v_i for the snapshots and values of task i. Each data structure comprises a linked list of hash maps. We exploit the fact that snapshots are immutable to share some of these hash maps between the data structures.

In the code example, task 1 (the root task) first sets A. A second task is forked and sets B. Consequently, after this step, s_1 is empty, v_1 and s_2 both contain A, and v_2 consists of A and B. Figure 7b illustrates how this is stored in memory. The snapshots are shared between the two tasks: these are immutable structures only used for look-up. The values of both tasks, v_1 and v_2, consist of a linked list that first contains a hash map that stores their private changes and next contains the shared snapshots. When ref B is updated in the second task, it is updated in the first hash map pointed to by v_2. When a ref is read in the second task, we iterate over the linked list pointed to by v_2, up the tree, until the ref is found.

Creating a new task, as in step 4, is now a matter of modifying some pointers. When task 3 is forked by task 2, the node that represented v_2 becomes the snapshot s_3 of the new task, with two empty children to contain the new values of tasks 2 and 3. After the last step, in Figure 7c, tasks 2 and 3 have updated their values with D and E respectively. Hence, v_3 now consists of the new values of task 3, the snapshot of task 3, the snapshot of task 2, and the snapshot of task 1.

By representing the `snapshot` and `values` data structures of a transactional task as a linked list of hash maps, the memory overhead of duplicated entries is eliminated. In exchange, the look-up time slightly increases as we need to iterate over the list of maps. The time to update a value is unchanged: a write happens directly in the first hash map. Forking is a matter of creating two new maps and adjusting an existing pointer. Joining still means copying values and potentially resolving conflicts, as explained earlier.

Furthermore, we performed another optimization for a common use case. In many programs it is common to create several tasks immediately after another, for example in a parallel map as in Section 3. This leads to a sequence of empty nodes in the tree. Instead of pointing a new child to an empty node, we directly point to the previous non-empty node. This optimization avoids the need to traverse empty nodes on look-ups. It is a safe optimization as non-leaf nodes in the tree are always snapshots and therefore immutable.

7 Case Studies and Experimental Results

We evaluate the applicability and performance benefits of transactional tasks using the STAMP benchmark suite [21]. STAMP consists of the eight applications listed in Table 1. It has been composed so as to represent a variety of use cases for transactions, from several application domains and with varying properties. We are interested in two characteristics in particular: (1) the transaction length, i.e. the average number of instructions per transaction in an execution; (2) time spent in transactions, i.e. the ratio of instructions in transactions over the total number of instructions in an execution of the program. When most of the execution time is spent in transactions, we can assume that those transactions execute performance-critical parts of the application. If these transactions are also long-running, they potentially benefit from more fine-grained parallelism.

As shown in Table 1 (page 8), five out of the eight applications in the STAMP suite spend a large proportion of time in transactions, three of which have long-running transactions: Labyrinth, Bayes, and Yada. In the rest of this section, we study Labyrinth and Bayes.

We also examined the Yada application. While its transactions have a relatively long execution time, they contain sequential dependencies that make it difficult to parallelize the different steps. Hence, we were unfortunately not able to achieve a speed-up using transactional tasks. The overhead of creating transactional tasks overshadows the benefit of performing only a small piece of the program in parallel. We therefore will not look at this application in further detail.

For these case studies, we first ported the STAMP applications from C to Clojure. Afterwards, we used transactional tasks to parallelize individual transactions where applicable.[8] In the performance results, the implementations with transactional tasks are compared with the original version in Clojure.

Experiments ran on a machine with two Intel Xeon E5520 processors, each containing four cores with a clock speed of 2.27 GHz and a last-level cache of 8 MB. HyperThreading was enabled, leading to a total of 16 logical threads. The machine has 8 GB of memory. Our transactional tasks are built as a fork of Clojure 1.6.0, running on the Java HotSpot 64-Bit Server VM (build 25.66-b17) for Java 1.8.0. Each experiment ran 30 times; the graphs report the median and the interquartile range.

7.1 Labyrinth

The Labyrinth benchmark was already introduced in Section 3. The goal of the benchmark is to connect given pairs of points in a grid using non-overlapping paths. For each pair, a breadth-first search is executed in a new transaction. In Section 4 we discussed how each iteration of the breadth-first search can process its elements in parallel using transactional tasks. Here, we optimized this solution to first distribute the elements into a configurable number of partitions, and then process these partitions in parallel. Furthermore, we ran the experiments on a three-dimensional grid of $64 \times 64 \times 3$ with 32 input pairs.

In the original version, there is one parameter t that influences the amount of parallelism: t worker threads will process input pairs in parallel. The maximal ideal speed-up in the original version is therefore t: in an ideal case where no transactions fail and the overhead is zero, we can expect a speed-up of maximally t. In the version that uses transactional tasks, another parameter p affects the parallelism: the number of partitions created on each iteration of the breadth-first search. Each of the t worker threads can create at most p partitions; therefore, the maximal number of threads and thus the maximal ideal speed-up in the version with parallel search is $t \times p$.[9]

In Figure 8 we measure the speed-up of running the program with several values of t and p. The speed-up is calculated relative to the version with sequential search and only one worker thread ($t = 1$). For the version that uses sequential search, the number of worker threads is increased (blue line). For the version with parallel search, both the number of partitions (different lines) and the number of worker threads (different points on the same line) are varied. The x axis denotes the maximal number of threads, i.e. t for the sequential search and $t \times p$ for the parallel search. In an ideal case, the measured speed-up would be equal to the maximal number of threads.

The blue line depicts the results of the original version of the Labyrinth application. Increasing the number of threads causes only a modest speed-up, because they find overlapping paths and consequently need to be rolled back and re-executed. This is shown in Figure 9, which lists the average number of attempts per transaction. If there is only one thread, each transaction executes only once, but as the number of threads increases each transaction re-executes several times on average. For 16 threads, the average transaction executes 2.10 times, i.e. it rolls back more than once. This hampers any potential speed-up.

In the version with parallel search, as the parameter p increases the speed-up improves, for small values of p. Each transaction now spawns p tasks, and consequently each transaction

[8] The code for these applications is available in the artifact, or at `https://github.com/jswalens/ecoop-2016-benchmarks`.

[9] To minimize the overhead of forking tasks, we ensure that each partition contains at least 20 elements.

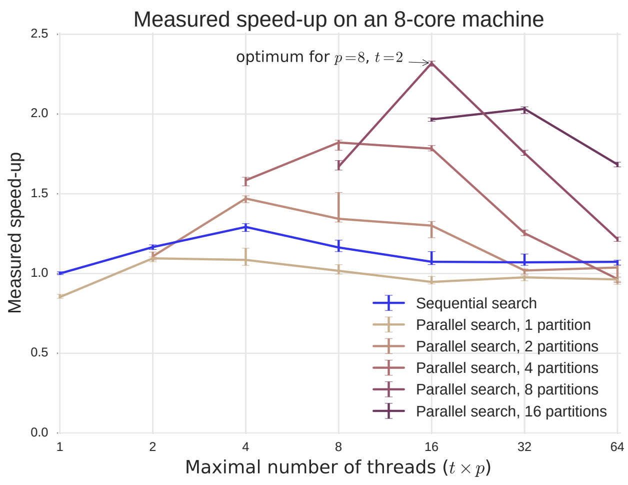


Figure 8 Measured speed-up of the Labyrinth application for the version with sequential search (blue line) and parallel search (other lines), as the total number of threads ($t \times p$) increases. Each point on the graphs is the median of 30 executions, the error bar depicts the interquartile range.

can finish its execution faster. On the tested hardware, an optimal speed-up of 2.32 is reached for $t = 2$ and $p = 8$, when two worker threads process elements and create up to eight partitions. For this case, the number of conflicts is low: each transaction executes 1.11 times on average (Figure 9).

Further increases in p lead to worse results: the additional parallelism does not offset the overhead of forking and joining tasks. Joining transactional tasks is expensive for this benchmark, as conflicts between the tasks are likely (two points expanding into a shared neighbor), and each conflict calls a conflict resolution function (the minimum, as explained in Section 4.2).

The parallel search with $p = 1$ (which does not actually search in parallel as only one partition is created) is slower than the sequential search. This is due to the different algorithms: the parallel algorithm creates several sets on each iteration to keep track of the work queue, while the sequential algorithm uses one list throughout.

These results demonstrate two benefits of transactional tasks. Firstly, the execution time of each transaction decreases by exploiting parallelism in the transaction. Secondly, the lower execution time of a transaction also means that the cost of conflicting transactions is decreased: each attempt takes less time. By varying the two parameters t and p, we can find an optimum between running several transactions simultaneously but risking conflicts (t) and speeding up the transactions internally but with more fine-grained parallelism (p).

Finally, to transform the original version with sequential search into the one with parallel search, out of 682 lines, 30 lines (4%) were removed and 78 lines (11%) were added. This corresponds to changing the sequential search algorithm into the parallel search algorithm, which is more complex.

7.2 Bayes

The Bayes application implements an algorithm that learns the structure of a Bayesian network given observed data [11, 21]. A Bayesian network consists of random variables and their conditional dependencies. Each variable in the Bayesian network is represented as a transactional variable that contains references to its parents and children. Initially, there

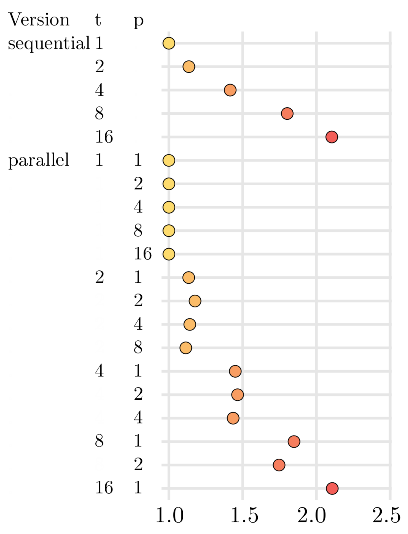


Figure 9 Average number of attempts per transaction for different values of t (transactions executing simultaneously) and p (partitions per transaction).

are no dependencies between the variables. A shared work queue contains the dependencies to insert next, and is initialized to one dependency per variable. t worker threads process the work queue in parallel: they insert the dependency into the network, and then calculate which dependencies (if any) could be inserted next, appending the best candidate to the work queue. The best candidate dependency is the one that maximizes a score function that calculates the capability of the network to estimate the input data. This is encapsulated in a transaction to prevent two dependencies from being added to the same variable simultaneously. Dependencies are inserted until the work queue is empty. As more dependencies are discovered, connected subgraphs of dependent variables form in the network.

Before the algorithm starts, the application generates the input data. Then, the t worker threads process the work queue in parallel. Figure 10 indicates that a typical execution spends 11.8% of its total time generating the input data, 88.1% learning the dependencies, and 0.1% validating the solution. We focus on the middle part only. In that part, 93.2% of the execution time is spent in the transaction that determines the best next dependency. The transaction contains a loop that calculates the score for each candidate and then selects the maximum. Each of the iterations of this loop is independent, and can therefore run in parallel using transactional tasks.

In Figure 11, we measure the speed-up of the learning phase as the number of worker threads (t) increases, for a network of 48 variables. The blue line is the original version: t threads process dependencies in parallel. The red line shows the version in which the loop is executed in parallel. Here, in each transaction, up to v transactional tasks run in parallel, where v is the number of Bayesian variables in the network (48 in our experiment). Therefore, the maximal ideal speed-up in the original version is t, while in the version with the parallel loop it is $t \times v$.

The speed-up of the original version (blue line) increases as number of threads increases, up to a speed-up of 2.75 for 16 threads. After this point, the speed-up plateaus. By examining the execution of the program, we find that even though a larger number of worker threads are created, only a limited number of them actually perform any work. The others are idle as not enough work is available after a certain point in the execution of the program.

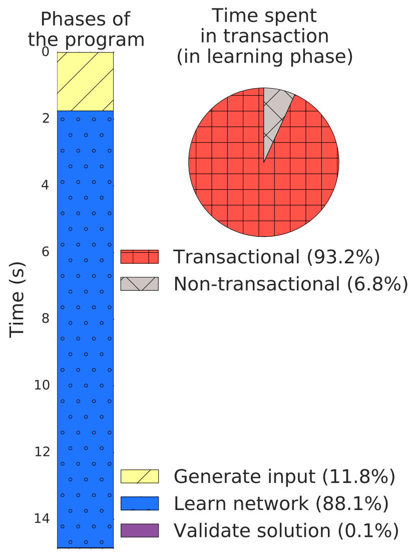


Figure 10 Proportion of time spent in different parts of the Bayes application (with $v = 48$).

In the version with parallel tasks, we see that even when there is only one worker thread processing one transaction at a time, the parallelization of its internal loop produces a speed-up of 2.88. By increasing the number of worker threads, a maximum speed-up of 3.45 can be produced for 5 worker threads. Again, the speed-up reaches a plateau as not enough work is available for all worker threads. However, the reached speed-up is higher than the original version as more fine-grained parallelism is available in each unit of work.

This result demonstrates another benefit of transactional tasks. In the original version, the amount of parallelism corresponded to the number of transactions, which is equal to number of work items. Hence, if at a certain point there are fewer work items than cores in the machine, not all potential parallelism is exploited. By introducing parallelism inside the transactions, we make better use of the available hardware. Even if there is limited work and therefore a limited number of transactions, transactional tasks allow us to make use of more fine-grained parallelism in the transactions.

Lastly, to modify the original version into the one that uses transactional tasks, only one line (out of 1248) had to be changed: the keyword `for` was replaced by `parallel-for`, a macro that uses transactional tasks internally to execute its iterations in parallel.

7.3 Conclusions

Based on these experiments, we draw the following conclusions:

- Out of the eight applications in the STAMP benchmark suite, which represents a variety of use cases for transactions, five spend a large proportion of their time in transactions and three of these have long-running transactions. We parallelized two of these three applications using more fine-grained transactional tasks.
- In the Labyrinth application, transactional tasks allow the breadth-first algorithm to be parallelized. This leads to a speed-up, by tuning the parameters for parallelism to have faster transactions and fewer conflicts.
- The Bayes application spends most of its execution time in a loop (in a transaction) that can be trivially parallelized. As there is only limited work available, at a certain point the

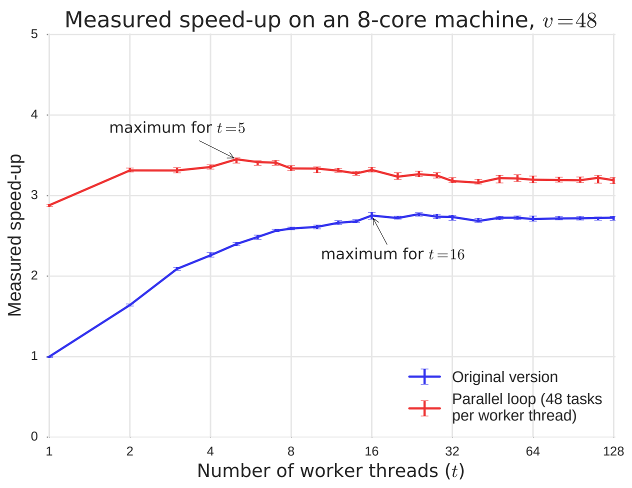


Figure 11 Measured speed-up of the learning phase for the Bayes application, as the number of threads increases. The blue line shows the original version. The red line shows the version with a parallel for loop, where each of the (at most) 48 iterations is executed in parallel.

number of transactions is lower than the number of cores in the machine. Transactional tasks allow us to introduce more fine-grained parallelism. This is a matter of changing `for` into `parallel-for`, and increases the maximal speed-up on an eight-core machine from 2.75 for the original version to 3.45 for the version with transactional tasks.

These results lead us to believe that transactional tasks allow developers to improve the performance of their transactional applications with only limited effort. Moreover, as our implementation is a relatively simple prototype in Clojure, further optimizations could decrease the overhead and improve its performance (e.g. for the Bayes application). In future work we would also like to explore the applicability of transactional tasks for other programs.

It should be possible to apply transactional tasks to other STM systems, such as Haskell or ScalaSTM. In those systems, the fact that the studied applications allow parallelism in the transaction also applies, the development effort to introduce them should be similar, but depending on the implementation the speed-up may be different.

8 Related Work

We will briefly discuss three categories of related work: nested and parallel transactions, concurrency models with deterministic access to shared memory, and work that allows parallel tasks to share memory.

Nested, multithreaded, and nested parallel transactions *Nested transactions* [23, 5, 24, 25] are subtransactions created in a transaction. Nested transactions attempt to commit separately from their parent. Hence, they can fail separately, thus requiring only a portion of the work to be rolled back. This can improve the performance of large transactions. In contrast to transactional tasks, nested transactions do not execute in parallel, as they correspond to nested `atomic` blocks and not the nesting of `fork` in `atomic`.
Haines et al. [15] and Moore and Grossman [22] allow threads to be created in a transaction, i.e. *multithreaded transactions*. However, there are no guarantees on the access to shared memory by threads within a transaction: they may read and modify shared transactional variables concurrently, thus permitting race conditions.

Transactional Featherweight Java [28] combines *nested and multithreaded transactions*: a transaction can spawn threads that contain nested transactions. When a nested transaction commits, its changes are written to its parent. Conflicts are explicitly forbidden: the value of a variable in a child must be the same as its value in the parent. *Nested parallel transactions* [1, 4, 2, 29] are similar, but resolve conflicts using the traditional serializability of transactions: the transaction that commits last wins. This is the main difference with transactional tasks, which resolve conflicts deterministically using conflict resolution functions, thus guaranteeing *in-transaction determinacy* (Sections 4.3 and 5.2). The second major difference is that nested parallel transactions roll back on conflicts between siblings, while transactional tasks do not, as they resolve the conflicts instead. In an application without such conflicts, both models operate equivalently. However, in our motivating example, the Labyrinth application, such conflicts frequently occur: the parallel expansion of the breadth-first search causes conflicts on overlapping cells. Nested parallel transactions would cause frequent rollbacks of the inner transactions, essentially sequentializing them, detrimental to performance. Transactional tasks run in parallel, but rely on the developer for an appropriate conflict resolution function. We expect both models to be suited for different applications. In applications in which non-top-level sibling transactions conflict, we expect transactional tasks to perform better.

Deterministic access to shared memory The model provided by transactional tasks *in* the transaction is similar to two existing concurrency models. Firstly, Concurrent Revisions are a model for task parallelism with shared memory [9]. Its concurrent tasks share memory using *versioned variables*. When a task is forked, a conceptual copy is made of the versioned variables; when a task is joined, the changed variables are merged into the joining task. This resembles how transactional variables behave *inside* a transaction in our model. However, between the transactions we provide serializability. As such, Concurrent Revisions provide determinacy for the complete program, while transactional tasks provide determinacy *in* the transactions, and serializability of the transactions as a whole.

A similar model is provided by Worlds [30], in which the program state is reified as a *world*. The world can be forked into a child world, a conceptual copy of all program state. The state in a child world can be updated, and eventually committed back (merged) into its parent world. As such, worlds also behave similarly to the transactional variables *in* a transaction. However, the Worlds model does not provide parallelism: a child world does not run in parallel with its parent. Instead, Worlds are used as a mechanism to 'undo' changes to the program state. As a result, when a child world is merged in its parent there will be no conflicts, as the parent has not changed in the mean time. In the case of transactional tasks and Concurrent Revisions, it is possible for the parent to have changed as well, and a form of conflict resolution between parent and child is needed.

Parallel tasks with shared memory Otello [34] allows parallel tasks to access shared memory, while still running the tasks in isolation. To this end, it introduces *assemblies*, which consist of a task and the set of shared objects it owns. When two assemblies conflict, one is re-executed after the other has finished. However, while Otello re-executes code, it does not provide transactions and as such does not guarantee serializability.

9 Conclusion

Many modern programming languages and frameworks support multiple concurrency models. However, the combination of these concurrency models is often either not supported or not well defined. In existing languages, using futures to increase the parallelism within a single transaction is either not allowed (Haskell), or leads to unexpected behavior (Clojure, Scala).

This paper introduces transactional tasks as a mechanism to enable safe parallelism within transactions. Each transactional task executes within the context of its encapsulating transaction, hence, transactions remain serializable. Furthermore, the different tasks of a single transaction are coordinated: they are all committed or retried together. Lastly, transactional tasks do not introduce non-determinism, and behave as nested transactions, thereby providing a straightforward and familiar mental model to the programmer.

This paper provides a formalization of transactional tasks, and discusses its implementation on top of Clojure. Our approach is validated through a case study of several applications from the STAMP benchmark suite. We show that our approach allows finer-grained parallelism of performance-critical, long-running transactions to be exploited, leading to a higher speed-up. As a result, we believe that transactional tasks successfully combine futures and Software Transactional Memory, allowing the parallelism of a program to be fully exploited with limited developer effort, while preserving the properties of the separate models where possible.

Acknowledgements. We would like to thank Stefan Marr for insightful discussions and comments on early drafts. We also thank the anonymous reviewers for their feedback.

References

1 K. Agrawal, J. T. Fineman, and J. Sukha. Nested Parallelism in Transactional Memory. In *PPoPP*, 2008.

2 W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating nested parallel transactions in software transactional memory. In *SPAA*, 2010.

3 H. C. Baker and C. Hewitt. The incremental garbage collection of processes. In *Symposium on Artificial Intelligence and Programming Languages*, pages 55–59, 1977.

4 J. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapałka. Leveraging parallel nesting in transactional memory. In *PPoPP*, 2010.

5 C. Beeri, P. A. Bernstein, and N. Goodman. A model for concurrency in nested transactions systems. *Journal of the ACM*, 36(2):230–269, 1989.

6 P. A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2):185–221, 1981.

7 G. E. Blelloch, P. B. Gibbons, and Y. Matias. Provably efficient scheduling for languages with fine-grained parallelism. *Journal of the ACM*, 46(2):281–321, 1999.

8 R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, 1995.

9 S. Burckhardt, A. Baldassin, and D. Leijen. Concurrent programming with revisions and isolation types. In *OOPSLA*, 2010.

10 S. Burckhardt and D. Leijen. Semantics of Concurrent Revisions. In *ESOP*, 2011.

11 D. M. Chickering, D. Heckerman, and C. Meek. A Bayesian approach to learning Bayesian networks with local structure. In *UAI*, 1997.

12 M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 2009.

13 C. Flanagan and M. Felleisen. The Semantics of Future and Its Use in Program Optimizations. In *POPL*, 1995.

14 R. Guerraoui, M. Kapałka, and J. Vitek. STMBench7: A Benchmark for Software Transactional Memory. In *European Conference on Computer Systems*, 2007.

15 N. Haines, D. Kindred, J. G. Morrisett, S. M. Nettles, and J. M. Wing. Composing first-class transactions. *ACM Transactions on Programming Languages and Systems*, 16(6):1719–1736, 1994.

16 R. H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

17 T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, 2005.

18 M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. *ACM SIGARCH Computer Architecture News*, 21:289–300, 1993.

19 S. Imam and V. Sarkar. Cooperative Scheduling of Parallel Tasks with General Synchronization Patterns. In *ECOOP*, 2014.

20 C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, 1961.

21 C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Symposium on Workload Characterization*, 2008.

22 K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, 2008.

23 J. E. B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, Massachusetts Institute of Technology, 1981.

24 J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

25 Y. Ni, V. S. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *PPoPP*, 2007.

26 N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

27 S. Tasharofi, P. Dinges, and R. E. Johnson. Why Do Scala Developers Mix the Actor Model with Other Concurrency Models? In *ECOOP*, 2013.

28 J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A Semantic Framework for Designer Transactions. In *ESOP*, 2004.

29 H. Volos, A. Welc, A. Adl-Tabatabai, T. Shpeisman, X. Tian, and R. Narayanaswamy. NePaLTM: Design and Implementation of Nested Parallelism for Transactional Memory Systems. In *ECOOP*, 2009.

30 A. Warth, Y. Ohshima, T. Kaehler, and A. Kay. Worlds: Controlling the Scope of Side Effects. In *ECOOP*, 2011.

31 I. Watson, C. Kirkham, and M. Lujan. A Study of a Transactional Parallel Routing Algorithm. In *PACT*, 2007.

32 A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. In *OOPSLA*, 2005.

33 Y. Zhang and E. A. Hansen. Parallel Breadth-First Heuristic Search on a Shared-Memory Architecture. In *Heuristic Search, Memory-Based Heuristics and Their Applications*, 2006.

34 J. Zhao, R. Lublinerman, Z. Budimlić, S. Chaudhuri, and V. Sarkar. Isolation for nested task parallelism. In *OOPSLA*, 2013.

A Syntactical and Semantical Differences with Clojure and Haskell

The syntax and operational semantics introduced in Section 2 closely matches Clojure and Haskell, with only a few insignificant differences. These are detailed in this appendix.

A.1 Clojure

Clojure 1.7.0 (released June 30, 2015) differs from the presented syntax in the following ways:

- Clojure encapsulates all forms in parentheses, as S-expressions. Furthermore, it has a slightly different syntax for `let`.
- `fork` and `join` are named `future` and `deref` respectively. That is, `deref` is overloaded for both futures and transactional variables.
- Clojure's (`dosync` $\overline{e}$) is equivalent to our `atomic` (`do` $\overline{e;}$).

The semantics differ only on these two points:

- Clojure allows `ref` and `deref` to be used outside a transaction: Clojure's (`ref` e) and (`deref` e) are equivalent to our `atomic` (`ref` e) and `atomic` (`deref` e), but with an optimized implementation.
- Clojure supports `alter`, `commute`, and `ensure`, which are essentially variations of `ref-set` with different performance characteristics.

Finally, Clojure allows futures to be created in a transaction, as described in Section 3.

A.2 Haskell

Syntactically, Haskell writes `forkIO`, `atomically`, `newTVar`, `readTVar`, and `writeTVar` for `fork`, `atomic`, `ref`, `deref`, and `ref-set`. The semantics differ in the following ways:

- Haskell's `forkIO` returns a thread identifier, and not a future. Nevertheless, our formalization models the use of transactions in tasks, in Haskell one uses `atomically` in a task created using `forkIO`.
- Moreover, Haskell does not support the `join` operation on thread identifiers. Instead, waiting for a thread and retrieving its result is usually implemented manually using an `MVar`[10]; while our language uses futures for this purpose. This aspect does not affect the problem statement of Section 3, but porting our solution to Haskell does require adding a `join` operation to Haskell.
- Transactions are encapsulated in the `STM` monad, and the main program is encapsulated in the `IO` monad. This leads to a different semantics of the `do` block, which in Haskell is syntactic sugar for monad sequencing. Haskell's do notation allows monadic binding (`<-`) and `let` binding, and may require `return`.
- Haskell does not allow multiple `atomically` blocks to be nested: the type signature of `atomically` is `STM a -> IO a`, and a result of type `IO a` cannot be used in an `STM` block.
- Haskell supports `retry` to abort a transaction, and `orElse` to compose two alternative `STM` actions.

Lastly, one could see transactional tasks as the addition of `forkSTM` to Haskell, extending its `forkIO` to usage in the STM monad.

[10] As indicated in the documentation of the Control.Concurrent package at `https://hackage.haskell.org/package/base-4.8.1.0/docs/Control-Concurrent.html#g:12`.