Context-oriented Software Transactional Memory in Common Lisp

Pascal Costanza Charlotte Herzeel Theo D'Hondt

Software Languages Lab
Vrije Universiteit Brussel
B-1050 Brussels, Belgium
pascal.costanza - charlotte.herzeel - tjdhondt @vub.ac.be

Abstract

Software transactional memory (STM) is a promising approach for coordinating concurrent threads, for which many implementation strategies are currently being researched. Although some first steps exist to ease experimenting with different strategies, this still remains a relatively complex and cumbersome task. The reason is that software transactions require STM-specific *dynamic crosscutting adaptations*, but this is not accounted for in current STM implementations. This paper presents CSTM, an STM framework based on *Context-oriented Prorgamming*, in which transactions are modelled as dynamically scoped layer activations. It enables expressing transactional variable accesses as user-defined crosscutting concerns, without requiring invasive changes in the rest of a program. This paper presents a proof-of-concept implementation based on ContextL for Common Lisp, along with example STM strategies and preliminary benchmarks, and introduces some of ContextL's unique features for context-dependent variable accesses.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Concurrent Programming; D.1.5 [*Software*]: Programming Techniques—Object-oriented Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Languages

Keywords Software transactional memory, framework design, context-oriented programming

1. Introduction

1.1 Software Transactional Memory

A common problem in multithreaded systems is that of *data races* that can appear when concurrent read and write accesses to shared data are not coordinated. Such problems are traditionally dealt with by using low-level mechanisms such as *locks* for controlling the progress of threads. Programming with locks is difficult because code using them may suffer from *deadlocks* and does not easily compose. *Software transactional memory* (STM) [20] proposes the use of a transactional model to alleviate many of these problems, by providing a well-defined protocol for automatically coordinating reads and writes to shared data.

The essential idea is that software transactions inherit the *atomicity* and *isolation* properties from database transactions. *Atomicity* requires a transactional piece of code to execute completely or, in case of failure, to roll back any side effects in order to give the illusion of not having been executed at all. *Isolation* requires the result of executing a transaction not to influence the result of other concurrently executing transactions. A correct implementation of these properties assures that transactions avoid data races.

Both library-based and language-based realizations of STM exist. STM libraries [8, 9, 16] provide APIs for opening and closing transactions, while language support for STM [6, 19] typically extends a language with a keyword `atomic` for enclosing code, and the underlying STM implementation then ensures that such atomic blocks are executed transactionally. For example, if Lisp had an `atomic` operator, a thread-safe implementation of the `insert` operation for a double-linked list could look like as follows.

```
(defun insert (node new-node)
  (atomic (set-previous new-node node)
          (set-next new-node (next node))
          (if (not (null (next node)))
            (set-previous (next node) new-node))
          (set-next node new-node)))
```

However, STM introduces a large runtime overhead due to the need to monitor read and write accesses to shared transactional memory. Numerous strategies to balance that overhead have been proposed, based on pessimistic vs. optimistic concurrency control, early vs. late conflict detection, direct vs. deferrred memory updates, and so on [16]. There is no definitive winner because each of these options perform better or worse for different applications.

Therefore, a number of *benchmark suites* have been developed for assessing different variations of STM algorithms [2, 15]. Such benchmark suites define a number of dedicated benchmarks that are developed against a generic STM interface, which in turn allows the STM algorithm to vary silently underneath. This enables comparing the runtime overheads of different algorithms under different circumstances. However, apart from providing a generic STM interface, such benchmark suites do not support the development of STM algorithms further, for example by providing reusable building blocks for implementing them. The latter is the focus of *STM frameworks* that provide common STM functionality and hooks.

1.2 STM Frameworks

Herlihy et al. previously proposed such an STM framework called *DSTM2* [8]. Their approach is implemented as a library for Java that takes advantage of Java's reflective capabilities and its class

loader architecture to create new classes at runtime for user-defined Java types. These new classes contain pairs of getter and setter methods that operate on instance variables, with additional behavior as required by the various STM algorithms. New STM strategies can be plugged in that provide templates for new such getter and setter methods.

To be able to deploy an STM strategy with DSTM2, each class, whose instance variables must be accessed transactionally, needs to be expressed as a specially annotated interface type. That interface type then needs to be explicitly passed to an STM-specific transactional factory to create another factory that must be used for creating instances of the respective interface type. This ensures that their getter and setter methods adhere to the respective STM strategy.

The changes to a program as required by DSTM2 are *invasive*: The transactional factory needs to be installed globally, all affected classes must be expressed as such annotated interface types instead, and for creating new instances, programmers must use the various generated factories throughout the program. The reason why such invasive changes are needed is that in Java, as in most object-oriented programming languages, instance variables and their getters and setters are associated with the classes to which they belong, so an STM framwork needs to find a way to insert their own STM-specific variable access semantics into those classes. However, the semantics of a particular STM strategy are conceptually independent of the involved classes, and in fact must be the same for all of them. In this sense, an STM strategy is a crosscutting concern, requiring invasive modifications of the source code when using mainstream object-oriented programming languages.

The only other STM framework we are aware of that enables expressing one's own STM strategy as a user-defined extension is our own interpreter of a subset of Scheme, which is based on modelling memory locations as explicit entities [10]. In that framework, the semantics of memory locations can be globally modified by defining reflective methods for read and write accesses to such memory locations. Since all of the data structures of our Scheme interpreter – such as plain variables, Scheme pairs, and vectors – use such memory locations internally, there is no need for any invasive changes in a Scheme program to enable the use of software transactions, except of course for inserting the necessary *atomic* blocks: As soon as the reflective methods for writing and reading memory locations are loaded into a running system, all accesses to all data structures are performed using the same transactional semantics. Except for the fact that our solution is so far only available as an interpreter, the main disadvantage of our approach remains that the modification of read and write accesses is a global change: There is no straightforward way to distinguish between different STM strategies at runtime without redefining such accesses by completely replacing them with new definitions.

1.3 Context-oriented Programming

In Context-oriented Programming (COP), programs can be partitioned into behavioral variations that can be freely composed and activated at runtime with well-defined scopes. The typical ingredients of context-oriented programming languages are [12]:

Behavioral variations describe the context-dependent behavior of a software system as increments on the underlying context-independent program definitions.

Layers group such behavioral variations as first-class entities that can be referenced in a program at runtime.

Layer activation ensures that the increments defined by a particular layer are in effect for a well-defined scope - that is, for a well-defined part of a pogram and for a well-defined duration.

ContextL is our first language extension to provide support for Context-oriented Programming. In ContextL, layers consist of partial definitions for classes and methods, and such layers can be activated and deactivated both for dynamic and global scope.

The ideas behind COP and ContextL were developed without software transactional memory as an application in mind. However, two observations led to the ideas presented in this paper: One is that the layers in COP can indeed express crosscutting concerns.[1] The other is that both layer activation in ContextL and the atomic blocks in STM are dynamically scoped constructs: Both affect the the behavior of a program for the dynamic extent of the enclosed code, and both are restricted to the current thread of execution. On closer inspection, it is also indeed the case that memory accesses in the dynamic extent of an atomic block behave differently than outside of atomic blocks (where they either throw errors, or are treated as fine-grained atomic accesses).

This strongly suggests that COP can be used to express STM, and the focus of this paper is to show to what extent COP can indeed be used to design an STM framework that does not suffer from the problems of the other STM frameworks discussed above.

1.4 Contributions

The contributions of this paper are:

- We show how to design both a user interface and an extension interface for an STM framework in a context-oriented style.
- More specifically, we discuss how user-defined modifications of the operations relevant for implementing particular STM strategies can be associated with layers, whose dynamic activations and deactivations can be aligned with atomic blocks.
- An essential part of our approach is based on context-dependent slot access, a feature of ContextL's metaobject protocol that is discussed here for the first time.
- We illustrate our approach by describing two widely known STM strategies as extensions on top of our framework, together with some first benchmarks.

2. ContextL in a Nutshell

ContextL is an extension to the Common Lisp Object System (CLOS), and thus supports object-oriented features like classes with multiple inheritance, slots (fields) and methods with multiple dispatch. Like CLOS, ContextL is not based on message sending, but on generic functions [3].[2] For each of the core defining constructs in CLOS, there is a corresponding defining construct in ContextL: For defining classes, ContextL provides `define-layered-class` as an analogue to CLOS's `defclass`. Likewise, ContextL provides `define-layered-function` and `define-layered-method` for defining layered generic functions and methods.

Layers can be introduced in ContextL with `deflayer`, an associated name, and optionally one or more *superlayers* a layer inherits from,[3] as follows.

`(deflayer` *layer-name* `[(`{*superlayer*}*`)])`

Such layers can then be further specified with partial class and method definitions, which can be explicitly associated with such layers, as follows.

[1] albeit not in the the AOP sense, by expressing pointcuts, but rather by enumerating all modifications of program entities, like in FOP [12]

[2] Note, however, that our experiments with similar extensions for other object-oriented programming languages show that COP is compatible with message sending as well.

[3] as in class inheritance

```
(define-layered-class class-name
  [:in-layer layer-name]
  ({superclass}*)
  ({slot-specification}*)
  {class-option}*)

(define-layered-method function-name
  [:in-layer layer-name]
  {method-qualifier}* parameters method-body)
```

There is always a *root* layer present that defines the context-independent behavior of a program, and class and method definitions can be associated with the *root* layer by omitting the `:in-layer` specification.

By default, only the root layer is active at runtime, which means that only definitions associated with the root layer effect the behavior of a program. Other layers can be activated at runtime by way of `with-active-layers`, as follows.

```
(with-active-layers ({layer-name}*)
  body)
```

Such a layer activation ensures that all the named layers affect the program behavior for the dynamic extent of the enclosed program code (*body*). Layer activation is dynamically scoped: Both direct and indirect invocations of generic functions (methods) and slot accesses in the control flow of the layer activation are affected. Furthermore, layer activation is restricted to the current thread of execution in multithreaded Common Lisp implementations, to avoid race conditions and interferences between different contexts.

Furthermore, layers can be deactivated at runtime by way of `with-inactive-layers`, as follows.

```
(with-inactive-layers ({layer-name}*)
  body)
```

Such a layer deactivation ensures that none of the named layers affect the program behavior for the dynamic extent of the enclosed program code anymore. As with `with-active-layers`, layer deactivation is dynamically scoped and restricted to the current thread of execution.

There are cases where it is useful that layers are activated or deactivated globally for all threads without dynamic scope, for example for interactive testing and development, or for program deployment at system startup time. The functions `ensure-active-layer` and `ensure-inactive-layer` enable such global activation and deactivation of layers.

3. CSTM: STM in ContextL

3.1 Structure of an STM implementation

An STM algorithm monitors the reads and writes of memory executed within transactions, and implements an algorithm for checking whether any of these accesses causes a data race. In case there is a data race, the STM ensures that the conflicting execution is undone by rolling back one of the transactions.

According to Larus and Rajwar, two categories of STM implementations can be distinguished that follow fundamentally different strategies: Direct-update and deferred-update STMs [16]. Direct-update systems are based on a blocking synchronization strategy. Whenever a transaction attempts to perform a write access to a shared memory location, it obtains a lock for that memory location to guarantee exclusive access, and can then instantly perform a side effect. Rollbacks are expensive in such direct-update systems, as the STM system needs to store old content of memory locations on each write access in order to be able to restore them on rollback. However, in case there are few data race conflicts, such an approach can be very efficient, due to the instant side effects.

Conversely, deferred-update systems avoid blocking: When transactions access a memory location, they acquire a copy of its content and proceed executing in terms of that copy. Only when a transaction commits, the new content is effectively written back from the copy to the original memory location. Deferred-update systems may also need to roll back transactions in case of data races, but commits and rollbacks can be implemented cheaply by using double indirections in a clever way.

3.2 The Client Interface of CSTM

Our goal in designing an STM Framework in ContextL was to keep the amount of invasive code changes as minimal as possible. Thus, CSTM provides a very straightforward extension for a CLOS programmer. Here is an example class definition that is declared to be transactional:

```
(define-transactional-class person ()
  ((name)
   (address)))
```

Such a class is introduced with a `define-transactional-class` macro, which is a very thin layer on top of layered class definitions in ContextL. The definition of the `person` class above, in fact, expands into the following equivalent ContextL code.

```
(define-layered-class person ()
  ((name :transactional t)
   (address :transacional t))
  (:metaclass transactional-class))
```

In this definition, the only differences to a 'regular' class definition in CLOS are that the slots whose accesses need to be monitored must be declared to be `transactional`, and that the class itself must be declared to be an instance of the metaclass `transactional-class`. Note especially that a class definition does not need to state which particular STM strategy it should use. Compare this to the equivalent 'vanilla' CLOS class definition below, which is almost exactly the same as the original class definition above.

```
(defclass person ()
  ((name)
   (address)))
```

Furthermore, CSTM also provides an `atomic` block construct that wraps some code to be executed in a transaction in a straightforward way: `(atomic some code)`. By default, CSTM provides a simplistic, coarse-grained locking semantics for atomic blocks, such that all atomic blocks are forced to be executed in some strict sequential order. If one wants to use an STM strategy that allows for concurrent execution of atomic blocks in different threads, it can be enabled by activating one of the STM strategy layers. CSTM provides `deferred-update-mode` and `direct-update-mode` as predefined STM strategies, but other strategies can be defined as user-provided extensions. In fact, `deferred-update-mode` and `direct-update-mode` are implemented purely in terms CSTM's extension interface (see Section 3.3).

A straightforward way to activate a particular STM strategy is to globally activate it, for example by issuing `(ensure-active-layer 'direct-update-mode)`. However, different threads can use different strategies by using the standard dynamically scoped layer activation construct, as follows.

```
(with-active-layers (direct-update-mode)
  ...)
```

Note, though, that CSTM currently does not deal with negotiating between different STM strategies in case they access the very *same* memory locations at the same time.

This is all a plain user of CSTM needs to know. Especially, there is no need to prepare classes for transactional access in any further way, and the standard means of creating instances in CLOS can be used unchanged.

3.3 The Extension Interface of CSTM

The typical steps for defining one's own STM strategy are:

- defining how transactional contexts are set up and torn down
- extending memory locations with further information (like version information, locks, etc.)
- defining both 'regular' and transactional accesses to memory locations
- defining additional actions and checks to be performed on commit and rollback

For each of these steps, CSTM provides hooks that can be extended in a context-oriented style.

Defining transactional contexts In CSTM, the `atomic` block construct is actually a thin macro on top of the layered function `call-atomic`, such that `(atomic` *`some code`*`)` expands to `(call-atomic (lambda ()` *`some code`*`))`. The default definition for `call-atomic` defines the coarse-grained locking semantics as follows.

```
(defvar *global-lock* (make-lock))

(define-layered-method call-atomic (thunk)
  (with-lock (*global-lock*) (funcall thunk)))
```

Here, a layered method definition on `call-atomic` associated with a specific STM strategy can provide different semantics for setting up and tearing down transactions.

Extending memory locations In CSTM, each slot in an object does not store the actual slot value, but rather an explicit memory location object that, by default, stores only the actual slot value. It can be extended in ContextL layers to store more information as needed by the different STM strategies. A memory location object is an instance of the following layered class.

```
(define-layered-class transactional-slot-content ()
  ((v :layered-accessor transactional-slot-value)))
```

The value of such an explicit memory location object can be accessed via `transactional-slot-value`, which is in turn a layered function to enable more fine-grained context-dependent slot access semantics.

Defining slot access semantics In order to be able to define STM-specific semantics for memory accesses, it is important that such memory accesses can be exposed as layered functions so that their behavior can be modified in a context-oriented style. Fortunately, ContextL provides a layered slot access protocol, which is an extension of the slot access protocol of the underlying CLOS Metaobject Protocol (CLOS MOP, [13]), and which is indeed a layered protocol for defining context-dependent slot access semantics.

In brief, the lowest level of slot access defined by CLOS is provided by the reader function `slot-value` and its corresponding writer. When passed an object and a slot name as parameters, `slot-value` returns the corresponding slot value. So for example, `(slot-value object 'name)` returns the slot `name` of an object, and corresponds to low-level field accesses in other languages, like for example `object.name` in Java. The CLOS MOP enables intercepting such slot accesses by defining methods on the reader `slot-value-using-class` and its corresponding writer. The function `slot-value` is specified to be implemented as follows.[4]

```
(defun slot-value (object slot-name)
  (slot-value-using-class
    (class-of object) object slot-name))
```

The corresponding writer is defined in an analogous way. Such a slot access protocol allows CLOS programmers to redefine slot access semantics for user-defined metaclasses, for example to provide seamless integration of object-relational mappings [17].

In ContextL, the slot access protocol is extended in such a way that the default definition for `slot-value-using-class` in turn calls `slot-value-using-layer` (and the same for their corresponding writers). The layered function `slot-value-using-layer` allows associating one's own slot access sematics with layers, which can be used to implement STM-specific slot access semantics in CSTM. In fact, CSTM does not need to provide any further hooks for modifying slot accesses. In Section 4, we show in detail how this feature of ContextL is used for STM.

Defining commit and rollback semantics The default semantics for atomic blocks in CSTM do not require specific semantics for commit and rollback, since the coarse-grained locking semantics already ensure sequential execution of all atomic blocks. However, since commits and rollbacks are typically used in STM strategies, they are provided as layered functions `commit-transaction` and `roll-back`. By default, they just throw exceptions since they do not have useful semantics in the default STM strategy, but they can be extended in a context-oriented style for particular STM strategies. Furthermore, a `retry` exception class and a plain (non-layered) function `retry-transaction` is provided that calls `roll-back` and throws a retry exception. Again, these are utility definitions which are typically needed in STM strategies.

The Layers in CSTM CSTM defines a layer `stm` that is globally active by default. It modifies the slot access protocol of CLOS in such a way that all slot accesses transparently assume the presence of explicit memory locations of class `transactional-slot-content`. For this purpose, methods on `slot-value-using-layer` and its corresponding writer are defined as follows.

```
(define-layered-method slot-value-using-layer
  :in-layer stm :around
  ((class transactional-class) object slot)
  (transactional-slot-value (call-next-method)))

(define-layered-method (setf slot-value-using-layer)
  :in-layer stm :around
  (new-value (class transactional-class) object slot)
  (let* ((new-loc (make-instance
                   'transactional-slot-content
                   :value new-value))
         (return-loc (call-next-method
                      new-loc class object slot)))
    (transactional-slot-value return-loc)))
```

In the reader `slot-value-using-layer`, an invocation of `call-next-method` returns the current memory location object, and by calling `transactional-slot-content` on the returned location, we can retrieve the actual slot contents. In the writer `(setf slot-value-using-layer)`, the new value to be assigned to the slot is wrapped in a fresh instance of `transactional-slot-content` and passed as a changed argument to `call-next-method`. As is customary in CLOS, we also return the value just

[4] However, an implementation of CLOS typically optimizes the invocation of `slot-value-using-class` away if no intercepting method is defined.

assigned from the writer, by accessing the value from the memory location returned by `call-next-method`.

On top of this default layer, CSTM defines two further abstract layers `stm-mode` and `transaction`, which are supposed to be subclassed by user-defined STM strategies. This reflects the necessity to provide two layers in an STM strategy: The main purpose of an *STM mode* layer is to define the semantics of 'regular' slot accesses (outside of transactions) and to define how transactional contexts are set up and torn down in `call-atomic`. A *transaction* layer defines the semantics of transactional slot accesses, and how commits and rollbacks are performed. The activation of a transactional context in the mode layer is typically achieved by dynamically scoped activation of the corresponding transaction layer in `call-atomic`.

This concludes the discussion of the extension interface of CSTM. In the following section, we illustrate how it can be used to define two well-known STM strategies.

4. Example Strategies in CSTM

4.1 Implementing a direct-update STM

Our first example STM is based on 2-phase locking with optimistic reads (as for example in BSTM [7]). Here, every read access to a shared memory location is just logged, but for write accesses, a transaction first needs to acquire an exclusive lock. When the lock is successfully acquired, the transaction first records a copy of the memory location's content, and then updates its content with the new value. A transaction releases all the locks it has acquired when it successfully finishes. However, at the end of a transaction, the STM algorithm first checks for *read-after-write* conflicts by ensuring that none of the memory locations that were read were updated in other transactions afterwards. When there are no such conflicts, the transaction can indeed commit and release all its locks. Otherwise, in case a *read-after-write* conflict is detected, the transaction rolls back, reverts all of the write accesses it performed, releases its locks, and eventually restarts. *Write-after-write* data races are avoided due to the locks aquired for write accesses, which block other transactions from performing further write accesses before the current transaction is finished.

Direct-update layers We define two layers `direct-update-mode` and `direct-update-transaction` that inherit from the corresponding abstract layers provided by CSTM.

```
(deflayer direct-update-mode (stm-mode))
(deflayer direct-update-transaction (transaction))
```

Setting up and tearing down a direct-update transactional context requires preparing read and write sets for the current transaction, such that information about read and written slots can be recorded; activating the direct-update transaction; performing a commit (with implicit *read-after-write* checks); ensuring the release of all locks taken by the transaction; and dealing with attempts to retry a transaction.

```
(define-layered-method call-atomic
  :in-layer direct-update-mode (thunk)
  (handler-case
    (let ((*read-set* (make-hash-table))
          (*write-set* (make-hash-table)))
      (with-active-layers (direct-update-transaction)
        (unwind-protect
            (let ((result (funcall thunk)))
              (commit-transaction)
              result)
          (release-locks))))
    (retry () (call-atomic thunk))))
```

This layered method for `call-atomic` is associated with the `direct-update-mode`. It sets up an exception handler for `retry` exceptions using `handler-case`[5]; it binds the dynamically scoped variables `*read-set*` and `*write-set*` to fresh hash tables for recording read and write access information; it activates the `direct-update-transaction` layer with dynamic scope; it sets up an unwind handler to ensure that locks are always released using `unwind-protect`[6]; and it ensures that, after executing the code provided by the user, `commit-transaction` is called, which implicitly also performs the *read-after-write* checks.

Direct-update memory locations Memory locations in direct-update mode require two more pieces of information apart from the actual slot value: the version of a memory location and a lock that can be used for performing write accesses. This can be achieved in a straightforward way by extending the layered class `transactional-slot-content` for the layer `direct-update-mode` as follows.

```
(define-layered-class transactional-slot-content
  :in-layer direct-update-mode ()
  ((version :initform 0 :accessor slot-version)
   (lock :initform (make-lock) :reader slot-lock)))
```

Direct-update slot accesses The slot access semantics for direct-update mode, when there is no transaction active in the current thread, is not well-defined in the general case, since other threads may have left a slot in an intermediate state as part of an ongoing transaction. However, it is useful to be able to read and write slots when there is no transaction currently active at all, for example for performing initialization of as-yet unshared objects. For this reason, we leave slot access semantics in direct update mode outside of transactions untouched, and consider it the responsibility of the programmer to use unmonitored slot accesses wisely.

The slot access semantics for direct-update *transactions*, though, is well-defined: When reading a slot during a transaction, we have to register the current version of the slot such that we can perform a *read-after-write* check on a later commit. For that purpose, we define a method on the layered function `slot-value-using-layer` as follows.

```
(define-layered-method slot-value-using-layer
  :in-layer direct-update-transaction
  ((class transactional-class) object slot)
  (let ((loc (call-next-method)))
    (if (not (gethash loc *read-set*))
      (setf (gethash loc *read-set*)
            (slot-version loc)))
    loc))
```

The value returned by `call-next-method` is an explicit memory location object, and this is also what we need to return from this particular method (cf. Section 3.3).

Writing a slot during a direct-update transaction is more complex: We first have to retrieve the memory location object and then attempt to grab its lock, before performing a read ceck, recording the current value for later recovery on roll-back, and finally setting the actual slot value.

[5] similar to Java's `try-catch`

[6] similar to Java's `try-finally`

```
(define-layered-method (setf slot-value-using-layer)
  :in-layer direct-update-transaction
  (new-loc (class transactional-class) object slot)
  (let* ((old-loc (with-inactive-layers
                    (transaction stm-mode stm)
                    (slot-value object slot)))
         (locked (try-lock (slot-lock old-loc))))
    (if (not locked) (retry-transaction))
    (let ((read-version
           (gethash old-loc *read-set*)))
      (if read-version
        (if (> (slot-version old-loc) read-version)
          (retry-transaction)
          (remhash old-loc *read-set*))))
    (if (not (gethash old-loc *write-set*))
      (setf (gethash old-loc *write-set*)
            (cons
             (with-inactive-layers (transaction)
               (slot-value object slot))
             (slot-version old-loc))))
    (setf (slot-value old-loc 'value)
          (slot-value new-loc 'value))
    old-loc))
```

This method first reads the raw memory location object. This is achieved by temporarily deactivating all layers that are involved in the STM machinery, so that no read accesses are recorded and no conversion to actual slot values takes place. The method then tries to grab the lock of the memory location object. If it fails to do so, it immediately retries the transaction. (A more sophisticated contention manager may try to force the conflicting transaction to roll back here based on some criteria.) We then perform a check to ensure that some other transaction has not successfully committed a write access to the slot in question since the last time the current transaction has read the slot. If the check fails, we again retry the current transaction, otherwise we remove the slot from the current read set. We then record in the write set both the current slot value – here we need the actual value, not the memory location object, but without recording another read access – and the current slot version. Finally, we can perform the actual write access by directly modifying the memory location object we just locked. The memory location object passed by the underlying `stm` layer holding the new value is silently discarded.

Direct-update commit and rollback A commit during a direct-update transaction first has to check whether all slot values accessed in read mode still have the same version as when they were first read. Otherwise they were updated by some other transaction in between and are not valid anymore. Only then can the version of the slot be increased to signal a successful commit to other transactions.

```
(define-layered-method commit-transaction
  :in-layer direct-update-transaction ()
  (maphash (lambda (loc read-version)
             (if (> (slot-version loc) read-version)
               (retry-transaction)))
           *read-set*)
  (maphash (lambda (loc info)
             (incf (slot-version loc)))
           *write-set*))
```

A rollback for a direct-update transaction simply has to go through all the slots accessed in write mode and write back the previously recorded old values of those slots. However, this must not happen when other threads already successfully committed a write access to the slot in question in the meantime.

```
(define-layered-method roll-back
  :in-layer direct-update-transaction ()
  (maphash (lambda (loc info)
             (let ((old-value (car info))
                   (write-version (cdr info)))
               (if (not (> (slot-version loc)
                           write-version))
                 (setf (slot-value loc 'value)
                       old-value))))
           *write-set*))
```

Both for commits and rollbacks, the direct-update mode ensures in `call-atomic` above that all locks grabbed by a direct-update transaction are released by invoking `release-locks`. That plain function just iterates over the write set for performing this task.

```
(defun release-locks ()
  (maphash (lambda (loc info)
             (unlock (slot-lock loc)))
           *write-set*))
```

This concludes the implementation of the STM algorithm based on 2-phase locking.

4.2 Implementing a deferred-update STM

The second example we illustrate is a lock-free, deferred-update STM that implements a nonblocking synchronization strategy, as originally implemented in the DSTM system by Herlihy et al. [9]. In DSTM, a memory location does not store only *one* single current value, but *two* different such values, one of which is considered *new* and the other *old*. On top of that, a third value indicates which of the two former values should be considered valid. That third value is a reference to the transaction that did the last write to the memory location, and depending on the state of that transaction – which can be *active*, *committed* or *aborted* – either the old or the new value of the memory location is selected. By using two different values and selecting them based on the state of the transaction that performed the last write access, costly roll backs can be avoided (see below).

On each read access to a memory location, DSTM checks the state of the transaction that performed the last write. If it is committed, the new value is returned. Otherwise, if it is aborted, the old value is returned. In both cases, the read is successful and recorded in the current read set. However, if that transaction is not the current one but still active, it still uses the memory location. Therefore, none of the two values can be considered valid, but a conflict resolution must be performed to determine whether the current or the other transaction needs to be aborted.

Likewise, on each write access, DSTM also checks the state of the transaction that performed the last write. If it is either committed or aborted, a new memory location object is created, where the old content is the current memory location's *valid* content (the *new* content for a committed transaction, the *old* content for an aborted transaction), and where the new content is the value to be written. Additionally, the reference to the transaction that performed the last write is set to the current transaction performing the write access. Finally, using a *compare-and-swap* operation, the old memory location object is atomically replaced by the new one.[7] Again, if the transaction that performed the last write to the old memory location object is still active, a conflict resolution strategy determines which of the two involved transactions to abort.

DSTM checks for data races in two places: Firstly, on every read access to a memory location, *write-after-read* races are handled by

[7] *compare-and-swap* is a common hardware primitive that atomically compares the content of a memory location with an old value, and if they are the same, changes that memory location's content to a new value.

checking that the transaction that performed the last write access is not active. Otherwise, that transaction wrote a value that still needs to be read by the current transaction. Secondly, on every attempt to commit a transaction, *read-after-write* races are handled by ensuring that none of the memory locations recorded in the current read set were updated by other transactions in between. If that check fails, the current transaction has to abort, but does not have to revert any of the write accesses it performed: Since the memory locations in question still have copies of their old contents, future accesses will return these old contents due to the aborted state of the current transaction (which indeed performed the last writes to these locations). *Write-after-write* races do not need to be explicitly checked, since a transaction can only obtain write access to a memory location when no other transaction is actively using it, and then remains active itself until it commits or aborts.

Deferred-update layers We define two layers `deferred-update-mode` and `deferred-update-transaction` that inherit from the corresponding abtsract layers provided by CSTM.

```
(deflayer deferred-update-mode (stm-mode))
(deflayer deferred-update-transaction (transaction))
```

Setting up and tearing down a deferred-update transactional context is very similar to the direct-update case: It requires preparing a read set (but no write set) for the current transaction; activating the deferred-update transaction; performing a commit (with implicit *read-after-write* checks); and dealing with attempts to retry a transaction. Note that deferred-update transactions do not use locks, so no locks need to be released here. Instead, we need to bind a dynamically scoped `*current-tx-state*` variable to a new instance representing the current transaction state.

```
(define-layered-method call-atomic
  :in-layer deferred-update-mode (thunk)
  (handler-case
    (let ((*current-transaction-state*
           (make-instance 'transaction-state
                          :state :active))
          (*read-set*
           (make-hash-table :test #'equal)))
      (with-active-layers
           (deferred-update-transaction)
        (let ((result (funcall thunk)))
          (commit-transaction)
          result)))
    (retry () (call-atomic thunk))))
```

The variable `*current-tx-state*` records whether the current transaction is committed, aborted or active. To enable performing cheap commits and rollbacks, the transaction state is stored indirectly in a transaction state object that is shared by all slots on which a transaction has performed a write access. Thus both commits and rollbacks are essentially simple assignments to a slot of that one object.

We define the transaction state as a regular class. We also define `*current-tx-state*` as a global variable that is initialized such that slot accesses outside of transactions are always considered as immediately committed.

```
(defclass transaction-state ()
  ((state :initarg :state :accessor tx-state)))

(defvar *current-tx-state*
  (make-instance 'transaction-state
                 :state :committed))
```

Deferred-update memory locations Memory locations in deferred-update mode require an additional slot for storing an old slot value that potentially still needs to be accessible, and a reference to a transaction state as defined above, which indicates whether the transaction that performed the last write access to the slot is committed, aborted or active. As for the direct-update mode above, we can just extend the layered class `transactional-slot-content` to add these pieces of information.

```
(define-layered-class transactional-slot-content
  :in-layer deferred-update-transaction ()
  ((old-value :initarg :old-value)
   (most-recent-tx-state
              :initform *current-tx-state*
              :reader most-recent-tx-state)))
```

Deferred-update slot accesses The slot access semantics for deferred-update mode, when there is no transaction active in the current thread, has to consider whether other threads currently perform any transactions. For read accesses, this means that we have to check on each access whether the transaction that performed the last write on a slot is committed or aborted – in that case we can just return the current or old value – or whether it is still active, in which case we can either wait until it commits or aborts, or we can try to force it to abort to be able to make progress ourselves.

Since the slot access methods defined for the `stm` layer require that slot access methods in `stm-mode` layers receive and return instances of `transactional-slot-content`, we have to define methods both for reading the transactional slot content itself, as well as for the actual slot read accesses.

Recall that the accessor for the value of a transactional slot content is itself defined as a layered function. This allows defining a layered method associated with `deferred-update-mode` for it.

```
(define-layered-method transactional-slot-value
  :in-layer deferred-update-mode
  ((loc transactional-slot-content))
  (case (tx-state (most-recent-tx-state loc))
    (:committed (slot-value loc 'value))
    (:aborted (slot-value loc 'old-value))
    (:active (if (eq (most-recent-tx-state loc)
                     *current-tx-state*)
                 (slot-value loc 'value)
                 (retry-transaction)))))
```

Here, when the state of the most recent transaction is committed, the current value of the slot content is returned, whereas if that transaction is aborted, the old value of the slot content is reused. When the state of the most recent transaction is active, but it is actually the current transaction that performed the update, then we can also just return the current value.[8] If the state is active, but some other transaction has updated the object, we have no choice but to abort the current transaction, since this method does not have enough information to determine the original slot that contained this particular slot content object, which is necessary for negotiating more complex contention resolution. (As we show below, slot updates are performed by replacing full slot content objects in deferred-update transactions!) Note that an attempt to retry a transaction will throw an error outside of transactions, since there is no transaction to meaningfully abort, but that such an attempt is useful semantics under transactions.

The method for `slot-value-using-layer` associated with `deferred-update-mode` is very similar, except that it knows about the object that stores a reference to the memory location,

[8] Note that this case occurs only under transaction: Outside of transactions, the current transaction is always considered committed (see above).

so it has a chance to wait for it to be updated in case of contention (when another transaction that performed the last write to the slot in question is still active).

```
(define-layered-method slot-value-using-layer
  :in-layer deferred-update-mode
  ((class transactional-class) object slot)
  (let ((loc (call-next-method)))
    (case (tx-state (most-recent-tx-state loc))
      (:committed loc)
      (:aborted loc)
      (:active
       (if (eq (most-recent-tx-state loc)
               *current-tx-state*)
         loc
         ... else resolve contention ...)))))
```

We do not discuss contention managament here in detail. What can be done, for example, is to retry accessing the memory location object (as returned by `call-next-method`) until its corresponding transaction is either committed or aborted. If this also fails, we can then try to determine which of the two involved processes is older, and give it priority to ensure progress by forcing an abort of the other. Note that, like above, an attempt to eventually retry the current transaction will throw an error outside of transactions.

Write access to slots for direct-update mode is also very similar: Again, when the transaction that performed the last write is committed or aborted, we can just invoke `call-next-method` to perform the actual assignment. When that transaction is still active, we need to resolve the contention in some way before we can either perform the write access or decide to abort. (This definition does not need to consider semantics when we are inside an atomic block, because this will be handled in a separate method below.)

```
(define-layered-method (setf slot-value-using-layer)
  :in-layer deferred-update-mode
  (new-loc (class transactional-class) object slot)
  (let ((old-loc (with-inactive-layers
                   (transaction stm-mode stm)
                   (slot-value object slot))))
    (case (tx-state (most-recent-tx-state old-loc))
      (:committed (call-next-method))
      (:aborted (call-next-method))
      (:active ... else resolve contention ...))))
```

Like with the two read methods above, it can happen that an attempt to write to a slot that is in active use by another process may result in an error, because there is no transaction to be aborted outside of transactions.

Under transaction, the deferred-update strategy requires modified slot access semantics. For read accesses, we need to record the value just read in the current read set. Apart from that, the read method for `deferred-update-mode` is already sufficient, and can just be invoked using `call-next-method`.

```
(define-layered-method slot-value-using-layer
  :in-layer deferred-update-transaction
  ((class transactional-class) object slot)
  (let ((current-loc (call-next-method))
        (recorded-loc
         (gethash (list object slot) *read-set*)))
    (if recorded-loc
      (if (not (eq recorded-loc current-loc))
        (retry-transaction))
      (setf (gethash (list object slot) *read-set*)
            current-loc))
    current-loc))
```

For write access, we try to store information about the old slot content in the new memory location object we receive, and then attempt to replace the old memory location object by way of `compare-and-swap`.

```
(define-layered-method (setf slot-value-using-layer)
  :in-layer deferred-update-transaction
  (new-loc (class transactional-class) object slot)
  (letrec
    ((perform-write ()
       (let* ((old-loc
               (with-inactive-layers
                   (transaction stm-mode stm)
                 (slot-value object slot)))
              (rec-loc
               (gethash (list object slot)
                        *read-set*)))
         (if rec-loc
           (if (not (eq rec-loc old-loc))
             (retry-transaction)))
         (case (tx-state
                (most-recent-tx-state old-loc))
           ((:committed :aborted)
            (setf (slot-value new-value 'old-value)
                  (transactional-slot-value old-loc))
            (if (not (compare-and-swap
                      object slot old-loc new-loc))
              (perform-write))
            new-loc)
           (:active
            (cond
              ((eq (most-recent-tx-state old-loc)
                   *current-tx-state*)
               (setf (slot-value old-loc 'value)
                     (slot-value new-loc 'value))
               old-loc)
              (t ... resolve contention ...)))))))
    (let ((result-loc (perform-write)))
      (remhash (list object slot) *read-set*)
      result-loc)))
```

In this method, we first get hold of the memory location object currently stored in the slot, as above by deactivating all layers involved in the transaction machinery. We then perform a check that the object we just replaced is the same as a memory location object possibly previously recorded by a read access. If that is not the case, the current write access is invalid and we have to roll back.

Otherwise, we proceed by checking whether the transaction that performed the last write to the slot in question is committed or aborted. In that case, we prepare the new memory location we just received by storing as its old value the value of the current memory location. (For that purpose, we use `transactional-slot-value`, which already returns its new or old value correctly depending on its transaction state.) We then attempt to atomically replace the old memory location object by the new one using `compare-and-swap`, which returns true if the old location object was still stored in the slot in question and thus was successful at replacing it, or returns false otherwise, because some other transaction replaced the memory location object in the meantime in some other process. In the latter case, we try another attempt to replace the memory location, until we succeed.

In case the transaction that performed the last write to the slot in question is still active, it may just be the current transaction. In that case, we can just overwrite the current value of the 'old' memory location by the current value of the 'new' memory location: Since no other transaction will attempt to replace a memory loca-

tion owned by an active transaction, we do not need to take further precautions here. If the active transaction is some other transaction than the current one, we have to negotiate with that other transaction to resolve the conflict.

After a successful write (in case a `compare-and-swap` succeeded, or in case the current transaction owned the 'old' location anyway), we can remove the entry for this particular slot from the read set before we return the now current memory location.

Deferred-update commit and rollback A commit during a deferred-update transaction has to check whether all memory locations accessed in read mode are still the same as currently stored in the respective slots. Otherwise they were updated by some other transaction in between and are not valid anymore. Afterwards, the state of the current transaction can be set to committed, by performing a `compare-and-swap` from active to committed. It is necessary to use a `compare-and-swap` operation here because some other transaction may have forced the current transaction to abort as part of contention management in between, so we have to ensure that the state is active immediately before we change it to committed.

```
(define-layered-method commit-transaction
  :in-layer deferred-update-transaction ()
  (maphash
   (lambda (hash-key rec-loc)
     (let* ((object (first hash-key))
            (slot (second hash-key))
            (cur-loc (with-inactive-layers
                       (transaction stm-mode stm)
                     (slot-value object slot))))
       (if (not (eq cur-loc rec-loc))
         (retry-transaction))))
   *read-set*)
  (if (not (compare-and-swap
            *current-tx-state* 'state
            :active :committed))
    (retry-transaction)))
```

Aborting a transaction is even easier: It consists of simply setting the state of the current transaction to aborted.

```
(define-layered-method roll-back
  :in-layer deferred-update-transaction ()
  (setf (tx-state *current-txstate*) :aborted))
```

This concludes the implementation of the STM algorithm based on deferred updates.

4.3 First Benchmarks

In order to convince ourselves of the viability of our approach, we have performed some first benchmarks using the two example strategies above. A major issue is that most benchmarks for parallel programming are nowadays typically developed for static languages like C++ and Java, rather than for dynamic languages like Smalltalk and Lisp, so a major part in our effort consisted in meticulously translating existing benchmarks from (in our case) C++.

We are aware of two benchmark suites that can be used for testing software transactional memory, STAMP [2] and Lonestar [14]. While the algorithms in the Lonestar benchmark suite are known to yield suboptimal performance when software transactional memory is used to coordinate concurrent threads [15], it is easier to translate those benchmarks to Common Lisp / CLOS due to the fact that they are expressed in an object-oriented style. In contrast, the STAMP benchmark suite primarily consists of imperative algorithms operating on large arrays of basic value types.

We have selected the Delaunay Mesh Refinement algorithm from the Lonestar suite for testing purposes, which is the first algorithm discussed in [14]. The input for that algorithm is a triangulation of a set of points in a plane, with some triangles marked as "bad" according to some quality criterion. The algorithm operates on the direct environments of the bad triangles, and attempts to improve the quality by retriangulating those areas. Delaunay Mesh Refinement can in principle easily be parallelized by operating on several bad triangles in parallel, but this may cause conflicts when two threads happen to operate on overlapping environments. STM is an appropriate low-level approach for resolving such conflicts.

We have run Delaunay Mesh Refinement with CSTM in default mode using the coarse-grained locking semantics, which effectively yields a sequential algorithm, as well as in direct-update mode and in deferred-update mode. We have chosen a medium-sized input consisting of 10156 triangles and 4837 bad triangles (data set "B" in the Lonestar benchmark suite). On an Intel Xeon chip (E5345) with eight cores at 2.33 GHz, using LispWorks 6.0 with support for symmetric multiprocessing,[9] we achieve an average runtime of ca. 4 minutes in default mode (using one thread), ca. 4:30 – 4:40 minutes in direct-update mode (using two, four, eight and sixteen threads), and ca. 4:50 in deferred-update mode (using two, four, eight and sixteen threads).

The fact that the multi-threaded executions are slower than the single-threaded ones, and stay relatively constant independent of the number of threads, confirms the observations about Delaunay Mesh Refinement and similar algorithms reported previously [15]. The main reason is that in those algorithms, *too many* read accesses are monitored and eventually cause rollbacks, although these rollbacks are conceptually not necessary. It is important to keep in mind that STM is primarily concerned with correctness, not with speed. Performance improvements come from choosing and fine-tuning the right parallelization strategies. For example, the unnecessary rollbacks above can be avoided by describing the coordination in such algorithms at a level of abstraction higher than low-level memory accesses [15].

The fact that direct-update mode performs somewhat better than deferred-update mode confirms previous reports about improved performance of direct-update mode over deferred-update mode [9].

5. Discussion and Related Work

CSTM exposes a relatively straightforward extension interface for integrating new STM strategies as layers that cut across both read and write accesses to slots in a program, as well as the atomic blocks that need to be inserted in the right places of a program, but can otherwise remain unaware of the concrete STM strategy in use. The only modification for a program to take advantage of transactional semantics based on CSTM is that class and slot definitions must be marked as transactional.

The extension interface can be used in a context-oriented style: New information can be inserted into explicit memory locations simply by adding to the existing definition of `transactional-slot-content` of the base `stm-mode` layer of CSTM. The activation of the slot access semantics as well as commit and rollback semantics, which are both context-dependent in the sense that they are different inside and outside of transactions, perfectly align with the dynamic extent of atomic blocks.

Since the transaction layers are repeatedly activated and deactivated when entering and leaving atomic blocks, it is important that layer activation and deactivation do not cause serious overhead. However, we have already reported that these operations can be implemented efficiently [4], and ContextL uses the optimizations described in that paper.

It is noteworthy that the direct-update and deferred-update modes described above are implemented purely in terms of CSTM's

[9] For LispWorks ®, see http://www.lispworks.com.

extension interface. This gives us confidence that other STM strategies can also be implemented on top of CSTM. The core reason why this is possible is the fact that memory locations are modelled as explicit entities in CSTM, an approach we have already discussed as part of an interpreter-based solution before [10].

Although our approach is based on ContextL, we expect it to be transferable to other COP languages as well, such as the ones discussed in [1]. It should also be possible to transfer our approach to AOP languages with support for dynamic aspect weaving.

STM implementations for dynamic languages have already been suggested before: Renggli and Nierstrasz discuss an STM for Smalltalk [18], while Clojure is a Lisp dialect on top of the Java Virtual Machine that supports STM [11]. Both approaches implement each one particular STM approach and do not provide a dedicated extension interface for plugging in one's own STM strategy. Gonzalez, Denker and Mens also discuss a sketch of an STM implementation based on their own context-oriented programming language Ambience, again providing one particular STM strategy without providing a more general extension interface [5].

6. Conclusions and Future Work

Software transactional memory can be regarded as a concern that cuts across various operationally distinct events in a program, that is, read and write accesses to slots, entering and leaving atomic blocks, and commits and rollbacks. However, existing STM implementations and frameworks have not yet accounted for the dynamic crosscutting nature of STM. Based on our own previous work on simplifying the extension interface of an STM framework based on explicit memory locations, we have presented a context-oriented design for an STM framework. The essential insight is that user-defined modifications of the operations relevant for implementing STM can be associated with layers, whose activations and deactivations can be aligned with the atomic blocks defined in a base program. Especially read and write accesses to slots are highly context-dependent, in that they differ inside and outside of transactions, and ContextL's extension of the slot access protocol of the CLOS MOP proves to be a valuable tool for expressing this context dependency, a feature we have discussed in this paper for the first time. Finally, we have shown how two widely known STM strategies can be implemented in our approach, and have gained some confidence in the viability of our approach by running some first preliminary benchmarks.

The user interface of CSTM is very straightforward and requires users only to use `define-transactional-class` for defining classes with transactional slots, and to use `atomic` blocks for wrapping transactions. The extension interface is also relatively straightforward and requires the implementer of an STM algorithm only to inherit from two abstract layers, define how transactional contexts are set up and torn down, extend an explicit representation of memory locations with further information, define both regular and transactional accesses to such memory locations, and define additional actions and checks to be performed on commits and rollbacks. An open research question, though, is how to deal with different STM algorithms being active at the same time in different threads when they access the same shared memory locations.

As next steps, we plan to perform more extensive benchmarks by translating more of the known test suites for STM, more specifically the STAMP test suite [2]. It would then be possible and interesting to see how well CSTM compares to implementations based on C++ and Java. We also plan to implement more STM strategies, for example Clojure's approach based on multiversion concurrency control and snapshot isolation [11].

Acknowledgments We thank Dave Fox, Usha Millar and Martin Simmons from LispWorks ® for letting us use an alpha version of LispWorks 6.0. This work is partially funded by the Research Foundation – Flanders (FWO). Charlotte Herzeel's research is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

[1] M. Appeltauer, R. Hirschfeld, M. Haupt, J. Lincke, and M. Perscheid. A Comparison of Context-oriented Programming Languages. In *International Workshop on Context-oriented Programming, co-located with ECOOP 2009*. ACM Digital Library, 2009.

[2] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[3] P. Costanza and R. Hirschfeld. Language constructs for context-oriented programming: an overview of ContextL. In *DLS '05: Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10, New York, NY, USA, 2005. ACM Digital Library.

[4] P. Costanza and R. Hirschfeld. Reflective layer activation in ContextL. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied Computing*, pages 1280–1285, New York, NY, USA, 2007. ACM.

[5] S. Gonzales, M. Denker, and K. Mens. Transactional Contexts: Harnessing the Power of Context-Oriented Reflection. In *International Workshop on Context-oriented Programming, co-located with ECOOP 2009*. ACM Digital Library, 2009.

[6] T. Harris and K. Fraser. Language Support for Lightweight Transactions. *OOPSLA'03, Proceedings*, 2003.

[7] T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. *PLDI'06, Proceedings*, 2006.

[8] M. Herlihy, V. Luchanco, and M. Moir. A Flexible Framework for Implementing Software Transactional Memory. In *OOPSLA 2006, Proceedings*, 2006.

[9] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *PODC '03, Proceedings*, 2003.

[10] C. Herzeel, P. Costanza, and T. D'Hondt. Reusable building blocks for software transactional memory. In *Second European Lisp Symposium (ELS'09)*, 2009.

[11] R. Hickey. Clojure. http://clojure.org/.

[12] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-Oriented Programming. *Journal of Object Technology (JOT)*, 7(3):125–151, 2008.

[13] G. Kiczales, J. des Rivieres, and D. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, USA, 1991.

[14] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A Suite of Parallel Irregular Programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[15] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, New York, NY, USA, 2007. ACM.

[16] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Publishers, USA, 2007.

[17] A. Paepcke. PCLOS: Stress Testing CLOS – Experiencing the Metaobject Protocol. *OOPSLA/ECOOP '90, Proceedings*, 1990.

[18] L. Renggli and O. Nierstrasz. Transactional memory in a dynamic language. *Computer Languages, Systems & Structures*, 35(1):21–30, 2009.

[19] M. F. Ringenburg and D. Grossman. AtomCaml: First-class Atomicity via Rollback. *ICFP'05, Proceedings*, 2005.

[20] N. Shavit and D. Touitou. Software Transactional Memory. In *PODC '95, Proceedings*, 1995.