# Reusable building blocks for software transactional memory

**Charlotte Herzeel, Pascal Costanza and Theo D'Hondt**
(Programming Technology Lab, Vrije Universiteit Brussel, Belgium
{charlotte.herzeel — pascal.costanza — tjdhondt } @vub.ac.be)

**Abstract:** Software transactional memory (STM) is a promising approach for coordinating concurrent threads, for which many different implementation strategies are currently being researched. In this paper we show that if a language implementation provides *reflective access to explicit memory locations*, it becomes straightforward to both (a) build a STM framework for this language and (b) to implement STM algorithms using this framework. A proof-of-concept implementation in the form of a Scheme interpreter (written in Common Lisp) is presented.

## 1 Introduction

Software transactional memory (STM) [10] is a promising approach for coordinating concurrent threads. It proposes the use of a transactional model for coordinating reads and writes of shared data in a multithreaded system. Without such a mechanism, the (relative) order of these reads and writes is undefined, during the execution of which a program can cause problems if two threads try to write the same memory location. Such problems are known as *data races* and are traditionally dealt with by the programmer by using low-level mechanisms such as *locks* for controlling the progress of threads. Programming with locks is known to be difficult because the programmer can easily write code that introduces mistakes such as *deadlocks* or code that does not easily compose. STM alleviates many of these problems by offering a well-defined protocol for managing reads and writes of shared data automatically.

An efficient implementation of STM is however *hard*, and numerous strategies have been proposed, but there is no definitive winner [8]. For example, a STM's *transaction granularity* determines the unit of storage over which the system operates (object or word/pointer-based). Other design decisions include the use of pessimistic or optimistic concurrency control, early or late conflict detection, direct or deferred memory updates, and so on. For a detailed taxonomy we refer to Larus and Rajwar's work [8]. Each of these options results in a STM that performs better for different applications.

A number of *benchmark suites* have been developed for assessing the different variations of STM algorithms [1, 7]. Benchmark suites focus on getting comparable benchmark results, by providing sets of dedicated test applications that can be run without change for different STM algorithms. To make this work,

a benchmark suite defines a so-called "generic" STM interface that is used in those test applications, so that the implementations of the STM algorithm can silently vary underneath. However, such benchmark suites typically don't provide reusable building blocks for implementing the actual STM algorithms, but leave the programmers of such algorithms on their own. The latter is the focus of a *STM framework* that provides common STM functionality and hooks. In this paper we propose such a STM framework.

Herlihy et al. previously proposed a framework for STM [5], but their approach differs greatly from ours. Their framework, in line with other STM implementations we know of [6, 3, 8, 9], is built on top of an existing compiler that was not designed for supporting STM. In contrast, we start by designing a language architecture from scratch that exposes the hooks for supporting STM as a plugin. This simplifies both the implementation of the framework itself as well as the use of the framework for plugging in different STM algorithms.

The contributions of this paper are:

- an analysis of the hooks a language implementation needs to provide for implementing STMs as plugins, and our solution that proposes to provide reflective access to memory locations for this purpose;
- an interpreter framework with such explicit memory locations as a proof of concept, here implemented for Scheme, but transferable to other languages;
- an implementation of two example STM algorithms as extensions of this framework to validate our approach.

## 2 Concepts of Software Transactional Memory

### 2.1 Multiprocessing with shared memory

In multithreaded programs, the execution of threads is typically synchronized using *locks*, which is a mechanism for temporarily granting threads exclusive access to shared resources, for example shared memory locations. Though locks can be used to avoid data races, programming with locks is notoriously difficult and alternative synchronization strategies are still an important research topic[3]. Recently, software transactional memory was proposed.

The idea behind software transactional memory (STM) is to use transactions for coordinating the execution of concurrent threads [8]. "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 pretend to never have been executed at all (i.e. any side effects are undone). 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 do not lead to data races.

STM has been realized both as libraries [6, 5, 8] and language extensions [3, 9]. STM libraries offer programmers APIs for making transactions, while language support for STM typically consist of a keyword `atomic` for delimiting a block of code that needs to execute transactionally. For example, if Scheme had an `atomic` construct, then a thread-safe implementation of the `insert` operation for a double linked list could look like the code below. The underlying STM implementation assures the code inside `atomic` executes transactionally.

```
(define insert (node new-node)
  (atomic (set-previous new-node node)
          (set-next new-node (next node))
          (when (has-previous-p (next node))
            (set-previous (next node) new-node))
          (set-next node new-node)))
```

### 2.2 Structure of a STM implementation

A 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 makes sure the conflicting execution is undone by rolling back one of the transactions.

Larus and Rajwar divide STM implementations into two categories: Deferred-update and direct-update STMs [8]. They differ strongly in the general implementation strategy. Deferred-update STM systems are implemented following a nonblocking synchronization strategy. When transactions access a memory location, they acquire a copy of its content and proceed execution in terms of the copy. Only when a transaction commits, the STM system replaces the content of the accessed memory locations with such copies. In case the STM detects a data race, transactions are cheaply rolled back, since their side effects are not yet global and hence do not need to be undone.

Conversely, direct-update systems rely on a blocking synchronization strategy. Transactions can temporarily get exclusive access to a memory location and side effects are performed instantly. A more expensive rollback mechanism than for deferred-update systems is needed, as the STM system needs to store old content of memory locations to be able to restore them on a rollback. However, in case there are few data race conflicts, such systems can be very efficient.

We claim that if a language implementation provides an explicit representation for memory locations, the implementation of both kinds of STMs is much easier to realize than when this is not the case. Consequently, we also claim that explicit memory locations are a key ingredient for a framework in which to express different kinds of STM. In what follows we sketch a design of a Scheme interpreter with explicit memory locations, and discuss the implementation of both a direct-update and a deferred-update STM on top of the memory location

abstraction. Afterwards, we discuss the inherent complexity of a STM implementation on top of a language implementation without an explicit representation for memory locations.

## 3 STM for a Scheme implemented in CLOS

Our experiment consists of extending a Scheme interpreter written in CLOS with explicit memory locations. The interpreter implements a non-trivial subset of Scheme. Additionally, it supports parallel variants of familiar constructs like `parallel-do`, `parallel-let`, etc as found in QLisp [2]. It also implements the `atomic` construct for executing a piece of code transactionally [3]. Our interpreter is written using LispWorks[1] and relies on its multiprocessing package for threading and locking functionality. Our implementation is primarily meant to illustrate our claims, but does not focus on efficiency. We will discuss efficiency concerns in the discussion section.

### 3.1 Transactional execution

Our interpreter extends the prototypical Lisp interpreter with a clause for evaluating *atomic* expressions. The code for `eval-atomic` is listed below. Note that `mp:*current-process*` is part of the LispWorks API for getting hold of the current active thread.

For evaluating an atomic expression, we put the current active thread into a transactional state (see `push-transactional-mode`) and let it evaluate the expression. Afterwards, `commit` is called for finalizing the transaction and restoring the thread to a non-transactional state for the rest of the execution (see `pop-transactional-mode`). For this, we made it possible to add a transactional state to LispWorks threads, which is accessible through the methods `push-transactional-mode`, `pop-transactional-mode` and `peek-transactional-mode`. The transactional state of a thread itself is modeled as a stack of transaction objects, for supporting the evaluation of nested atomic expressions.

```
(defmethod eval-atomic (exp env cont)
  (let ((transaction (make-transaction exp env cont)))
    (push-transactional-mode mp:*current-process* transaction)
    (eval expression environment #'commit)))

(defmethod commit (result)
  (funcall (cont (pop-transactional-mode mp:*current-process*)) result))
```

Transactions are modeled as objects that store a reference to their thread of execution, and the interpreter's state at the time the transaction is created. The latter is needed for rolling back a transaction:

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

```
(defmethod roll-back ()
  (let ((transaction (pop-transactional-mode mp:*current-process*)))
    (eval-atomic (atomic-block transaction) (env transaction) (cont transaction))))
```

The methods for commit and roll-back shown here provide the default implementations for these operations. They do not by themselves deviate from normal execution without transactions. However, by defining them as methods, we have established a protocol for transactional execution: Client code that extends our interpreter with STM can override these two methods to include the extra functionality required for committing or rolling back a transaction.

### 3.2 Memory locations as objects

Memory locations are modeled as instances of the class `memory-location`, which defines a slot for storing a memory location's content. The class can be extended to hold additional information necessary for implementing a particular STM.

A method `make-memory-location` is the constructor for making new memory location objects. It takes the content of the memory location as an argument. The methods `memory-location-value` and `(setf memory-location-value)` are used to respectively read and write the content of a memory location object. The methods `registered-read` and `registered-write` implement a read or write of a memory location that is registered by the STM. Both registered and non-registered accesses to memory locations are necessary because some internal memory accesses must *not* be registered to implement STM correctly.

## 4 Explicit memory allocation and access in Scheme

For implementing STM, it must be possible to advise all possible reads and writes of memory. For Scheme, this means it should be possible to extend reads and writes of *variables*, *cons cells* and *vectors*, that is, there are no other *primitive* means for allocating and accessing memory. Fig. 1 gives an overview of the constructs in Scheme for manipulating variables, cons cells and vectors. We next identify the methods in the interpreter that implement these operations and open them up for extension. An overview of these methods is shown in Fig. 2.

### 4.1 Variable allocation and access

Variable bindings are stored in an *environment* structure, a dictionary-like structure allocated on the Common Lisp heap that maps variable names onto values. Internally, the latter mappings are represented using a structure `mapping`.

Creating a new variable/value mapping (for interpreting a `define`) is handled by a method `add-binding`: It creates a new instance of the structure `mapping` and stores it into the global environment. Updating a variable/value mapping

| | Allocation | Reading | Writing |
|---|---|---|---|
| Variables | `(define x obj)` | `x` | `(set! x obj)` |
| Cons cells | `(cons obj obj)` | `(car cons-cell)` | `(set-car! cons-cell obj)` |
| | | `(cdr cons-cell)` | `(set-cdr! cons-cell obj)` |
| Vectors | `(vector size)` | `(vector-ref vector idx)` | `(vector-set! vector idx obj)` |

**Figure 1:** Allocating and accessing memory in Scheme

| | Allocation | Reading | Writing |
|---|---|---|---|
| Variables | `add-binding` | `binding` | `set-binding` |
| | `make-mapping` | | |
| Cons cells | `make-cl-cons` | `cl-list-car` | `cl-list-cdr` |
| | | `cl-list-set-car` | `cl-list-set-cdr` |
| Vectors | `make-cl-vector` | `cl-vector-ref` | `cl-vector-set!` |

**Figure 2:** Methods implementing memory allocation and access

(for interpreting a `set!`) is done by a method `set-binding`. Finally, looking up a variable (for interpreting a variable reference) is done by a method `binding`.

In a next step, we can now override these methods to make the memory locations referenced by variables explicit, by inserting explicit memory location objects in the mappings. The code for creating and accessing a mapping is appropriately changed:

```
(defmethod make-mapping :around ((atom atom) value)
  (list atom (make-memory-location value) 'mapping))

(defmethod memory-location-of-value (mapping) (second mapping)) ; new

(defmethod mapping-value :around (mapping)
  (memory-location-value (memory-location-of-value mapping)))

(defmethod (setf mapping-value) :around (value mapping)
  (setf (memory-location-value (memory-location-of-value mapping)) value))
```

Next, we override the methods `binding` and `set-binding`, which respectively implement variable lookup and update, to work on the new mappings. Variable lookup and update are operations that need to be monitored by the STM algorithm, hence the use of `registered-read` and `registered-write`, the methods we previously defined for monitored accesses.

```
(defmethod binding :around ((atom atom) (environment environment))
  (let ((mapping (binding-mapping atom environment)))
    (registered-read (memory-location-of-value mapping))))

(defmethod set-binding :around ((environment environment) (atom atom) (handle handle))
  (let ((mapping (binding-mapping atom environment)))
    (if mapping (registered-write (memory-location-of-value mapping) handle)
      (error "Cannot assign to an undefined variable"))
    mapping))
```

### 4.2 Vector allocation and access

Vectors are implemented using Common Lisp arrays, allocated on the Common Lisp heap. We represent vectors using a wrapper class `cl-vector` whose instances hold references to such Common Lisp arrays. A method called `make-cl-vector` is responsible for creating new vectors (for interpreting a `make-vector`). Methods `cl-vector-ref` and `cl-vector-set` implement reading and updating vector entries (for interpreting a `vector-ref` and `vector-set!`).

To allow advising of vector allocation and access, we now make the memory locations vectors reference explicit. In the code listed below, we override the constructor for vector objects. As previously discussed, vectors are represented by a class `cl-vector` that wraps a Common Lisp array. Here, we initialize the entries of the array with a memory location object:

```
(defmethod make-cl-vector :around (nr &optional initial-content)
  (if initial-content
      (make-instance 'cl-vector :cl-array (make-array nr :initial-contents
                                            (mapcar #'make-memory-location initial-content)))
    (make-instance 'cl-vector :cl-array (let ((new-array (make-array nr)))
                                          (dotimes (i nr)
                                            (setf (aref new-array i) (make-memory-location)))
                                          new-array))))
```

Next, we override `cl-vector-ref` and `cl-vector-set` to operate on the explicit memory locations:

```
(defmethod cl-vector-ref :around ((cl-vector cl-vector) nr)
  (let ((memory-location (aref (cl-array cl-vector) nr)))
    (registered-read memory-location)))

(defmethod cl-vector-set :around ((cl-vector cl-vector) nr val)
  (let ((memory-location (aref (cl-array cl-vector) nr)))
    (registered-write memory-location val)))
```

#### 4.2.1 Cons cells and other data structures

Cons cells are essentially vectors of fixed length two, so they are implemented in a similar fashion as `cl-vector`, namely by providing a Common Lisp class for wrapping Common Lisp cons cells and providing corresponding methods for the respective operations on pairs. Similarly to what we did for vectors and variables, memory locations are introduced into the cons cell implementation, so that cons cell accesses can be advised to implement STM. Other data structures, like classes, can be supported in a similar fashion as extensions of the interpreter, or can be built on top of vectors and closures as user code in Scheme itself.

## 5 Plugging in STM implementations

### 5.1 Implementing a direct-update STM

Our first example STM is based on 2-phase locking with optimistic reads (as for example used in BSTM [4]). When a transaction reads a memory location, the

transaction takes a note of this. For writing a memory location, a transaction needs to acquire an exclusive lock. On acquiring the lock, the transaction first records a copy of the memory location's content and only then updates its content with the new value. The lock is released when the transaction successfully finishes. The STM checks for data races at well-defined times. *Write-after-read* conflicts are checked on reading a memory location by verifying that no other transaction has a lock on it. Additionally, when a transaction finishes, the STM checks for *read-after-write* conflicts by checking that none of the memory locations read by the transaction were updated afterwards. When there are no conflicts, the transaction finishes (commits) and releases all of its locks. Conversely, when a conflict is detected, the transaction rolls back, undoes any of the writes it performed, releases its locks and restarts. *Write-after-write* data races are avoided as transactions have to acquire an exclusive lock for writing a memory location, and these locks are only released when a transaction commits.

### 5.1.1 Memory location and transaction extensions

From the description above we derive the following extensions to our interpreter. We extend memory locations with a *version* for making it possible to check on commit whether a read memory location was updated by comparing its current version with the version on read. A memory location's version consists of a counter and a reference to the transaction that performed the last write.

```
(defclass versioned-memory-location (memory-location)
  ((lock    :initform (mp:make-lock)          :accessor memory-location-lock)
   (version :initform (make-instance 'version) :accessor memory-location-version)))
```

The code listed above shows the implementation of the class `versioned-memory-location` that extends `memory-location` with slots for holding a lock and a version. The slots are initialized with default values, respectively a new lock and version object. The function `mp:make-lock` for creating a lock comes from the LispWorks `MP` package. The accessor `memory-location-value` for accessing a memory location's content remains unchanged. The constructor `make-memory-location` is overridden to create an instance of the class `versioned-memory-location`:

```
(defmethod make-memory-location :around (&optional value)
  (make-instance 'versioned-memory-location :value value))
```

We also extend transactions with a read and a write set. The sets are modeled as property lists, mapping each accessed memory location object onto a version object (in case of the read set) or a copy of the memory location's previous content (in case of the write set). Acessors `get-read-set` and `get-write-set` are defined for accessing the read or write set of a transaction.

### 5.1.2 Advising access of memory locations

In this STM, `registered-read` and `registered-write` are implemented as follows. `registered-read` makes a copy of the memory location's current version and, together with the memory location, pushes it onto the transaction's read set. Subsequently, it calls `locked-by-other-thread-p` to check if another thread holds a lock on the memory location: If so, there is a potential *write-after-read* data race, and the transaction is rolled back. Otherwise, the memory location's content is returned.

```
(defmethod registered-read ((memory-location versioned-memory-location))
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (when transaction
      (setf (get-read-set transaction)
            (list* memory-location (duplicate (memory-location-version memory-location))
                   (get-read-set transaction)))
      (when (locked-by-other-thread-p memory-location)
        (roll-back)))
    (memory-location-value memory-location)))
```

`registered-write` calls `obtain-lock` for getting an exclusive lock on the memory location. Subsequently, it pushes the memory location and its current content on the transaction's write set. The latter is a sufficient "copy" since we register all memory writes by default. The next two expressions are responsible for increasing the version number and updating the "process-that-did-the-last-write." Subsequently the memory location's content is replaced by the new value.

```
(defmethod registered-write ((memory-location versioned-memory-location) value)
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (if transaction
        (obtain-lock memory-location
                     (lambda ()
                       (setf (get-write-set transaction)
                             (list* memory-location (memory-location-value memory-location)
                                    (get-write-set transaction)))
                       (setf (process-that-did-last-write (memory-location-version memory-location))
                             transaction)
                       (incf (version-nr (memory-location-version memory-location)))
                       (setf (memory-location-value memory-location) value)))
      (setf (memory-location-value memory-location) value))))
```

We also show the code for `obtain-lock` below, which tries to acquire the lock (via a call to `mp:process-lock`), but if that fails – because another transaction has the lock and waiting to get it takes too long – the transaction rolls back. `roll-back` removes the current transaction from the current process, undoes the writes it performed, releases its locks, removes the recorded read and write sets, and finally, the transaction is restarted (`call-next-method`).

```
(defmethod obtain-lock ((memory-location memory-location) cont)
  (let* ((lock (memory-location-lock memory-location))
         (lock-is-mine (mp:process-lock lock :timeout 3)))
    (if lock-is-mine (funcall cont) (roll-back))))

(defmethod roll-back :around ()
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (undo-writes-from-process transaction)
    (release-locks transaction)
    (call-next-method)))
```

The code for committing a transaction is shown below. It calls `verify-reads` to check for data races. Recall that the read set of a transaction is modeled as a property list: For each pair in that list, consisting of a memory location object and its version at the time it was read, we check if the current version of the memory location object is different from the old. If so, there is a data race and the commit fails.

```
(defmethod commit :around (result)
  (if (verify-reads)
      (progn
       (release-locks)
       (call-next-method))
   (roll-back)))

(defmethod verify-reads ()
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (loop for (memory-location version-on-read) on (get-read-set transaction) by #'cddr
          when (version-changed-p memory-location version-on-read) return nil
          finally (return T))))
```

This concludes the implementation of the STM algorithm based on 2-phase locking. Note that the implementation is purely an extension of our memory location and transaction abstractions: No other parts of the interpreter need to be changed to plug in the STM.

### 5.2 Implementing a deferred-update STM

The second example we implement is the DSTM system by Herlihy et al. [6]. It is a lock-free, deferred-update STM that implements a nonblocking synchronization strategy. In DSTM, a memory location does not store *one* content, but *two*. It also stores a reference to the transaction that did the last write of the memory location. Depending on the status of that transaction – "active," "aborted" or "committed" – one content takes the role of the memory location's content before the last write (the "old" content) and the other plays the role of its current content (the "new" content).

When a transaction reads a memory location, DSTM checks the status of the transaction that performed the last write. If its status is "aborted," then the memory location's "old" content is returned. Otherwise, if the status is "committed," then the "new" value is returned. In both cases, the read is successful and recorded in the transaction's read set. Finally, when the status is "active," the memory location is in use by another transaction, and a conflict resolution is started to see if that transaction can keep using it or hands it over.

Similarly, for writing a memory location, DSTM checks the status of the transaction that performed the last write. Again, when the status is "active," the transactions negotiate for "ownership" of the memory location. Otherwise, when the status of the transaction that performed the last write is "committed" (or "aborted"), a new memory location object is created, with as "old" content

a copy of the old memory location's "new" (or "old") content and as "new" content the write value. Additionally, the "transaction that did the last update" of the new memory location object is set to the transaction performing the write. Then, using a *compare-and-swap* operation, the old memory location object is atomically *replaced* by the newly created one.[2]

There are two places where DSTM checks for data races. On reading a memory location, DSTM checks for *write-after-read* data races by checking that the status of the transaction that did the last write is not "active." Otherwise, that transaction wrote a value the other transaction still had to read. On committing a transaction, DSTM checks for *read-after-write* data races by going through the transaction's read set and checking if any of the read memory locations was updated by another transaction in between. If so, the transaction aborts, setting its status to "abort." However, it is not necessary to undo any of the writes the aborted transaction performed, as the written memory locations still have a copy of the "old" content, and because the transaction's status is set to "abort," future accesses will return this "old" content. *Write-after-write* data races are avoided by DSTM, since a transaction can only obtain write access when no other transaction is actively using the memory location, and the write access is only given up when the transaction's status changes to "commit" or "abort."

### 5.2.1 Memory location and transaction extensions

Given the above description, we need to extend memory location objects and transactions as follows. We create a new class `dstm-memory-location` for representing DSTM memory locations. We do not add new slots, but the idea is that a memory location's content is an instance of the class `content-unit`, as shown in the code below (see the initialization of the slot `memory-location-value` in `make-memory-location`). The class `content-unit` is a container for holding the version, the "old" content and the "new" content of a memory location. For convenience, we define methods `version`, `memory-location-new-content` and `memory-location-old-content` for accessing the the latter three objects directly from a memory location object (not shown).

```
(defmethod make-memory-location :around (&optional value)
  (make-instance
   'dstm-memory-location
   :memory-location-value (make-instance 'content-unit :new-value value)))

(defclass content-unit ()
  ((version :initarg :version :initform (make-instance 'version) :accessor version)
   (new-content :initarg :new-content :initform nil :accessor new-content)
   (old-content :initarg :old-content :initform nil :accessor old-content)))
```

[2] *compare-and-swap* is a known hardware primitive that atomically compares the content of a memory location to a value, and if they are the same, changes the content of the memory location to a new, given value.

The `memory-location-value` reader is overridden as shown below. It dispatches on the status of the transaction that performed the last write: If the status is "committed," the transaction returns the "new" content. In case it is "aborted," it returns the "old content." Otherwise, when the transaction that performed the last write is still active, the current transaction negotiates with that transaction to obtain access to the memory location. In our current implementation, the negotiation strategy is to simply wait, but any other strategy can be implemented here. Also note that in case there is no transaction that performed the last write – when a variable was only initialized, but never written – then `memory-location-value` also returns the "new" content of the memory location, which is set when defining a variable.

```
(defmethod memory-location-value ((memory-location dstm-memory-location))
  (let ((last-writer (process-that-did-last-write (version memory-location))))
    (cond ((or (null last-writer) (committed-p last-writer))
           (memory-location-new-content memory-location))
          ((aborted-p last-writer)
           (memory-location-old-content memory-location))
          (t (negotiate-for memory-location last-writer)))))
```

We extend transactions with a read set and a status flag. A transaction can be in three states: When a transaction starts, its status is set to "active," on roll back it is set to "aborted" and on commit to "committed." The predicates `aborted-p`, `committed-p` and `active-p` are defined for querying the status of a transaction. Additionally, methods are defined for switching the status of a transaction (`change-status-to-aborted`, `change-status-to-committed` and `change-status-to-active`).

### 5.2.2 Advising access of memory locations

For DSTM, `registered-read` and `registered-write` are implemented as follows. The code for `registered-read` is quite straightforward: It makes an entry in the current transaction's read set, checks for (read after write) data races and if that succeeds, it returns the memory location's content, otherwise the transaction is rolled back:

```
(defmethod registered-read ((memory-location dstm-memory-location))
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (when transaction
      (setf (get-read-set transaction)
            (list* memory-location (duplicate (version memory-location))
                   (get-read-set transaction)))
      (if (verify-reads) (memory-location-value m)
        (roll-back)))
    (memory-location-value m)))
```

The implementation of `registered-write` is a bit more tricky due the use of the `mp:compare-and-swap` operation.[3] We first get hold of the memory location's content unit (see `content-unit-before-write`) and we create a new

[3] `mp:compare-and-swap` is introduced in LispWorks 6.0, which is not yet publicly available, but accessible to us for testing. An alternative implementation is to use a lock.

content unit by calling `make-new-content-unit-from`. Subsequently, we try to replace the memory location's content with `new-content-unit` through the call to `mp:compare-and-swap`. For this, the latter fetches again what is in the slot `memory-location-value` of the memory location object and compares it to the previously fetched `content-unit-before-write`. When the *compare-and-swap* fails, because these two do not point to the same object anymore, we know that in between time, the memory location's content was updated by another transaction. To resolve this, the current transaction is rolled back and restarted.

```
(defmethod registered-write ((memory-location dstm-memory-location) new-val)
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (if transaction
        (let ((content-unit-before-write (slot-value memory-location 'memory-location-value))
              (new-content-unit
               (make-new-content-unit-from content-unit-before-write new-val transaction)))
          (unless (mp:compare-and-swap (slot-value memory-location 'memory-location-value)
                                       content-unit-before-write new-content-unit)
            (roll-back)))
      (setf (memory-location-value memory-location) new-val))))
```

On committing a transaction, we verify if the transaction is involved in a *read-after-write* conflict: See the call to `verify-reads` in the code below. If there is no conflict, the transaction's status is changed to "committed", otherwise the the transaction is rolled back.

```
(defmethod commit :around (result)
  (if (verify-reads)
      (let ((transaction (peek-transactional-mode mp:*current-process*)))
        (change-status-to-committed transaction)
        (call-next-method))
    (roll-back)))
```

Rolling back a transaction in DSTM is quite cheap: We just change its status to "aborted." Then the transaction can safely restart. There is no need to roll back any of the side effects it performed, since memory locations store a copy of the content before the transaction performed any update, and that copy will from then on be accessed by other transactions.

```
(defun roll-back :around ()
  (let ((transaction (peek-transactional-mode mp:*current-process*)))
    (change-status-to-aborted transaction)
    (call-next-method)))
```

That's it for the implementation of the DSTM algorithm. We stress again that the implementation is purely an extension of our memory location and transaction abstractions, and that there is no other parts of the interpreter that needs to be changed to plug in the STM.

## 6 Discussion and Related Work

The implementation of the STMs we just discussed shows that it is indeed possible to use our memory location model for implementing STMs as plugins. We

now address our original claim, that it is much harder to implement STM as part of a language that doesn't provide reflective access to explicit memory locations.

Assume we try to implement STM on top of plain Common Lisp. Common Lisp provides some predefined data structures, like variables, cons cells, vectors, and arrays, and ways of defining new user-defined data structures using defstruct, defclass and define-condition. This means that the number of potential datatypes in Common Lisp is open-ended, which is true for most general-purpose languages.

It is possible to implement STM algorithms for Common Lisp by deciding to support one or more specific kinds of datastructures, for example by using custom slot accessors in the CLOS MOP[4], or by shadowing accessors for cons cells. However, because of the open-endedness of Common Lisp, such STM libraries can never provide complete coverage of all possible data structures.

This is also true for DSTM2, which is the only other framework with support for implementing STM algorithms we are aware of [5]. Their approach is implemented as a library for Java that takes advantage of Java's reflection capabalities and its class loader architecture to create new classes at runtime for speciallly annotated Java interface definitions. These new classes contain pairs of getter and setter methods with additional behavior as required by the various STM algorithms, much like the adaptations of accesses to memory locations that we described above. New STM algorithms can be plugged in that provide templates for new such getter and setter methods. However, such STM algorithms can only operate on *instance variables* of Java classes, but not, for example, on class variables or array entries. This restriction is due to the fact that Java does not provide reflective access to its internal representation of memory locations.

Our interpreter framework provides a single abstraction for memory locations, and guarantees that all memory accesses always go through a handful of well-defined accessor methods. So it is sufficient to override these accessor methods once to plug in new STM algorithms, without having to do this for each and every kind of data structure over and over again.

A current drawback of our approach is that it doesn't pay a lot of attention to efficiency concerns: It introduces overhead because of the wrapping of internal representations of data structures and because each memory location access goes through a generic function call. This is due to the fact that we focused on illustrating the essential idea of explicit memory locations as our primary first goal. It has been shown in the past that reflective architectures like the one presented in this paper can indeed be implemented efficiently, but it remains to be shown to what extent we can do this for our approach as well.

[4] See for example CL-STM http://common-lisp.net/project/cl-stm/

## 7 Conclusions and Future Work

In this paper we have shown that if a language implementation provides reflective access to explicit memory locations, it becomes straightforward to implement both (a) a framework for software transactional memory, and (b) different STM algorithms using this framework. We have presented a proof-of-concept implementation in the form of a Scheme interpreter with such explicit memory locations and subsequently implemented a deferred-update and a direct-update STM algorithm in terms of the memory location abstraction to back our claims.

Future work includes more experiments with other STMs, to confirm that our approach is stable enough for a wide range of such algorithms. We then intend to investigate efficient implementation techniques, by removing overhead that is caused (a) by unnecessary wrappers in the internal representation of basic data types and (b) by unnecessary generic function calls for accessing memory locations that are never accessed by more than one thread. A final path for future work is to implement some standard benchmarks for STMs. Even without a more efficient implementation, we can already gain interesting insights from them, by counting the number of unnecessary rollbacks under different STM algorithms and under different, simulated access patterns in competing threads.

## References


1. 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.
2. R. P. Gabriel and J. McCarthy. Queue-based multi-processing lisp. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 25–44, New York, NY, USA, 1984. ACM.
3. T. Harris and K. Fraser. Language Support for Lightweight Transactions. *OOPSLA'03, Proceedings*, 2003.
4. T. Harris, M. Plesko, A. Shinnar, and D. Tarditi. Optimizing Memory Transactions. *PLDI'06, Proceedings*, 2006.
5. M. Herlihy, V. Luchanco, and M. Moir. A Flexible Framework for Implementing Software Transactional Memory. In *OOPSLA 2006, Proceedings*, 2006.
6. M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software Transactional Memory for Dynamic-sized Data Structures. In *PODC '03, Proceedings*, 2003.
7. 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.
8. J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Publishers, USA, 2007.
9. M. F. Ringenburg and D. Grossman. AtomCaml: First-class Atomicity via Rollback. *ICFP'05, Proceedings*, 2005.
10. N. Shavit and D. Touitou. Software Transactional Memory. In *PODC '95, Proceedings*, 1995.