Software Transactional Memory for Implicitly Parallel Functional Language

Petr Krajca[*][†]
SUNY Binghamton, Watson School
Binghamton, NY 13902, U. S. A.
+1 (607) 777 5690
petr.krajca@binghamton.edu

Vilem Vychodil[‡]
SUNY Binghamton, Watson School
Binghamton, NY 13902, U. S. A.
+1 (607) 777 5690
vychodil@binghamton.edu

ABSTRACT

During the last decade, software transactional memory (STM) gained wide popularity in many areas of parallel computing. In this paper, we introduce LISP-derived language equipped with automatic parallel execution and mutations based on software transactional memory. The novel idea is that we do not incorporate transactions as a language construct but rather use them as a mean of runtime environment to run each computation in a consistent memory and thus provide correct results. STM enables us to efficiently manage concurrent object updates and resolve collisions. In this paper, we describe a variant of deferred lockless STM mechanism with direct updates that enables us to use mutations with minimal costs.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*; F.1.1 [**Computation by Abstract Devices**]: Models of Computation—*automata*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms

Design, Languages, Theory

Keywords

Software transactional memory, data parallelism, implicit parallelism, functional programming, Scheme.

[*]The authors' second affiliation is Dept. Computer Science, Palacky University, Olomouc, Czech Republic.

[†]`krajcap@inf.upol.cz`

[‡]`vilem.vychodil@upol.cz`

1. INTRODUCTION

Parallel computing is a well-established area of computer science and computer engineering although for many programmers the development of parallel applications is complicated due to the extra effort of keeping programs consistent. Programmers often have to use special means (e.g., locks, mutexes, semaphores,etc.) to ensure correct access to shared resources, e.g., memory, input, and output. Avoiding problems in parallel environments belong to the most complex tasks and their mishandling is a common source of malfunctions which are often hard to figure out.

A typical solution to difficulties related to parallel programming is to equip well-known programming languages with more or less abstract means that help the programmer to manage difficulties of parallelism. One such mean that seems to be very practical is the software transactional memory that deals with the memory similarly to the database transactions and allows to run parts of programs in parallel even if these parts are using shared objects.

Other possible solution to the issues of parallel programming is to leave all aspects of parallelization in runtime environment and let the users to write traditional *sequential programs*. We may call this way of running programs an implicit parallelism. The key benefit of the implicit parallelism is in the significantly less effort needed to write parallel programs. This approach to parallel programming will probably gain in importance with the aim to exploit the current hardware trend that shifts from increasing clock speed to processors able to process multiple threads simultaneously.

The main goal of this paper is to propose an implicitly parallel environment that synchronizes concurrent memory accesses using a software transactional memory. One of the novel ideas compared to similar approaches is that our software transactional memory is used transparently (i.e., with no interference of programmers) and is fully compatible with the idea of our implicit parallel execution model. Since our language allows for mutations, it is generally a difficult task to have such a parallel environment because we have to take into account that any object may be subject of mutation. On the other hand, we exploit the fact that our language is mostly functional and mutations are used rarely and responsibly.

In this paper, we first outline the evaluation model in its sequential and parallel variants. Then, we explain the structure of our transactional memory and the related operations. The paper concludes with experiments using our implicitly parallel interpreter with software transactional memory. The rest of this section recalls basic notions related to the software transactional memory.

1.1 Software Transactional Memory

Software transactional memory has its origin in relational database systems. Despite the fact that a parallelism and a related concurrency usually cause difficulties, database systems deal with these issues with no extra effort. The central concept that allows this is a *transaction*. From the user's point of view, a database transaction represents a powerful feature enabling large number of concurrent queries and updates in data tables. Moreover, transactions provide simple and comprehensible user interface. One can see that data tables used in databases are indeed very similar to main memory used in computers. This is the main idea behind the software transactional memory.

Software transactional memory allows splitting of programs into logical blocks (transactions) and execution of these block that fulfills criteria of *atomicity, consistency and isolation*. This means, all operations in a transaction are processed entirely or not at all, the system is going from one consistent state to another one, and processing of a transaction does not affect other running transactions.

For more information on transactional memory see [15].

2. SCHEMIK: AN IMPLICITLY PARALLEL LANGUAGE

In this section we provide a brief overview of our language called Schemik. Schemik is a programming language highly inspired by Scheme (R5RS) and Common LISP. We focus on the sequential and parallel models of program evaluation. In further sections, we describe how the evaluation model can be equipped with STM. Detailed description of Schemik and its evaluation model is introduced in [13]. In the sequel we recall only necessary notions.

We assume that readers are acquainted with some dialect of LISP or Scheme. In the sequel, we tacitly use the usual terminology which can be found in [1, 10, 11]. Just to recall, programs and all expressions in Schemik are represented as symbolic expressions. For instance, an expression "$1+2\times3$" is represented in Schemik as a list `(+ 1 (* 2 3))`. All symbolic expressions which are intended for evaluation are written in the prefix form, i.e., the first place of a list (intended for evaluation) is occupied by an object that evaluates to a function or a special operator and the remaining objects in the list are used for arguments.

2.1 Sequential Evaluation

The main idea of our evaluation model consists in decomposing the evaluation process into several comprehensible steps such as initiation of the evaluation, argument evaluation, and function application. In order to capture such steps, we propose a stack-based evaluator that is a deterministic pushdown automaton with two stacks: *execution stack* (denoted E) containing *stack operations* controlling the evaluation, and *result stack* (denoted R) containing Schemik objects playing the roles of operands and intermediate results of operations.

The input (first expression to be evaluated) is encoded on the execution stack and the output (result of evaluation) is pushed on the result stack at the end of the computation. The content of both stacks is what completely describes the current state of the evaluator. Therefore, a setting of the stacks shall be called a *configuration* of the evaluator. The result stack contains first-class objects of Schemik in their internal representation which are basically the same objects as in Scheme [11] (i.e., numbers, symbols, functions, special operators, etc). The execution stack can be seen as a stack for pending operations. Unlike the result stack, it contains stack operations which are not Schemik objects.

The stack operations are represented by tuples of the form $\langle\textsf{operation-name}, arg, \mathcal{E}, \textsf{flag}\rangle$ where operation-name is a name for a step of evaluation; in the sequel we will use the following names: DEFINE, DISCARD, DROP, EVAL, FEVAL, FUNCALL, IF, INSPECT, RETURN, SET, and WAIT; *arg* is an object representing the argument for stack operation; $\mathcal{E}$ is an environment (i.e., a table describing bindings of lexical variables) associated to the stack operation; flag is an indicator of tail-recursion optimization [11].

During the computation, the automaton changes its configuration. A change from one configuration to another will be called a *transition*. Each transition is determined by the stack operation which resides on the top of the execution stack. The computation halts if the execution stack is empty and the object on the top of the result stack is the *result* of the computation. Each transition of the automaton may be depicted by two pairs of stacks—configuration *before* and *after* the transition—and it may be written as follows:

E: *stack with operations before transition*$\rrbracket$
R: *stack with results before transition*$\rrbracket$

E: *stack with operations after transition*$\rrbracket$
R: *stack with results after transition*$\rrbracket$

The *start configuration* of the automaton contains a single stack operation $\langle\textsf{EVAL}, expr, \mathcal{E}_t, \textsf{N}\rangle$ on the execution stack and an empty result stack, meaning that symbolic expression *expr* will be evaluated in the top-level environment $\mathcal{E}_t$ (environment containing initial bindings of lexical variables), N says that *expr* does not appear in a tail position, see [11].

The operation $\langle\textsf{EVAL}, object, \mathcal{E}, \textsf{f}\rangle$ initiates evaluation of *object* (e.g., a list representing a symbolic expression) in environment $\mathcal{E}$. If this operation is on the top of the execution stack, an appropriate transition is made depending on the type of *object*. Three situations may occur: (i) *object* is a self-evaluating object (i.e., neither a symbol nor a non-empty list), in which case *object* is pushed on the result stack; (ii) *object* is a symbol (name of a lexical variable), its value in environment $\mathcal{E}$ is pushed on the result stack; (iii) *object* is a list `(`$head_\sqcup arg_1{}_\sqcup\cdots{}_\sqcup arg_n$`)` then the top of the execution stack is replaced as follows:

E: $\langle\textsf{EVAL}, \texttt{(}head_\sqcup arg_1{}_\sqcup\cdots{}_\sqcup arg_n\texttt{)}, \mathcal{E}, \textsf{f}\rangle\cdots\rrbracket$
R: $\cdots\rrbracket$

E: $\langle\textsf{EVAL}, head, \mathcal{E}, \textsf{N}\rangle, \langle\textsf{INSPECT}, \texttt{(}arg_1{}_\sqcup\cdots{}_\sqcup arg_n\texttt{)}, \mathcal{E}, \textsf{f}\rangle\cdots\rrbracket$
R: $\cdots\rrbracket$

The previous transition has prepared the *head* of the original list for evaluation. The role of INSPECT is to distinguish between different evaluation rules based on the value of the *head*, i.e. the first element in the list. The *head* may evaluate to a function, a special operator, or a macro. For instance, if INSPECT is on the top of the execution stack and the top of the result stack contains a *function*, INSPECT will prepare application of the function, i.e., it will prepare all arguments for evaluation and then it prepares a function call using FUNCALL:

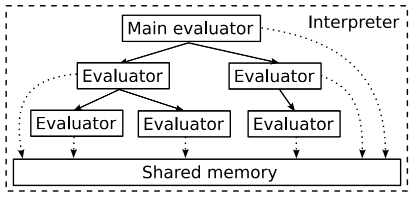

Figure 1: Organization of evaluators

E: $\langle \mathsf{INSPECT}, (arg_1 \sqcup \cdots \sqcup arg_n), \mathcal{E}, \mathsf{f}\rangle \cdots]\!]$
R: $\mathit{function} \cdots]\!]$

E: $\langle \mathsf{EVAL}, arg_1, \mathcal{E}, \mathsf{N}\rangle, \ldots, \langle \mathsf{EVAL}, arg_n, \mathcal{E}, \mathsf{N}\rangle,$
$\langle \mathsf{FUNCALL}, n, \mathcal{E}, \mathsf{f}\rangle \cdots]\!]$
R: $\mathit{function} \cdots]\!]$

The application of functions is performed by FUNCALL which is used in the form $\langle \mathsf{FUNCALL}, n, \mathcal{E}, \mathsf{f}\rangle$, where n is the number of arguments. Arguments and the function itself are stored on the result stack. In general, $\langle \mathsf{FUNCALL}, n, \mathcal{E}, \mathsf{f}\rangle$ takes $n+1$ objects from the result stack where first n objects represent arguments and the last object taken is the desired function. If the function is a primitive function (i.e., a built-in function), the arguments are passed to the function and the result is pushed on the result stack. Analogously, INSPECT handles user-defined functions, macros, and special operators. Further details are outside the scope of this paper and are available in [13].

2.2 Parallel Evaluation Model

In this subsection, we introduce our implicit parallel evaluation model. We begin with a parallel evaluator of a purely functional subset of Schemik, and later, we extend the language into its full extent.

2.2.1 Purely Functional Parallel Evaluation

Consider a *purely functional* subset of Schemik, i.e., the language without assignment operations and mutators of pairs (`set!`, `set-car!`, `set-cdr!`), I/O functions (`display`, etc.), and explicit continuations (`call/cc`).

Our parallel evaluator of purely functional subset of the language is based on the idea that each operation EVAL may be processed in an independent evaluator as far as the resulting value is dependent only on given arguments and environment. We introduce a new stack operation FEVAL (abbreviation for "fork eval") representing evaluation in an *independent evaluator*, i.e., a thread. Each independent evaluator has its own independent pair of stacks and environments are shared among all independent evaluators. We can share environments without any inconsistency issues because we are considering programs without side effects only. This means, once an environment is established, it does not change bindings of symbols (lexical variables). During the computation, all the independent evaluators form a tree structure with a single (main) evaluator always at the top of the hierarchy, as depicted in Figure 1.

2.2.2 Stack Operation FEVAL

Operation FEVAL is similar to EVAL; it prepares an expression for evaluation in an independent evaluator, i.e., using a new pair of stacks. Operation FEVAL is specific in that it is not a result of any transition caused by another operation. The appearance of FEVAL on any execution stack is caused by external entity called *scheduler* which acts as

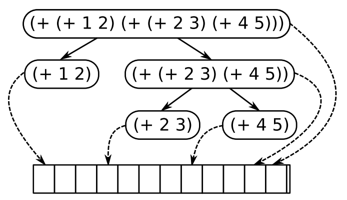

Figure 2: Approximate location of operations on the execution stack

deus ex machina and converts a suitable EVAL operation on execution stacks to the FEVAL operation and creates a new evaluator containing the original EVAL on top of its execution stack. This way a parallel branch of the evaluation is created. The transition created by the scheduler may be depicted as follows:

EV_1: E: $\cdots \langle \mathsf{EVAL}, \mathit{object}, \mathcal{E}, \mathsf{f}\rangle \cdots]\!]$
R: $\cdots]\!]$

$\vdots$

EV_1: E: $\cdots \langle \mathsf{FEVAL}, \langle EV_2, \mathit{object}\rangle, \mathcal{E}, \mathsf{f}\rangle \cdots]\!]$
R: $\cdots]\!]$

EV_2: E: $\langle \mathsf{EVAL}, \mathit{object}, \mathcal{E}, \mathsf{N}\rangle]\!]$
R: $]\!]$

An invocation of FEVAL represents merging of two branches of evaluation. If $\langle \mathsf{FEVAL}, \langle EV_i, \mathit{object}\rangle, \mathcal{E}, \mathsf{f}\rangle$ appears on the top of the execution stack, the evaluation in the referred evaluator EV_i is stopped and its stacks are appended to the corresponding stacks of the current evaluator processing the FEVAL operation.

This behavior of FEVAL is correct because the referenced evaluator EV_i has performed exactly the same transitions that should be otherwise performed by the evaluator containing the FEVAL operation. We can consider the creation of this operation as a request for parallel evaluation to other evaluator and processing of FEVAL operation as a request for obtaining (partial) results from the referenced evaluator.

2.2.3 Scheduling Strategy

An important issue in implicit parallel evaluation is how to decide which expression is worth evaluating in a parallel branch. In theory, an arbitrary expression may be parallelized but it is desirable to focus on particular expressions that require larger amounts of transitions.

The decision on which expression will be evaluated in a parallel branch is done by the *scheduler*, associated to each evaluator, regularly seeking through the execution stack for operations EVAL that may be converted into the FEVAL operations. A stack operation $\langle \mathsf{EVAL}, \mathit{object}, \mathcal{E}, \mathsf{f}\rangle$ is considered as a good candidate for parallel evaluation if *object* is a list fulfilling further criteria. For instance, *object* should be a list containing other list, or a list with a user-defined function as its first element.

Our scheduling strategy has its origin in the organization of the execution stack and in the assumption that each program consists of complex expressions decomposable to less complex expressions also decomposable to less complex expressions, etc. Therefore, our strategy anticipates that more complex expressions suitable for parallelization are at the bottom of the execution stack, see Figure 2.

Moreover, scheduler also collects information on each expression that has been evaluated in parallel. These information include a number of parallel evaluations and a number of parallel evaluations that were not successful. If the number of unsuccessful parallel evaluations exceeds threshold, scheduler no longer considers this expression for parallel evaluation.

Further details on scheduler and scheduling strategy are out of scope of this paper and are not necessary for understanding of this paper. For more information, see [13].

2.2.4 Side Effects

We now turn our attention to the full language including operations with side effects: assignment, I/O operations, and continuations. Their processing in parallel is generally a non-trivial task because of synchronization issues. Other parallel languages solve analogous issues in various ways, e.g., [4] forbids explicit mutations, [17] incorporates new synchronization primitives, etc. Schemik solves inconsistency issues by transactional memory and by proper ordering of critical operations.

As far as the purely functional subset of Schemik is considered, the order in which subexpressions are evaluated is generaly not important. Any order produces the same results. This rule does not apply for the full language. For instance,

```
(let ((foo (lambda (x) (display x) x)))
  (+ (foo 1) (+ 2 3) (foo 2) (- 10) (foo 3))) ⟹ 1
```

always evaluates to `1` but can produce different outputs displayed on the screen (`123`, `132`, `213`, `231`, `312`, and `321`), depending on the order in which `foo`'s are called. Therefore, in order to ensure unique results, we define a specific order in which all operations with side effects must be performed. We shall use the "left-to-right order" which postulates that any operation with side effects to the left must be performed before all operations with side effects to the right. In case of the previous example, the correct output will be `123`. The expressions without side effect may be evaluated in any order.

We have to be aware of the fact that we have actually two kinds of operations with side effect—*reversible* (operations that can be undone—memory updates) and *irreversible* (operations that cannot be taken back—I/O, calling an escape function, etc.) Handling of reversible operations (`set!`, `set-car!`, etc.) is done through the transactional memory that is described in the next section and irreversible operations are properly ordered to produce sound results.

The proper ordering of *irreversible* operations is ensured in our parallel model by a rule saying that *only the main evaluator can perform operations with side effects.* In other words, the main evaluator (thread) which resides on the top of the hierarchy of evaluators, see Figure 1, represents the main sequence of operations in a program and all other evaluators (threads) are helpers with limited capabilities.

Despite the fact that particular operations are restricted on main thread and have to be properly ordered, this does not bring significant performance loss since the computation continues in all other evaluators without any influences.

3. TRANSACTIONAL MEMORY

We now focus on particular operations having side effects that may be easily undone. This covers especially a group of operations imperatively changing objects—lexical environments, conses, vectors, etc. This kind of operations is not very typical for languages such as Schemik or Scheme since the languages have roots in functional programming. Despite this fact, these operations play an important and irreplaceable role. In the sequel, we describe our apparatus and methods of handling transactional memory in Schemik.

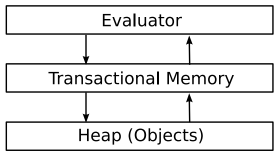

Figure 3: Layer organization of Schemik

3.1 Overview

Schemik's use of transactional memory has some specific properties. Especially, we do not incorporate transactions as a language construct but rather as means of the runtime environment to run each computation in a consistent memory. This makes the situation complicated since we do not have any information on objects which are to be updated and we have to consider that any object may be subject of modification. This is in contrast from, e.g., Concurrent Haskell [7] that requires all transactional objects to be explicitly declared.

Each evaluator in our model has its own transaction providing access to the objects in the memory and defining whether the computation has been correct or not. That is, if a transaction successfully commits, the underlying computation is correct and the result may be used. Otherwise the computation must be retried. All the transactions make a hierarchy of nested transactions that directly copies the hierarchy of evaluators. Despite the fact there is a one-to-one relationship between evaluators and their transactions, these concepts are indeed strictly isolated—transactional memory only encapsulates an access to objects stored in the memory and provides an abstraction barrier, as depicted in Figure 3.

The most important property of Schemik is its ability to run programs in parallel without any impact on the result. This property is reflected even in the memory management and, as a consequence of this, the transactions have to be committed or aborted in their logical order instead of the time they were started or the point where the transactions actually finished their jobs. Consider the following code with each of the assignments running in its own transaction:

```
(let ((foo 42))
  (set! foo (* foo 2))  ; #1
  (set! foo (+ foo 1))) ; #2
```

If we demand always the same results as in case of sequential evaluation, transaction `#1` has to be committed before `#2` no matter which one was created or completed first. This implies that transactions have to be committed in some logical order, as in our case in the left-to-right (top-down) order.

Furthermore, this logical organization of commits guarantees that each transaction successfully commits after a finite number of attempts (if the whole program terminates after a finite number of steps) because the first transaction always commits. Another consequence of keeping the required order is that we do not need a *contention manager* and all necessary decision are given during the evaluation.

Schemik tacitly assumes a functional programming style with rare and responsible use of mutations. Purely functional programming is well-suited for implicit parallelism and can be applied on wide range of problems but there are several situations where application of imperative aspects is necessary.

In conjunction with the anticipated functional programming style, we suppose that each evaluator (transaction) will create a considerable amount of intermediate or auxiliary objects (e.g., environments). It may be simply proved that these objects are accessible only from the current transaction or from its nested transactions, respectively. This, with the anticipated order of commits, makes it possible to implement efficient direct updates because only the transaction that has created these object can access them. A similar idea is described in [16].

3.2 Operations

To recall, our transactions form a hierarchy and each evaluator has associated its own transaction. Each reading of an object from the memory or its modification is performed in a uniform way through the associated transaction. This allows us to consistently deal with objects defined outside of the current transaction and avoid unwanted situations.

Each evaluator performs a sequence of commands in given order and, alternatively, it may create parallel branches of evaluation that can be seen as requests to "helpers" for computation of particular subexpressions. We equip each such a request with its own *image of memory* and, consequently, we can ensure that such computation is done in a consistent memory without influences from the outside. This allows us to separate memory updates into logical blocks and recognize and eventually avoid concurrent access to objects.

3.2.1 Structure of Transactions and Mutable Objects

Before we proceed to the description of transactions and operations on them, we outline the structure of our mutable objects—environments, conses, vectors, etc. Every object has an integer counter v representing the version of the object. This counter is increased each time the object is updated. Besides this, every object contains a pointer *owner* to the transaction that has created this object. In other words, the transaction that "owns" this object. If the transaction commits, the ownership of all its objects is taken over by the parent transaction.

Each transaction consists of three logs:

- *read log*—contains records on all objects that were read and that might be possibly updated from the superior transaction and therefore there is a risk of read-write collisions; these records are in the form of $\langle o, v\rangle$, o stands for the object that was read and v identifies its current version.
- *write log*—consists of objects representing deferred updates that were not applied directly into the objects; each log item is a tuple of the form $\langle o, m, cv, v, owner\rangle$, where o stands for the reference to the updated object, m is a mutation updating object. cv stands for "commit version" and it is set only if the mutation was applied to the object and contains version v of the object after this update. v stands for the version of the log record itself and *owner* is a reference to the transaction that has created this record.
- *parent write log*—represents the state of the *write log* and the *parent write log* from the superior transaction in the beginning of the current transaction, i.e., it provides an image of the memory with deferred updates from the parent transaction; all records are pointers to the records stored in the logs of the parent transaction.

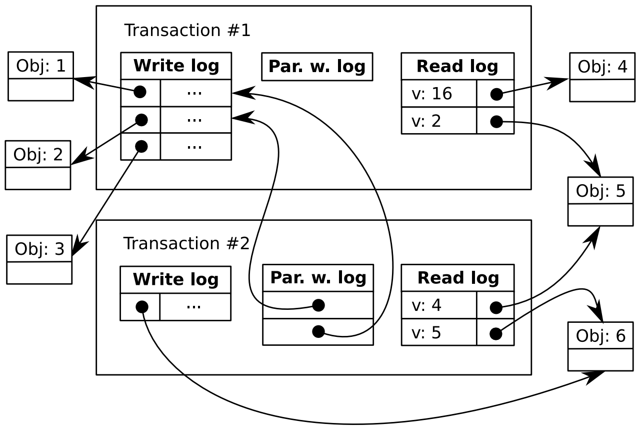

Figure 4: Example of a structure of two nested transactions

An example of a structure of two nested transactions and their relationship can be found in Figure 4.

We have to emphasize that records in the *write log* and *parent write log* are objects in the exactly same sense as regular Schemik objects and their readings and updates have to be handled in the same way as in case of the other objects.

The write log contains a description of the mutation only. This is in contrast with STMs proposed, for example, by [3, 5, 8] that stores a copy of the updated object. It has several practical reasons. Since Schemik assumes rare use of object updates, this way of storing updates is more space efficient than copying of complete objects. Furthermore, this approach fits better to our evaluation model. For instance, let us consider an environment $\mathcal{E}$ containing value 10 bound to the symbol `foo`, i.e., pair $\langle \texttt{foo}, 10\rangle$ and record in the *write log* updating the value of `foo` to 20. Such write log record looks like $\langle \mathcal{E}, \langle \texttt{foo}, 20\rangle, \ldots\rangle$. While evaluating symbol `foo` in the environment $\mathcal{E}$, we can easily check whether this value has been changed or not, and return a new value or a value from the object, respectively.

3.2.2 Initiation of a New Transaction

A new transaction is started each time the *scheduler* decides to create a new evaluator. The new transaction contains empty *read log*, *write log*, and the *parent write log* that is filled with the content of the parent's *write log* and parent's *parent write log*, i.e., creates an image of the current transaction's memory. Records in the *parent write log* are not directly copied but rather referenced.

3.2.3 Access to the Objects in the Memory

Reading of an object from the memory is done in two steps. (1) The evaluator checks *write log* and *parent write log* (in this order) for updates, if an update exists in the

```
1: (let ((foo 10))  ; #1
2:   (set! foo 20)  ; #1
3:   (bar foo))     ; #2
```

Figure 5: Example of a read-write collision (a)

log, it uses the state of the object from the log; otherwise it returns the current state of the object. (2) If the object is at the risk of a read-write collision (i.e., update from another transaction), this is if and only if the object is not owned by the current transaction, it is necessary to make a record into the *read log*. A record consisting of the pointer to the object that was in fact read and its version is stored into the *read log*. For example, if the object was read from the *write log*, pointer to the log record is stored. While updating objects, three specific situations may occur:

(1) the transaction owns a given object and can directly update the object—direct update is performed and the version of the objects is increased by one,

(2) the object was already updated in this transaction—log record is updated and the version of this record is increased by one,

(3) otherwise the update is deferred and recorded into the write log.

3.2.4 Transaction Commit

An attempt to commit a transaction is performed only if the evaluator performs operation FEVAL. If we consider the creation of FEVAL and a new sub-thread as a *request for computation*, this may be seen as a *request for the result value*—parent thread takes over results from the sub-thread. To ensure that these results are sound, it is desired to check whether the state of the parent transaction has not changed in a way that might possibly affect result from the nested transaction. For this reason, *write logs* and *read logs* of committing and committed transactions are checked. A sub-transaction has to be aborted, if there is a read-write collision. This means, if one of the following situations has occurred:

(a) An object whose version has changed since it has been read is recorded in the sub-transaction's *read log*.

(b) The sub-transaction's *read log* contains a record with object that is recorded in the *write log* of the superior-transaction.

(c) The sub-transaction's *read log* contains a reference to a *write log* record that has already been applied to an object (during the commit) and the current version of this object is higher than the version at the time of the commit.

The situation (a) occurs, for instance, in the code in Figure 5. If the local environment with the lexical variable `foo` (modified object) is owned by transaction `#1` (line 1) and function call `(bar foo)` at line 3 is evaluated in a nested transaction `#2` which has started before the modification of the value `foo` at line 2. Since the transaction `#1` owns the local environment, it directly updates `foo` to its new value (line 2), but the transaction `#2` has read the old value. Collision is evidently resolved by comparing versions of the object at the time of reading and at the time of committing.

The situation (b) occurs under very similar circumstances as the previous one. The difference is that none of the colliding transactions owns a modified object. Let us consider the code from Figure 6 and the following conditions. Transaction `#1` creates a local environment with the lexical variable `foo` (line 1), transaction `#2` starts as its nested transaction (line 2). Function call `(bar foo)` is evaluated as a new transaction `#3`, a nested transaction of the transaction `#2`, before the modification of the value `foo` at line 3. In this case, the modification of the object (at line 3) is not applied to the object itself because the transaction `#2` does not own it. The update is deferred and recorded into the write log of the transaction `#2`. If the transaction `#3` attempts to commit, from its read log and the write log of the superior transaction it can be easily resolved that there is a read-write collision.

```
1: (let ((foo 10))    ; #1
2:   (let ()          ; #2
3:     (set! foo 20)  ; #2
4:     (bar foo))     ; #3
```

Figure 6: Example of a read-write collision (b)

The most complex colliding situation is represented by case (c). Let us consider that the code from Figure 7 is executed as follows. Transaction `#1` creates a local environment with the lexical variable `foo` (line 1) as in the previous case. Transaction `#2` starts as its nested transaction (line 2) and modifies value of `foo` at line 3. The update is deferred into the transaction `#2`'s write log. Afterwards, transaction `#3` is started, evaluating function call `(bar foo)`. Since the value of `foo` was modified by the transaction `#2` and the transaction `#3` has a record on update of `foo` in its *parent write log*, transaction `#3` reads the value of `foo` from its *parent write log* and this reading is recorded in the transaction `#3`'s read log. If the transaction `#2` is committed, then the object `foo` is updated to the value from the transaction `#2`'s write log and the new version number of the object is set in the write log record in the *cv* field. Now, we should emphasize that in the point where the transaction `#2` has been committed, it was not processed entirely and there are still pending operations (line 4 and 5). In case that the transaction `#1` performs the update of `foo` at line 4 and if the transaction `#3` attempts to commit after that, it will cause a read-write collision. To resolve such collisions, all entries in the *read log* of committed transaction that are pointing to the *write log* records are checked. If the *write log* record indicates (in the field *cv*) that the update was applied to the object (during the commit) and if the current version v of the object differs from the value stored in *cv*, the object has been updated and a read-write collision has occurred.

```
1: (let ((foo 10))   ; #1
2:   (let ()         ; #2
3:     (set! foo 20) ; #2
4:     (set! foo 30) ; #1
5:     (bar foo)))   ; #3
```

Figure 7: Example of a read-write collision (c)

If there is no collision, all entries from the sub-transaction's *read log* are recorded to the current transaction using the rules for reading objects from the previous section, i.e., if the nested transaction has read an object owned by the superior transaction, this record is not included in the resulting *read log*. Records in the *write logs* are also merged in a similar way. If the nested transaction has modified an object owned by the parent transaction, this record is also not included in the resulting *write log* but the direct update is performed instead and a new value of v is stored in the *cv* field of the record. This is necessary since the record may be referred from some nested transaction's *parent write log*.

As mentioned before, a transaction commits only if the operation FEVAL is processed. This allows us to preserve logical order of transactions in a similar way as in case of ordering operations with irreversible side effects.

Furthermore, this guarantees that each transaction successfully commits. First transaction has to be always successfully committed since there are only two options: (1) transaction is successfully committed. (2) Transaction is aborted and retried; then it is successfully committed because there is no other operation that may affect the computation in such a transaction as one can see from the following configuration:

E: $\langle \mathsf{FEVAL}, \langle EV_n, o_1\rangle, \mathcal{E}, \mathsf{f}\rangle, \cdots, \langle \mathsf{FEVAL}, \langle EV_{n+1}, o_2\rangle, \mathcal{E}, \mathsf{f}\rangle, \cdots \rrbracket$
R: $\cdots \rrbracket$

Remarks: (i) It may happen that an evaluator commits a transaction that has computed results only partially. At first glance this may seem to be violating the property of atomicity of a transaction (i.e., all-or-nothing proposition). But in our case, due to given order of commits, we may decide whether the transaction will be successfully committed or not using only records that are available. With this decision we have two options—abort transaction and retry the computation or proceed through the transaction.
(ii) The top level transaction has no parent transaction and it is not necessary to collect information on reads or writes. Due to this, transactional memory has no performance impact on a single-thread evaluation.
(iii) If the transaction has no nested transaction, there is no need to increase object versions during each update. This allows us to decrease potential risk of integer overflow in version number. Moreover, if there is no sub-transaction, it is even possible to reset the version number back to the initial value. This is sufficient to eliminate unwanted version number overflows—if the version number reaches the limit, it is sound to abort all nested transactions and reset the version back to an initial value.

4. EXPERIMENTS

To take observations on efficiency, we have implemented our software transactional memory model in our implicitly parallel interpreter of Schemik. The implementation of STM is a thin layer between the evaluator and objects on the garbage collected heap. For performance reasons, STM is not using any kind of locks and employs only atomic operations commonly available on contemporary computers.

For our experiments we have selected five programs using explicit mutations in different ways. Programs sorting data using quicksort algorithm, computing combinations with and without replacement are using destructive concatenation of lists. A program computing value of number π by *monte carlo method* is imperatively changing internal state of the lexical environment to generate pseudo-random numbers. And the last example modeling transfers between bank accounts is imperatively changing state of the lexical environments representing bank accounts in the very similar way as in the previous case, but due to nature of this example, there is a large number of collisions.

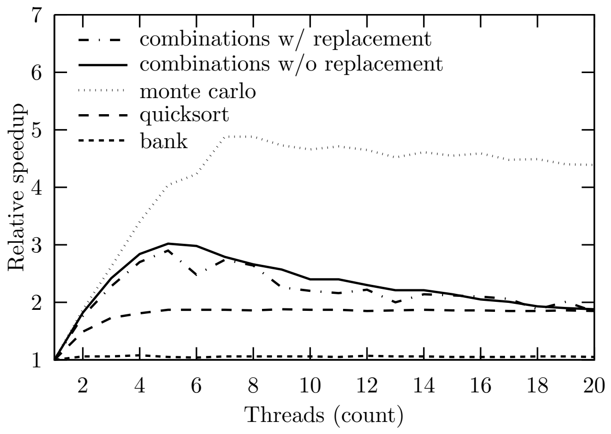

Figure 8: Scalability of various programs

	Schemik (1 thr.)	Schemik (8 thr.)	Guile	MzScheme
bank	2.72	2.57	0.97	1.75
combi. 1	20.08	8.19	10.19	24.92
combi. 2	6.24	2.33	3.24	7.93
monte-carlo	11.39	2.31	12.06	55.91
quicksort	7.68	4.13	3.38	6.91

Table 1: Run-time of benchmarks involving explicit object updates (in seconds)

The absolute running times of benchmarks of single-threaded and multi-threaded evaluation are presented in Table 1. For comparison, Table 1 also contains benchmark results of long-time developed production quality interpreters of Scheme. As one can see, Schemik provides competitive results to these interpreters.

While evaluating parallel programs, besides the overall performance, we have to take into account an important feature called scalability. In the second set of experiments, we focus on this feature. To represent scalability, we are using *relative speedup*—a ratio $S = \frac{T_1}{T_n}$, where T_1 is the running time of program using only one evaluator and T_n is the time running time of the program using up to n evaluators.

Figure 8 depicts relative speedups of programs we have considered. One can see that, with the exception of the bank example, all programs show significant improvement in speed while running in parallel. The common thing for these programs is a "very responsible" use of mutations. On the other hand, example with bank accounts is using mutations very extensively, therefore the improvement in speed is not so significant.

For the sake of completeness, we have to mention, that for all experiments we had used computer equipped with eight-core UltraSPARC T1 processor and 8 GB RAM.

5. RELATED WORKS

In the last decade, transactional memory gained wide popularity and became into the focus of many researchers. Wide overview of recent research provides [15]. Generally, three main streams of transactional memory evolved—*software transactional memory* (STM), *hardware transactional memory* (HTM) and combination of the previous two as *hybrid transactional memory* (HyTM).

The common practice is to equip a well-established language (Java, C#, Haskell, etc.) with libraries or language constructs enabling transactional handling of objects. This approach is the major one and the commonly used methods are described for example in [2, 3, 5, 7, 6, 8, 18]. From our point of view, this approach has significant disadvantage since it requires additional explicit modifications of programs to exploit parallelism.

Several LISP systems were proposed to automatically parallelize running programs. One of these is Parcel [9], compiling Scheme source code into the intermediate code and consequently parallelizing this intermediate code. This system is not using any kind of transactional memory. Another system is described in [12], this system speculatively decompose programs into the blocks (transactions) and memory collisions are detected by cache in multiprocessor system, i.e., use hardware transactional memory.

We have already proposed a variant of implicitly parallel dialect of Scheme in [14, 13], but these papers focus mostly on the implicit parallel evaluation and the issues of side effects are discussed only partially. The formal model in these papers deals with side effects only from the point of view of correctness of results and the performance issues are not considered.

6. CONCLUSIONS

Software transactional memory seems to be an elegant and efficient way of handling concurrent memory access into objects. We propose a variant of software transactional memory suitable for use in implicitly parallel functional languages allowing us to use imperative object updates. This method of handling memory access seems to be an appropriate mean that enhances implicit parallel evaluation with fine-grained collision detection of such updates.

The future research will focus on advanced features of implicit parallel evaluation and STM including performance improvements. Furthermore, we would like to use our model to create efficient implicit parallel interpreters of other languages including JavaScript or Python.

7. ACKNOWLEDGMENTS

Supported by research plan MSM 6198959214.

8. REFERENCES

[1] John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., New York, NY, USA, 1978.

[2] C. Scott Ananian and Martin Rinard. Efficient object-based software transactions. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*, San Diego, CA, Oct 2005.

[3] Keir Fraser. Practical lock-freedom. Technical report UCAM-CL-TR-579, University of Cambridge, Computer Laboratory, February 2004.

[4] Guillaume Germain. Concurrency oriented programming in termite scheme. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 20–20, New York, NY, USA, 2006. ACM.

[5] Guerraoui, Herlihy, and Pochon. Polymorphic contention management. In *DISC: International Symposium on Distributed Computing*. LNCS, 2005.

[6] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 388–402, New York, NY, USA, 2003. ACM.

[7] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (10th PPOPP'2005), ACM SIGPLAN Notices*, pages 48–60, Chicago, IL, USA, June 2005.

[8] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[9] Williams Ludwell Harrison III. The interprocedural analysis and automatic parallelization of scheme programs. *Lisp and Symbolic Computation*, 2(2):179–396, 1989.

[10] Guy L. Steele Jr. *Common LISP the Language*. Digital Press, 2nd edition, 1990.

[11] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[12] Thomas F. Knight. An architecture for mostly functional languages. In *LISP and Functional Programming*, pages 105–112, 1986.

[13] Petr Krajca and Vilem Vychodil. Stack-Based Model of Implicit Parallel Execution of Functional Programs. Submitted.

[14] Petr Krajca and Vilem Vychodil. Data parallel dialect of scheme: outline of the formal model, implementation, performance. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 1938–1939. ACM, 2009.

[15] James Larus and Ravi Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.

[16] Yossi Lev and Jan-Willem Maessen. Toward a safer interaction with transactional memory by tracking object visibility. In *Proceedings, Workshop on Synchronization and Concurrency in Object-Oriented Languages*. San Diego, CA, October 2005.

[17] Toshihiro Matsui and Satoshi Sekiguchi. Multithread Implementation of an Object-Oriented Lisp, EusLisp. In *Advanced LISP Technology (Advanced Information Processing Technology)*, pages 59–80. Taylor & Francis, June 2002.

[18] Filip Pizlo, Marek Prochazka, Suresh Jagannathan, and Jan Vitek. Transactional lock-free objects for real-time Java. In *Proceedings of the PODC Workshop on Concurrency and Synchronization in Java Programs*, pages 54–62, 2004.