UDC 004.652

Developing Libraries Using Software Transactional Memory

Ricardo Dias[1], João Lourenço[1] and Gonçalo Cunha[1]

[1]CITI–Centre for Informatics and Information Technology, and
Departamento de Informática, Universidade Nova de Lisboa
Portugal
{rjfd, joao.lourenco}@di.fct.unl.pt
goncalo.cunha@gmail.com

Abstract. Software transactional memory is a promising programming model that adapts many concepts borrowed from the databases world to control concurrent accesses to main memory (RAM). This paper discusses how to support revertible operations, such as memory allocation and release, within software libraries that will be used in software memory transactional contexts. The proposal is based in the extension of the transaction life cycle state diagram with new states associated to the execution of user-defined handlers. The proposed approach is evaluated in terms of functionality and performance by way of a use case study and performance tests. Results demonstrate that the proposal and its current implementation are flexible, generic and efficient.

Keywords: Transactions; Software Transactional Memory; Compensation Actions; Revertible Operations.

1. Introduction

The current trend of having multiple cores in a single CPU chip is leading to a situation that many would believe absurd not long ago: one may have more computational processing power than can (easily) be used. To invert such a situation, there is the need to find and explore concurrency where before sequential code would be written. An appealing approach towards such a goal is the transactional programming model, which makes use of high-level constructs to deal with concurrency and is becoming a potential alternative to the classical lock based constructs such as mutexes and semaphores.

Since the introduction of Software Transactional Memory (STM) [1], this topic has received a strong interest by the scientific community. Opposed to the pessimistic approach used in lock-based constructs, transactional memory may use optimistic methods for concurrency control, stimulating and enhancing concurrency. The need for performance has already made some authors propose to drop some interesting features. Early STM frameworks [2, 3] were implemented using non-blocking synchronization [4] and, as a consequence, they supported arbitrary thread failures. Tests with lock based STM frame-

works [5, 6] suggest that this approach exhibits better performance and should be considered as an implementation alternative.

Until now, most STM implementations reside mainly in software libraries (e.g., for C or Java programs) with minimal or no changes at all to the syntax and semantics of the programming language, therefore relying in the programmer to do the transactional memory accesses by way of a library API. Ongoing research is using Concurrent Haskell and Java as a test bed for runtime and compiler changes to support STM [2, 7]. Many problems and difficulties still persist in using the STM programming model, supported only by software libraries [8] or directly by the compiler [9].

This paper discusses some of the problems related to using external libraries in a transactional context. The case study arose when developing small applications examples for the CTL [10] transactional memory library. CTL is a library-based STM implementation for the C programming language, derived from the TL2 [5] library. CTL extends the TL2 framework with a large set of new features and optimizations, and also solves some bugs identified in the original framework [11].

The remaining of this paper is organized as follows: Sect. 2 describes the motivation and context for this work; Sect. 3 proposes an approach to overcome the identified difficulties; Sect. 4 describes its implementation; Sect. 5 introduces some use case examples; Sect. 6 evaluates the overhead introduced by the current implementation; Sect. 7 discusses the related work; and Sect. 8 presents some concluding remarks, and some current and future research work within this topic.

2. Motivation

Programming using the STM model is straight forward: if one plans to do a set of memory operations atomically—do all or none of the operations—and/or want to access memory locations in isolation—the memory accesses will not interfere with concurrent accesses from other control flows (threads)—, such set of memory operations should be enclosed within a memory transaction.

```
atomic{
   operation_1;
   operation_2;
   ...
   operation_n;
}
```

```
start_T();
   operation_1;
   operation_2;
   ...
   operation_n;
end_T();
```

Fig. 1. Transactional code blocks supported by the programming language (on the left) or by a software library (on the right).

If the STM programming model is directly supported by programming language constructs, a transactional code block may look as illustrated in Fig. 1 on the left; while if the STM programming model is supported only by a software library, the same code block may look as illustrated in Fig. 1 on the right. Although syntactically different, both code blocks share a common semantic.

The transactional model assumes that all the operations executed within a non-committed transaction can be undone. This is not always the case for memory transactions. The operations executed within a memory transaction will fall into one of two categories:

- *Non-transactional operations*. These operations are irreversible and cannot be undone. Examples include output to the terminal that has been read by the final user;
- *Transactional operations*. These operations can be undone and are safe to be used within memory transactions. Transactional operations may be subclassified into one of three categories, according to how the transactional recovery is done:
  - *Pure transactional operations*. Can be automatically reverted by the transactional memory run-time whenever necessary. Examples include read and write memory accesses;
  - *Revertible operations*. Can be reverted or deferred by the transactional memory framework with the help of extra coding from a developer with some domain knowledge. Examples include explicit memory management operations (memory allocation and release);
  - *Compensable operations*. Can be compensated with another operation in another transaction that produces an opposed effect. Examples include truncating a file that was appended by a previously committed transaction. Compensable operations that are observed by entities external to the system (e.g., users) become non-transactional.

When developing a software library, the library programmer (programmer from hereinafter) aims at creating a black box behind a well-defined interface (API), hiding all the implementation details from the library user (developer from hereinafter). Revertible operations, executed and compensated within the same transaction, have a full transactional behavior from the developer's perspective and, therefore, should be available to and used freely by the programmer.

The most appropriate time and location to revert any operation are, frequently, at the end of the transaction and when it finishes with either success or failure. If the operations to be reverted are executed inside a software library, the transaction beginning and end are, most probably, outside the boundaries of the library itself. Thus, the programmer does not have access to those code points and cannot easily have the reverting operations be executed at the end of the transaction without violating the library abstraction and exposing the library internals to the developer.

Memory management operations (memory allocation and release), fall into the class of revertible operations. If the transaction aborts, the allocation of a memory block may be reverted by the release of that same memory block.

Reverting a memory release operation cannot always be achieved by allocating a new identical memory block, as the contents of the initial memory block may have been permanently lost. A reasonable solution for this problem is to defer the memory release operation until there is a guarantee that the transaction will commit and the free operation will not have to be reverted.

When and where to revert an operation, depends on the nature of the operation itself and on its usage context. As there is no general rule, we propose a solution that is both flexible and generic. Flexible because it can adapt to multiple needs/requirements by allowing to execute the reverting operations in a variety of well-defined steps in the transaction life-cycle; and generic because it is not tied to any particular problem or solution model, and it can be used towards aims other than executing reverting operations.

3. Concept and Model

Memory transactions have a life cycle as depicted in Fig. 2. Once the transaction begins, it switches to the *active* state, where all the operations will take place. When attempting to commit, the transaction switches to the *partially committed* state and enters a validation phase, where all the operations executed in the *active* state are validated to ensure the transactional properties such as atomicity and isolation. If the validation phase succeeds, the transaction will switch to the *commit* state, where all the operation executed in the *active* phase will be made effective and visible to other transactions and then ends with a transition to the *terminated* state. If the transactional framework detects a violation of any of the transactional properties, either when executing an operation or at the validation phase in *partially committed* state, the transaction must abort, thus switching to the *aborted* state. In this state, all the operations executed in the *active* state will be discarded/reverted by the transactional framework, and ends with a transition to the *terminated* state.

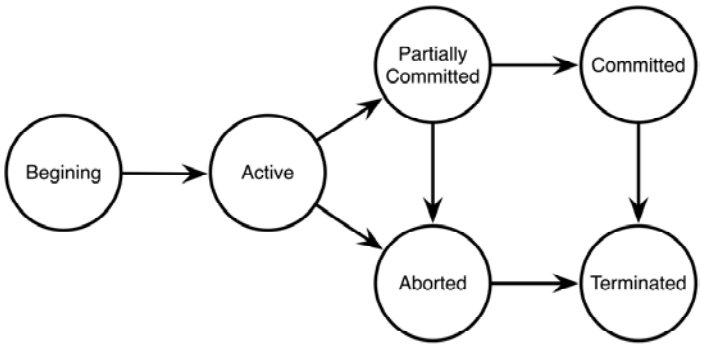

Fig. 2. Transaction life cycle state diagram.

Supporting revertible operations within transactions can be achieved by defining both, a reverting function (handler from herein after) and a point in the

transaction life cycle where that handler should be executed. We propose to extend the transaction life cycle with five new states associated to four important phases of the transaction life-cycle: *before commit*, *after commit*, *before abort*, and *after abort*. These phases and their associated states are illustrated in Fig. 3. The new states are represented with a dashed line.

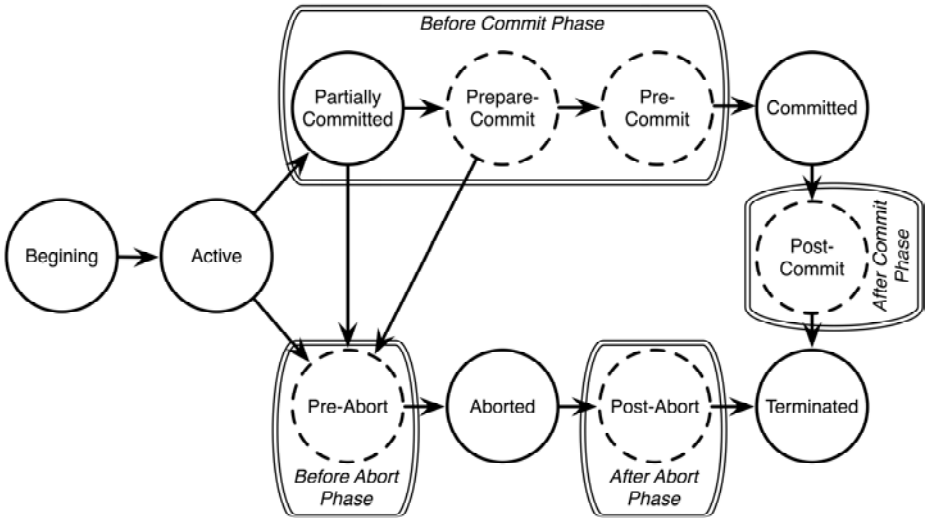

Fig. 3. Transaction life cycle state diagram with handler support

Each phase aggregates one or more states. In each of the new states, a set of user-defined handlers may be executed. Handlers must be registered after the beginning of the transaction and before its possible use. The handler lifetime is determined by its moment of registration until the transaction terminates. If the transaction is retried, handlers must be re-registered, thus there is no difference between the first execution of the transaction and its subsequent retries.

Prepare-commit handlers are executed in the context of the transaction to be committed and may force the transaction to abort. The memory validation step was done in the partially-committed state, thus prior to the *before-commit phase.* Therefore, prepare-commit handlers execute knowing that the memory transaction may commit. Each prepare commit handler returns a Boolean value: true meaning the transaction may proceed with commit, thus executing the next prepare-commit handler or switching to the pre-commit state; and false meaning the transaction must abort and, in this case, no more prepare-commit handlers will be executed and the transaction will switch to the pre-abort state.

Pre-commit handlers are also executed in the context of the transaction to be committed, but knowing that the transaction will definitely commit. Thus, if at least one of the pre-commit handlers is executed, all of them will.

Post-commit handlers are executed after committing the transaction and, therefore, are outside the transactional context.

Pre-abort handlers are executed just before aborting, thus in the context of the transaction to be aborted. If the STM framework supports automatic retrying of aborted transactions, these handlers are executed just before retrying.

Post-abort handlers are executed right after aborting the transaction, therefore outside the scope of a transaction. If the STM framework supports automatic retrying of aborted transactions, these handlers will not be executed, as the transaction will switch back form the aborted state straight to the beginning. However, they will be executed if the user explicitly aborts the transaction.

All the handlers in the *before-commit phase* are executed under the assumption that the memory transaction will commit, i.e., while a transaction T_i executes the *prepare-commit* and the *pre-commit* handlers, no other transaction T_j conflicting with transaction T_i can force T_i to abort. This leads to a situation where the code executed inside the *prepare-commit* and *pre-commit* handlers of T_i should never block the transaction, otherwise it may also block all the remaining T_j transactions conflicting with T_i or force them all to abort.

The adequate combination of *prepare-commit* and pre-*commit* handlers permit to execute a *two-phase-commit* protocol [12] between several transactional subsystems in which one of them is the memory transaction. As the name suggests, the *two-phase-commit* protocol includes two main phases: the *preparation phase*, where all transactions must first agree whether they will commit or abort; and the *commit/abort phase*, where all the transactions must perform the decision previously agreed. If the decision was to commit, the transaction manager will request that all the transactions commit. If the decision was to abort, the transaction manager will request that all the transactions abort.

Although both the *pre-commit* and the *pre-abort handlers* are executed within the context of the transaction, they cannot make use of transactional memory accesses. For *pre-commit* handlers, the memory transaction has already been validated and any new transactional access to memory could require new validations to be carried out. For *pre-abort* handlers, the transaction is potentially in an inconsistent state and no more transactional accesses to memory are allowed. Thus, the programmer must prevent and manage any potential data-races that may arise when processing/executing handlers.

4. Implementation

The model described in the previous section, was implemented as an extension to CTL [10], a Software Transactional Memory library for the C programming language.

In CTL, each handler is identified as a function pointer that is registered in the software transactional memory handler subsystem. There are two di`ff`erent types of function pointers, as illustrated in Fig. 4.

```
typedef int (*ctl_prepare_handler_t)(void *);
typedef void (*ctl_handler_t)(void *);
```

Fig. 4. Handler System API: handler functions types.

The type `ctl_prepare_handler_t` declares a pointer to a function to be executed as a *prepare-commit* handler. The type `ctl_handler_t` declares a pointer to a function to be executed as a *pre-commit*, *post-commit*, *pre-abort*, or *post-abort* handler.

Functions of type `ctl_prepare_handler_t` will return a `boolean` (*true*=1 or *false*=0) indicating, respectively, that the overall transaction can proceed or that it must abort. If and only if all of the *prepare-commit* handlers return *true*, the *pre-commit* handlers will be executed and the memory transaction will commit, otherwise none of the *pre-commit* handlers will be executed and the transaction will execute the *pre-abort* handlers and abort.

When registering a handler, the programmer must provide an extra argument with a pointer to a user-defined data structure. This data structure will later be given to the handler when (and if) it is executed. Following the C standard practice, if there is no need to pass data to the handler, the pointer may be initialized to NULL.

Each transaction has a list for each type of handlers. Each handler has an integer priority attribute, where smaller numbers correspond to a lower priority and bigger numbers correspond to a higher priority. The priority controls the order in which the handlers are executed. Handlers with higher priority are executed before handlers with a lower priority. Within the same priority, handlers are executed by registering order.

The API contains two functions for each type of handler: one requires the programmer to explicitly specify the priority attribute while the other does not, assuming a default priority. Figure 5 shows the registering functions for the *prepare-commit* handlers. The user defined data structure is referenced by the `void *args` pointer. The registering functions for the other classes of handlers are identical.

```
void ctl_register_prepare_handler_priority(
ctl_prepare_handler_t handler, void *args, int priority);

void ctl_register_prepare_handler(
ctl_prepare_handler_t handler, void *args);
```

Fig. 5. Handler System API: prepare-commit handler registering functions.

Handlers are eliminated once the associated transaction commits or aborts. In CTL, transactions aborting due to a concurrency conflict are automatically restarted and no *pos-abort* handlers are executed. P*ost-abort handlers* are

executed only for user-aborted transactions.

5. Using the Handlers

This Section will illustrate a case study—manipulating a linked list—, which makes use of the handler system just described to solve the problem introduced in Sect. 2, where the programmer needs to do some explicit memory management operations inside a library that will be executed in the context of a memory transaction.

When implementing the add operation for a linked list, there is the need to allocate memory to a new list node. If this memory allocation is executed inside a memory transaction that aborts and is automatically restarted, a new list node will be allocated, and the previously allocated list node will be dangling and will originate a memory leak in the program. In the proposed model, the library developer could register a pre-abort handler that would free the memory just allocated if the transaction aborts.

Figure 6 illustrates the situation just described, registering a pre-abort handler to revert the allocation of memory by releasing it. If the transaction aborts while adding a new node to a linked list, the handler will be executed and the memory released. Error processing, such as checking if the memory pointer is NULL, has been omitted for code simplicity.

```
void freevar (void *args) {
    free (args);
}

void add (List *list, void *item) {
    Node *typedef int (*ctl_prepare_handler_t)(void *);
typedef void (*ctl_handler_t)(void *);
r, (void *)node);
    node->next = NULL;
    node->value = item;
    TxStore (&(list->tail->next), node);
    TxStore (&(list->tail), node) ;
}
```

Fig. 6. Linked list add operation with handler system support.

A memory release cannot be directly reverted by allocating a new memory block, but the release operation can be deferred until the end of the transaction. This can be easily achieved with a *post-commit* handler. Figure 7 illustrates a conventional transactional memory based approach to remove the head node of the linked list.

```
void *removehead (List *list) {
    Node *node;
    void *value;
    node = (Node *)TxLoad (&(list->head);
    TxStore (&(list->head), node->next);
    value = (void *)TxLoad (&(node->value));
    free (node);
    return value;
}
```

Fig. 7. Linked list removehead() operation

Assuming that the `removehead()` function is called inside a memory transaction and that the transaction aborts after the `free()` operation, the transaction will be restarted and the `removehead()` function will be called once again. However, the list head pointer `list->head` would now be pointing to an invalid memory block, as it was already released in the call to `free()` in the previous execution. To avoid this problem, the memory release must be deferred until the transaction commits. This can be achieved by registering a *post-commit* handler to only release the node once the transaction commits. This solution is illustrated in Fig. 8. Much of the complexity of registering these handlers for explicit memory management can be hidden in C macros that would redefine `malloc` and `free` to alternative definitions and would call the replacement functions if necessary.

```
void freevar (void *args) {
    free (args);
}

void *removehead (List *list) {
    Node *node;
    void *value;
    node = (Node *)TxLoad (&(list->head);
    TxStore (&(list->head), node->next);
    value = (void *)TxLoad (&(node->value));
    ctl_register_pos_commit_handler (freevar, node);
    return value;
}
```

Fig. 8. Linked list removehead() operation with handler system support

This handler-based solution can be supported by a programming library or by a compiler and programming language constructs. The library-based solution was just described.

A compiler-based solution would necessarily have the compiler to transparently generate all the necessary code for registering the handlers and calling the replacement front-ends instead of the original functions without any user intervention.

In terms of general library development, each library function can register the appropriate handlers to revert or defer its effects to the commit/abort time, without the library user being aware of such handlers. In this way, even if the transaction boundaries are outside the scope of the library, the programmer has access to the transaction end by registering handlers that will revert or execute deferred operations whenever necessary. Revertible operations can, thus, be executed inside library functions that will be executed within memory transactions, and without changing the transactional behavior in the perspective of the library user.

6. Performance Evaluation

The overhead introduced by handler system into the CTL framework was evaluated by running a set of tests. These tests consist in series of operations over a set implemented as a Red Black Tree. The set implementation provides three methods: `insert()`, `delete()` and `get()`. Each set element has a key, which is used to index the element, and a value. Duplicate keys are not allowed, thus adding an element with an already existing key just updates its value. Only the `insert()` and `delete()`, which are read-write operations, make use of the handler system. In the `insert()` method, a *pre-abort* handler is registered to revert the allocation of a new node; and in the `delete()` method, a *post-commit* handler is registered to defer the release of a node memory until the commit of the transaction.

The testing program is parameterized with the relative probabilities of the `insert()`, `delete()` and `get()` methods. The tests ran were characterized by two different load patterns: one that simulates a read dominant context, with 5% of inserts, 5% of deletes and 90% of gets; and another that simulates a write dominant context, with 45% of inserts, 45% of deletes, and just 10% of gets.

The tests were performed in a Sun Fire X4600 M2 x64 server, with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz with 1024 KB cache. There were at most 16 threads competing for the 16 available processors, and the obtained results can be depicted in Fig. 9 for the read dominant context and in Fig. 10 for the write dominant context.

Each figure shows three different executions: one with no handlers support (*without handler*); another with the handler support activated but with empty handlers (*with empty handler*); and another with handler support activated and executing the appropriate code inside the handlers (*with handler do free*). For each test, seven runs of one minute each were executed, the maximum and minimum values were excluded and the average of the remaining five values was used for the benchmark.

As depicted in Fig. 9 all the three different executions are at the same level of performance. Because in the read dominant context most operations are read-only, one can infer that if the handlers are not used there is no significant performance penalty introduced by the handler system.

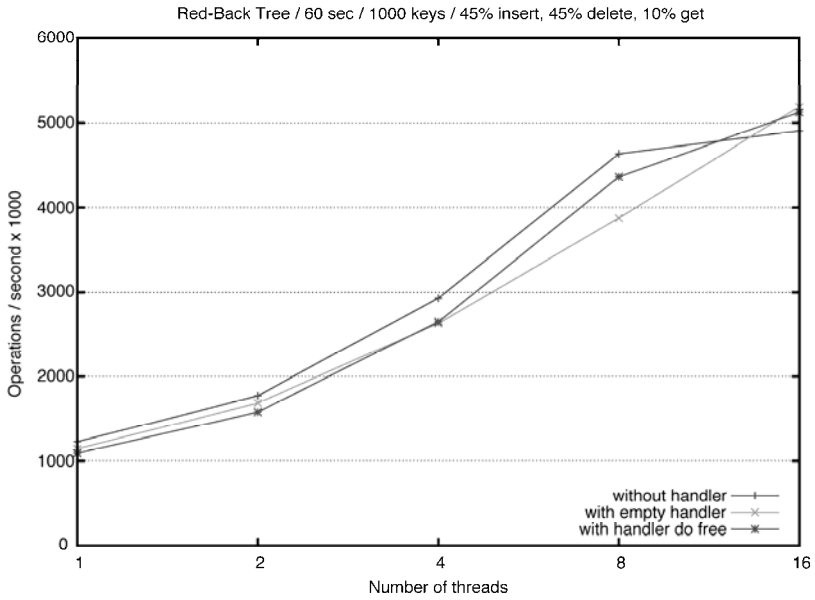

Fig. 9. Overhead introduced by the handler support in read-dominated environment

A slight increase in the performance penalty can be depicted Fig. 10, which shows a write dominant context. This increase is most notorious as the number of threads grows. There is a clear exception for sixteen threads, where the executions with handler support perform better than the one without handler support. Until now we could not find a definite explanation for this behavior, but we believe it may be related to data locality and cache coherency protocols. We may therefore conclude that the performance penalty inherent to the registration and execution of the handlers is still low. Additionally, when using the handler system, the program is correct and is freeing the memory where and when necessary, thus avoiding memory leaks.

7. Related Work

Harris in [13, 14] also uses callback handlers in the form of external actions to provide support for operations with side effects, such as console I/O in the Java programming language. The external actions are implemented using contexts, where each context represents a heap view for a specific thread. These contexts are exposed to programmers as immutable `Context` objects. Invoking an operation with side effects implies saving the heap in a `Context`

object and defers the operation until the end of the transaction, where it will be executed in the same context of the initial invocation. Besides the possible performance implications of this approach, there is a serious drawback on its usage to implement software libraries, as the library programmer would have to have control over the transaction boundaries (start and end). This restriction is not imposed by the approach proposed in this paper.

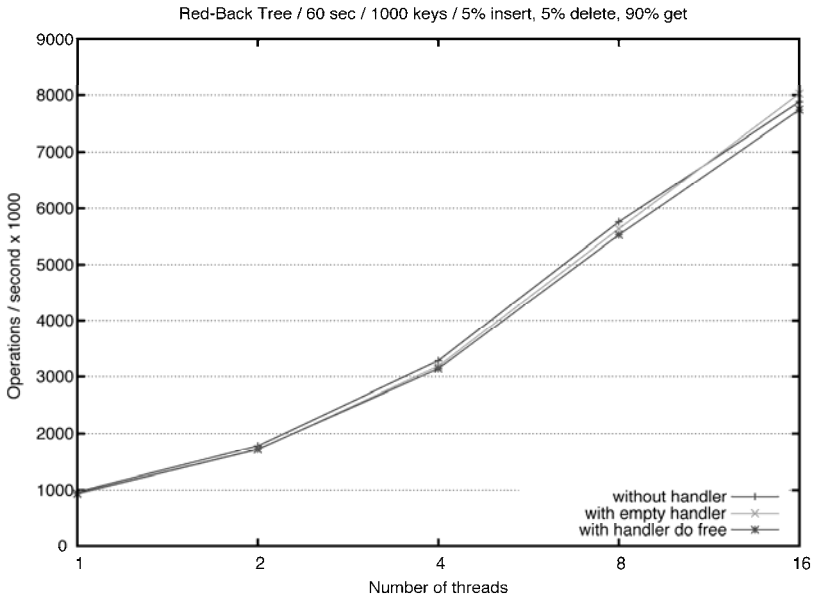

Fig. 10. Overhead introduced by the handler support in write-dominated environment.

DSTM2 [15] also provides the ability to register methods, in the form of `Callable<Boolean>` objects, to be invoked in specific instants of a transaction life-cycle: *onValidate*, *onCommit* and *onAbort*. *onValidate* methods are similar to the *prepare-commit* handlers, but executed before the transaction is validated (i.e., before the *partially-committed* state) and thus providing weaker guarantees, as even if all *onValidate* handlers return true, the transaction can still abort in the validation process. *onCommit* handlers are executed after committing the transaction and *onAbort* handlers are executed before aborting the transaction, and are equivalent, respectively, to the *post-commit* and *pre-abort* handlers. DSTM2 lacks the functionalities proposed with *pre-commit* and *post-abort* handlers, as well as the possibility to associate priorities with handlers to have control over the order in which handlers are executed.

Another recent STM library for the C programming language, TinySTM [16], also derived from TL2, implements function handlers that can be executed at di`ff`erent phases of a transaction. TinySTM handler system does not, however, support handlers of type *prepare-commit* and, thus, cannot participate in a *two-phase-commit* protocol. Handlers in TinySTM must be registered prior to the start of a transaction, making them unusable for library development.

8. Conclusions and Future Work

This paper described a way to extend the memory transaction life-cycle state diagram with new states and have these states associated to executing user-defined handlers. These user-defined handlers may be used to give revertible operations executed inside library functions a fully transactional behavior.

The handler-based technique described in this paper is flexible, generic and efficient. Flexible because by using an adequate combination of prioritized handlers it can easily adapt to new needs. Generic because the technique is not tied to any specific problem or solution, neither it is dependent on the specific model or implementation of a STM framework. Finally, efficient because the overhead introduced by the handler system is a low price to pay for having programs operating correctly (e.g., avoiding memory leaks). The handler-based proposed technique only depends on the programmer to correctly use the handlers and create the operationally effective solution.

Other completed works using the handler system proposed in this paper confirm the claimed properties: the full integration of memory and database transactions [17] was recently achieved by using the *two-phase-commit* to commit both the memory and database transactional systems or none of them; and a transactional file system [18] was also implemented an application library by making extensive use of handlers to defer some file system operations and to revert some others.

9. Acknowledgements

The authors would like to acknowledge the reviewers who made remarkable comments and suggestions. This work was partially supported by Sun Microsystems and Sun Microsystems Portugal under the “Sun Worldwide Marketing Loaner Agreement #11497”, by the CITI–Centro de Informática e Tecnologias da Informação and by the FCT/MCTES–Fundação para a Ciência e Tecnologia in the context of the Byzatium research project PTDC/EIA/74325/2006 and research grant SFRH/BD/41765/2007.

10. References

1. Shavit, N. and D. Touitou, Software transactional memory, in PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing. 1995, ACM: Ottowa, Ontario, Canada. p. 204-213.
2. Harris, T. and K. 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. 2003, ACM: Anaheim, California, USA. p. 388-402.
3. Herlihy, M., et al., Software transactional memory for dynamic-sized data

structures, in PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing. 2003, ACM: Boston, Massachusetts. p. 92-101.
4. Herlihy, M., V. Luchangco, and M. Moir, Obstruction-Free Synchronization: Double-Ended Queues as an Example. icdcs, 2003. **00**: p. 522.
5. Dice, D., O. Shalev, and N. Shavit, Transactional Locking II, in Distributed Computing. 2006, Springer Berlin / Heidelberg: Stockholm, Sweden. p. 194-208.
6. Ennals, R., Software Transactional Memory Should Not Be Obstruction-Free. 2006, Intel Research Cambridge Tech Report.
7. Cachopo, J. and A. Rito-Silva, Versioned boxes as the basis for memory transactions. Sci. Comput. Program., 2006. **63**(2): p. 172-185.
8. Dalessandro, L., et al., Capabilities and Limitations of Library-Based Software Transactional Memory in C++, in Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing. 2007: Portland, OR.
9. Adl-Tabatabai, A.-R., et al., Compiler and runtime support for efficient software transactional memory, in Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. 2006, ACM: Ottawa, Ontario, Canada.
10. Cunha, G., Consistent state software transactional memory, in Departamento de Informática. 2007, Universidade Nova de Lisboa: Lisbon.
11. Lourenço, J. and G. Cunha, Testing patterns for software transactional memory engines, in PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging. 2007, ACM: London, United Kingdom. p. 36-42.
12. Gray, J., Notes on data base operating systems. Operating Systems, 1978: p. 393-481.
13. Harris, T., Design Choices For Language-Based Transactions. 2003, UCAM-CL-TR.
14. Harris, T., Exceptions and side-effects in atomic blocks. Sci. Comput. Program., 2005. **58**(3): p. 325-343.
15. Herlihy, M., V. Luchangco, and M. Moir, A flexible framework for implementing software transactional memory, in OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. 2006, ACM: Portland, Oregon, USA. p. 253-262.
16. Felber, P., C. Fetzer, and T. Riegel, Dynamic performance tuning of word-based software transactional memory, in PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. 2008, ACM: Salt Lake City, UT, USA. p. 237-246.
17. Dias, R., Cooperative Memory and Database Transactions, in Departamento de Informática. 2008, Universidade Nova de Lisboa: Lisbon.
18. Martins, A., Transactional Filesystems, in Departamento de Informática. 2008, Universidade Nova de Lisboa: Lisbon.

Ricardo Dias has received his B.Sc. in Computer Science by the Faculty of Science and Technology of the New University of Lisbon in 2007. In 2008 he received his M.Sc. in Computer Science by the New University of Lisbon, and is currently doing his Ph.D. at the same university. His research activities are in the area of software transactional memory systems.

João Lourenço has received his B.Sc. in Computer Science by the Faculty of Science and Technology of the New University of Lisbon in 1991, and his M.Sc. and Ph.D. in Computer Science by the New University of Lisbon in 1994 and 2004, respectively. From 1992 to 2004 he has been a Teaching Assistant at the Computer Science Department of the Faculty of Science and Technology of the New University of Lisbon, and since 2004 he is an Assistant Professor at the same University. His teaching activities are in the area of Computer Architecture, Operating Systems and Parallel and Distributed Computing. His research activities are on parallel and distributed computing models and tools, and currently he is working on software transactional memory.

Gonçalo Cunha has received his B.Sc in Computer Science by the Technical Superior Institute of the Technical University of Lisbon in 2002. In 2007 he received his M.Sc. in Computer Science by the New University of Lisbon. His M.Sc. research activities were in the area of software transactional memory systems. He is currently working as a systems engineer in a consultancy company.

Received: July 16, 2008; Accepted: November 17, 2008.