Developing Libraries Using Software Transactional Memory

Ricardo Dias and João Lourenço

CITI–Centre for Informatics and Information Technology, and
Departamento de Informática, Universidade Nova de Lisboa
Portugal
{rjfd,joao.lourenco}@di.fct.unl.pt

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) locations. This paper aims at discussing how to support apparently irreversible operations within software libraries that will be used in a (software memory) transactional context.

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: we may have more computational processing power then we can (easily) use. To invert such a situation one needs to find and explore concurrency where before would write sequential code. The transactional programming model is an appealing approach towards such a goal, by making use of high-level constructs to deal with concurrency, being in this way a potential alternative to the classical lock based constructs such as mutexes and semaphores.

Since the introduction of Software Transactional Memory (STM) [9], this topic has received a strong interest by the scientific community. Opposed to the pessimistic approach used in locks-based constructs, transactional memory use optimistic methods for concurrency control, stimulating and enhancing concurrency. 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 programing language, therefore relying in the programmer to explicitly call a library API to do transactional memory accesses. Significant research work is using Concurrent Haskel as a testbed for runtime and compiler changes to support STM [7]. Many problems and difficulties still persist in using the STM programming model, supported by both software libraries [3] or directly by the compiler [1].

In this paper we will report on a problem that arose while developing small applications examples for the CTL [2] transactional memory library. CTL is a library-based STM implementation for the C programming language, derived from the TL2 [4] library. CTL extends the TL2 framework with new features and optimizations, and also solves some bugs found in the original framework [8].

Section 2 describes the motivation and context for the problem covered in this paper; Sec. 3 describes a new idea to overcame the difficulties found; in Sec. 4 will present the implementation of the solution; Sec. 5 will introduce some use case examples; inSec. 6 and Sec. 8 concludes and presents some future research work on this line.

2 Motivation

Programming using the STM model is straight forward: if we plan to do a set of memory operations atomically (do all or none of the operations) and/or want to do an access to a memory location that will potentially conflict with another concurrent control flow (*thread*), the set of operations should be enclosed within a memory transaction.

When the STM programming model is directly supported by a programming language, a transactional code block may look as illustrated in Fig. 1 on the left. When the STM programming model is supported by a software library, the same code block may look as illustrated in Fig. 1 on the right. Semantically, both code blocks are equivalent.

```
atomic {
    transact_code_block();
}
```

```
start_T();
    transact_code_block();
end_T();
```

Fig. 1. Transactional code block supported by the programming language (*left*) or by a software library (*rigth*).

Using the transactional memory model in every day programming may seem to be a simple idea, but it may turn out to be a nuisance. The fact is that an application is not solely made of memory changes and, typically, need to perform other operations that are not reversible by the STM libraries, such as write data into a file, read data from a socket and memory management (allocation/deallocation).

When developing a library, the programmer aims at creating a *black box* behind a well defined interface, hiding all the implementation details from the library user. If such libraries are to be used in concurrent environments, programmers must protect them against concurrency hazards, and Software Transactional Memory can be an approach towards such a goal. However, this approach can raise problems as illustrated in Fig. 2.

If a library code block is supposed to be executed within a memory transaction whose boundaries (start and end of the transaction) are defined by the library user, than the library developer has no control over neither the start nor the end of the transaction.

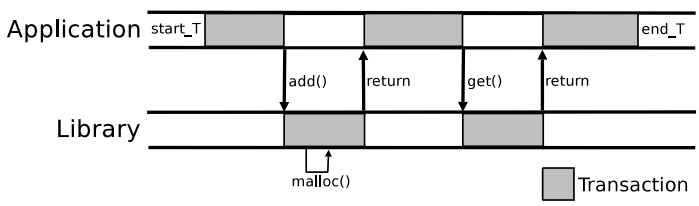

Fig. 2. Calling a transaction code block in a library.

Allocating and freeing memory are irrevocable operations and, thus, cannot be executed freely inside a memory transaction. Although these operations are non-trnasactional, some of them are compensable. This means that the operations can be undone not by reverting their effects but by execution a second operation that will compensate the effects of the first one. However, not all irrevocable operations are compensable and those that are compensable must obey to an isolation restriction: the effects of an irrevocable, but compensable, operation may be compensated only if no other section of code outside the currents transaction depends on (the effects of) the operation to be compensated.

Memory management (allocation and deallocation of memory) fall into the class of irrevocable but compensable operations. Allocating a memory block may be compensated by freeing that memory block. But freeing a memory block cannot be always compensated by allocating another memory block, as the initial memory contents may have been lost. Assuming that this irrevocable operations may be compensated, one must define when to execute the compensating operations.

We propose a solution that is, simultaneously, generic and elegant. Generic because it can be used to solve this and many other problems that arise when using the STM programming model. Elegant because allows the software library to execute compensating operations without the intervention and/or knowledge from the library user.

3 Concept and Model

Our proposal allows the programmer to create inverse functions and to decide when such inverse functions must (and will) be executed. Such a functionality is accomplished by the use of handlers. Handlers will be called at important moments in the life-time of a transaction, as illustrated in Fig. 3.

These important moments are:

- *Pre-commit handlers*: These handlers are executed in the context of the transaction to be committed. The memory validation step is done prior to the execution of the pre-commit handlers, thus, these handlers execute knowing that the memory transaction may commit;

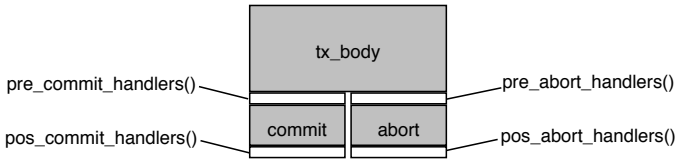

Fig. 3. Handlers for STM compensation actions.

- *Post-commit handlers*: These handlers are executed after committing the transaction and, therefore, are outside the transactional context;
- *Pre-abort handlers*: These handlers are executed just before aborting, therefore in the context of the transaction to be aborted. If the STM engine supports automatic retry of a transaction, the handlers are executed just before retrying;
- *Post-abort handlers*: These handlers are executed right after aborting the transaction, therefore outside the scope of a transaction.

All these type of handlers must be registered in the context of a transaction (inside the bounds of a transaction). The life-time of any type of handler is determined by the time of registration until the end of a transaction, either by committing or aborting/retrying. On registering a handler, the programmer may, optionally, pass some data to the handler. This data will be considered later when the handler is executed.

Pre-commit handlers are divided into two categories: *prepare-commit handlers* and *commit handlers*. The former may decide to allow (or not) the memory transaction to commit. The latter are still executed before committing the transaction, but the transaction will irreversibly commit. These handlers are executed sequentially: all *prepare-commit handlers* are executed in first place, then followed by the execution of the *commit handlers*.

Figure 4 represents the execution of each type/category of handler as a state in a transaction life-time state diagram.

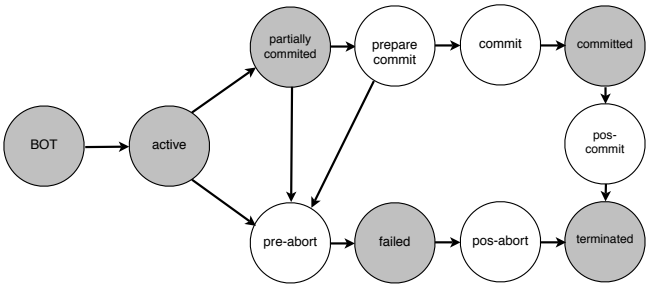

Fig. 4. Transaction life-time state diagram with handler support.

Although pre-commit and pre-abort handlers are executed within the context of the transaction, they cannot make use of transactional memory accesses, as the memory transaction has already been validated and new transactional accesses to memory could require new validations to be carried out. Thus, the programmer has the responsibility of preventing and managing any 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, a Software Transactional Memory library for the C programming language.

Each handler is identified as a function pointer that is registered in the handler system. There are two different types of function pointers:

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

The type `ctl_prepare_handler_t` declares a pointer to a function to be executed as a *prepare-commit handler*.

This function has one parameter that will point to a user-defined data structure to be passed as an argument to the handler when it is executed, and returns a `boolean` (*true=1* or *false=0*) indicating that the overall transaction can proceed, or that it must abort. If all of the *prepare-commit handlers* return *true* the *commit handlers* will be executed and the memory transaction will commit, otherwise none of the *commit handlers* will be executed and the transaction will abort.

Each transaction has a list for each type of handlers. Each handler has a priority attribute. The priority controls the order in which the handlers are executed. Handlers with a high 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 function for the *prepare-commit handlers*. When registering a handler, it is possible to pass data into the handler. This data must be passed as a `void` pointer.

Handlers are eliminated once the associated transactions commit or abort. In CTL, transactions aborting due to a concurrency conflict are automatically restarted. In this case, the handlers are eliminated after executing the last *pre-abort handler* and before the transaction is restarted. Also, *pos-abort handler* will only be executed for user-aborted transactions, otherwise they are restarted automatically and always finishes with a commit state.

```
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.

5 Using the Handlers

We will describe how to use the handler system as described above to solve the problem introduced in Sec. 2, where a library needs to manage memory inside memory transactions.

When implementing the `add` operation of a linked list, this operation needs to allocate memory to a new list node. If this memory allocation is executed inside a memory transaction, and if the transaction aborts and is automatically restarted, a new list node will be allocated and the previous one will originate a memory leak in the program. In this case the library developer could register a *pre-abort handler* to free the allocated memory in case of the abort/restart of the transaction.

Figure 6 ilustrates the use of a *pre-abort handler* to compensate the operation of memory allocation, when the transaction aborts while adding a new node to a linked list.

```
void freevar (void *args) {
    free (args);
}
void add (List *list, void *item) {
    Node *node;
    node = malloc (sizeof (*node));
    ctl_register_pre_abort_handler (freevar, 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.

The inverse problem of compensating a memory deallocation problem is also easy to solve with a *pos-commit handler*. As an example, we will consider the removing of the head node of the linked list, as illustrated in Fig. 7.

```
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.

If this `removehead()` function is called inside a memory transaction, and the transaction aborts after the `free()` operation, the transactions will be restarted and the function will be called once again, but now the head pointer `list->head` is pointing to an invalid memory block, because it was already released in the call to `free` in the previous execution. To solve this problem, one must delay the memory deallocation until the transaction commits. This can be achieved by registering a *pos-commit handler* to free the respective node as depicted in Fig. 8.

```
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 solution could be supported at either, programming language/compiler or library level. The library based solution was illustrated before. A compiler based solution would 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.

In terms of general library development, each library can register the appropriate handlers to delay or reverse its effects to the commit/abort time, without the library user being aware of such handlers.

6 Performance Evaluation

We preformed a simple test to evaluate the overhead introduced by handler system into the CTL engine. The test uses a single linked list, protected with CTL using the handler system support, and random inserts, removes and lookup of nodes to/from the linked list. The test ran in a computer with two dual-core nodes, Intel(R) Xeon(R) CPU 5150 @ 2.66GHz with 4096 KB cache, and the obtained results can be depicted in Fig. 9. We ran tests with as much as 16 threads competing for the 4 available computing nodes.

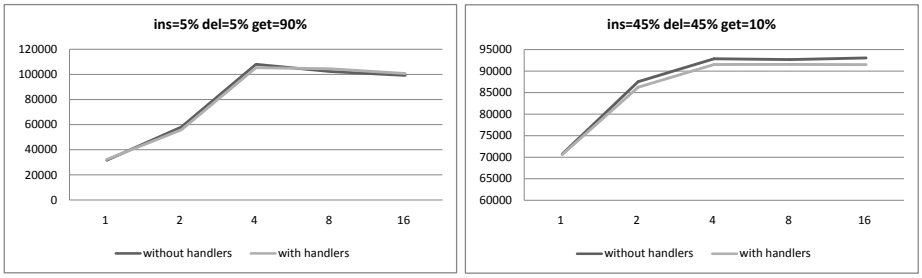

Fig. 9. Overhead introduced by the handler support in read-dominated (left) and write-dominated (right) environments.

In the evaluation test, write operations (insert and remove) require handlers to be executed, while the read-only operation (lookup) does not.

In the read-dominate environment, depicted in the left sub-figure, 5% of the transactions are inserts, other 5% are removes (deletes) and 90% are lookups. Only 10% of the completes transactions require a handler to be executed. From the results one can infer that the overhead introduced by the handler system when it is not used is null.

In the write-dominate environment, depicted in the right sub-figure, 45% of the transactions are inserts, other 45% are removes (deletes) and only 10% are lookups. Handlers are invoked in 90% of the completed transactions. From the results one can infer that the overhead introduced by the handler system is less than 4% in the worst case.

7 Related Work

Tim Harris in [6], also uses call-back 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. This work was derived by an earlier approach by the same author in [5]. These external actions are implemented using a copy of the heap in the moment of the invocation of an I/O operation inside a transaction. This invocation is delayed until the end of the transaction, and then executes the I/O operation in the same context (using the heap copy) in which the invocation was made. A drawback of this approach is that, if used to implment libraries as described in this paper, the library would have to have control over start and end of the transaction. To our best knowledge to date, no other work has addressed this matter.

8 Conclusions and Future Work

The handler-based technique presented in this paper is a generic and elegant approach to solve the problem of executing irrevocable (but compensable) operations in the context of a software memory transaction, at a negligible cost. It can also be used to easily revert irrevocable (but, again, compensable) operations inside a library that will be executed within a memory transaction. This technique is not tied to any specific problem and, therefore, to the solution of a single/unique problem; neither it is dependent on the specific model or implementation of a STM framework. The proposed technique only depends on the programmer to correctly use the handlers and create the operationally effective solution.

This handler system could also be a good solution to integrate database transactions with memory transactions, using the two phase commit to commit both memory and database transactional systems or none. Also, this solution can scale to use more than one database at the same time. Ongoing work towards such a goal is being carried out by the paper authors.

Acknowledgements

This work was partially funded 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.

References

1. Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 2006 Conference on Programming language design and implementation*, pages 26–37. Jun 2006.
2. Gonçalo Cunha. Consistent state software transactional memory. Master's thesis, Universidade Nova de Lisboa, November 2007.
3. Luke Dalessandro, Virendra J. Marathe, Michael F. Spear, and Michael L. Scott. Capabilities and limitations of library-based software transactional memory in c++. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Transactional Computing*. Portland, OR, Aug 2007.

4. Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Distributed Computing*, volume 4167, pages 194–208. Springer Berlin / Heidelberg, October 2006.
5. Tim Harris. Design choices for language-based transactions. Technical report, UCAM-CL-TR, August 2003.
6. Tim Harris. Exceptions and side-effects in atomic blocks. *Sci. Comput. Program.*, 58(3):325–343, 2005.
7. 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.
8. João Lourenço and Gonçalo 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*, pages 36–42, New York, NY, USA, 2007. ACM.
9. Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.