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 (STM) 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 a memory transaction.

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 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 using high-level constructs to deal with concurrency, being in this way a potential alternative to the lock based constructs such as mutexes and semaphores.

Since the introduction of Software Transactional Memory (STM) [10], 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, may it be supported by software libraries [3] or directly by the compiler [1].

In this paper 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; Section 3 describes a new idea to overcame the difficulties found; and Section 5 concludes and presents some future research work on this line.

2 Developing Programming Libraries

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 programming library, the same code block may look as illustrated in Fig 1 on the right.

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

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

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

Such a transactional code block, including the transaction delimiters, may be enclosed within a software library if the STM framework supports nesting of transactions [9]. However, there may be the case where the transactional code block is implemented inside a library and supposed to be executed within a memory transaction delimited by the programmer (library user), as illustrated in Fig 2. This means that the library developer has no control over the start neither the end of the transaction.

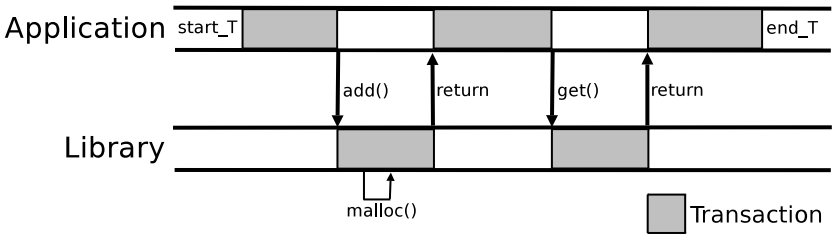

Fig. 2. Calling a transaction code block in a library

If the transactional code block needs to manage memory by allocating and/or freeing memory, a serious problem may arise: allocating and freeing memory are irrevocable operations, although one can define partial compensation 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 the irrevocable operations may be compensated, there is another difficulty: when to compensate the 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 compensate operations without the intervention and/or knowledge from the programmer (library user).

3 STM Handler System

Our proposal to solve the problems identified in the previous section is to allow 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 a handlers. Handlers will be called at important moments in the life-time of a transaction, as illustrated in Fig. 3.

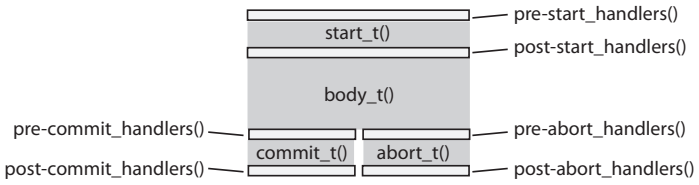

Fig. 3. Handlers for STM compensation actions

- *pre-start handlers*: These handlers are permanent and common to all transactions and will be executed just before any transaction starts, therefore they are not executed in a transactional context;
- *post-start handlers*: These handlers are permanent and common to all transactions and will be executed just after a transaction starts, therefore they are executed in a transactional context;
- *pre-commit handlers*: These handlers may be permanent or volatile. Permanent handlers will be executed first and are common to all transactions. Volatile handlers will be executed last and are specific to the current transaction and will vanish as soon as the transaction terminates. The handlers are executed in the context of the transaction to be committed and, when

called, the validation step has already been done and it is possible to commit the memory transaction;
- *post-commit handlers*: As above, these handlers may be permanent or volatile. When these handlers are called the memory transaction has already committed, therefore they are not executed in a transactional context;
- *pre-abort handlers*: These handlers may be permanent or volatile. If a transaction must be aborted, either due to an explicit (user called abort) or implicit abort (forced by the STM framework), these handlers are called 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 right before the retry;
- *post-abort handlers*: These handlers may be permanent or volatile and are executed right after aborting the transaction, therefore outside the scope of a transaction. In STM frameworks that support automatic retry of transactions these handlers are executed only for explicit (user requested) aborts.

Given the problem referenced in Sec. 2, we propose to have front-ends to the *malloc()* and *free()* calls. The *malloc()* front-end will register a pre-abort handler to release the memory in case of an explicit or implicit abort. The *free()* front-end will register a pos-commit handler to correctly quiesce [8] and release the memory. The pseudo-code for these operations is presented in Fig. 4.

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

```
void *ctl_malloc(size_t size)
{
  /*allocate some memory*/
  void *ptr = malloc(size);

  register_pre_abort_handler(
    freevar, ptr
  );
}
```

```
void *ctl_free(void *prt)
{
  /* delay the free action
     until the commit */
  register_pos_commit_handler(
    freevar, ptr
  );
}
```

Fig. 4. Pseudo-code for *malloc* and *free* front-ends

This solution could be supported at both, programming language/compiler or library level. A compiler supported solution would have the compiler transparently generate all the necessary code for registering the handlers and calling the replacement front-ends instead of the original functions. In a library level solution, the programmer (library user) has to explicitly register the compen-

sating functions as appropriate handlers, but there may be a problem on how to pass information from the current transaction scope to the handlers scope.

This handler system could also be a good solution to integrate database transactions with memory transactions, using the two phase commit approach to commit the changes in database with the changes made in memory. Also, this solution can scale to use more than one database at the same time. In terms of library development, each library could register the handlers which could be needed to reverse some effects, such as memory allocation, without the one that uses the library, be aware of such handlers.

4 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 when developing libraries. To use this approach, the library as to control the start and end of the transaction, because *external actions* only work inside the context of a memory transaction, hence a programmer can not develop a library using this approach unless it does know that the library functions are going to be invoked in the context of memory transactions. To our best knowledge to date, no other work as address this matter.

5 Conclusions and Future Work

The handler-based technique presented in this paper is a good and generic approach to solve the problem of executing apparently irrevocable operations in the context of a software 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 create the operationally effective solution.

Despite of the general technique as described in this paper, some details need to be specified in a more formal perspective, such as the specification of the trigger conditions and scope o such triggers. Implementations are also needed to test and evaluate the solution. We will implement this technique as an extension to the CTL framework and will use it aiming at integrating database and memory transactions.

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. J. Eliot B. Moss. *Nested transactions: an approach to reliable distributed computing.* PhD thesis, Massachusetts Institute of Technology, April 1981.
10. 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.