Technical Report

Revisiting Transactions in Ada

António Barros

Luís Miguel Pinho

HURRAY-TR-110707

Version:

Date: 07-25-2011

Revisiting Transactions in Ada

António Barros, Luís Miguel Pinho

IPP-HURRAY!

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8340509

E-mail:

http://www.hurray.isep.ipp.pt

Abstract

Classical lock-based concurrency control does not scale with current and foreseen multi-core architectures, opening space for alternative concurrency control mechanisms. The concept of transactions executing concurrently in isolation with an underlying mechanism maintaining a consistent system state was already explored in fault-tolerant and distributed systems, and is currently being explored by transactional memory, this time being used to manage concurrent memory access. In this paper we discuss the use of Software Transactional Memory (STM), and how Ada can provide support for it. Furthermore, we draft a general programming interface to transactional memory, supporting future implementations of STM oriented to real-time systems.

Revisiting Transactions in Ada

António Barros and Luís Miguel Pinho
CISTER/ISEP
Polytechnic Institute of Porto
Porto, Portugal
{*amb,lmp*}*@isep.ipp.pt*

***Abstract*—Classical lock-based concurrency control does not scale with current and foreseen multi-core architectures, opening space for alternative concurrency control mechanisms. The concept of transactions executing concurrently in isolation with an underlying mechanism maintaining a consistent system state was already explored in fault-tolerant and distributed systems, and is currently being explored by transactional memory, this time being used to manage concurrent memory access. In this paper we discuss the use of Software Transactional Memory (STM), and how Ada can provide support for it. Furthermore, we draft a general programming interface to transactional memory, supporting future implementations of STM oriented to real-time systems.**

I. Introduction

Current architectures based on multiple processor cores in a single chip are becoming widespread, and are challenging our ability to develop concurrent and parallel software. The tendency to integrate even larger number of cores will further impact the way systems are developed, as software performance can no longer rely on faster processors but instead on efficient techniques to design and execute parallel software. It is also important to note that developing scalable and safe concurrent code for such architectures requires synchronisation control mechanisms that are able to cope with an increasing number of parallel tasks, providing the necessary building blocks for modular, concurrent, and composable software.

In uniprocessor systems, lock-based synchronisation became the *de facto* solution to avoid race conditions, despite the well-known pitfalls, such as complexity, lack of composability [1] or (bounded) priority inversion. In multiprocessor systems, lock-based synchronisation becomes more problematic. Coarse-grained locks serialise non-conflicting operations (which could actually progress in parallel) on disjoint parts of a shared resource, and may cause cascading or convoying blocks [2], wasting the parallel execution opportunities provided by such architectures. Fine-grained locks increase the complexity of system design, affecting composability seriously, and produce an increasing burden for the programmer. In multiprocessors, non-blocking approaches present strong conceptual advantages [3] and have been shown in several cases to perform better than lock-based ones [4].

Transactional memory is a concept that has been researched in parallel systems for nearly two decades. Although the first proposal of transactional memory was hardware-based [5], it was soon followed by a software-based adaptation [6]. Software Transactional Memory has the advantage of being easily reconfigurable and to enable transactional support on architectures that do not have native support for atomic operations. Currently, STM research is well ahead comparatively with hardware-support for transactions.

Under the transactional memory paradigm, critical sections are executed optimistically in parallel while an underlying mechanism maintains the consistency of shared data. Data is kept consistent by serialising critical sections, that is, the outcome is as if critical sections were executed atomically, in sequence. Concurrent critical sections may attempt to perform conflicting data accesses; such conflicts are usually solved by selecting the critical section that will conclude, and aborting or delaying the contenders. Due to this similarity with database transactions, in which an atomic sequence of operations can either commit or abort, critical sections are called *transactions*.

Transactional Memory promises to ease concurrent programming: the programmer must indicate which code belongs to a transaction and leaves the burden of synchronisation details on the underlying STM mechanism to conserve the consistency of shared transactional data. This approach has proved to scale well with multiprocessors [7], delivers higher throughput than coarse-grained locks and does not increase design complexity as fine-grained locks do [8].

The advantages of STM are counterbalanced with increased times in memory accesses, memory utilisation overhead and speculative execution of code. The latter is, in fact, a challenge that has to be addressed if STM is to be applied to real-time systems, because it is not admissible that one transaction can be unboundedly aborted and forced to repeat. Some work on the field of STM on multiprocessor real-time systems was already published [9]–[12].

Several implementations of STM were implemented for programming languages such as java, C++, C# and Haskell, but currently there is no implementation for Ada. It is thus in this context that this paper revisits previous work on transactions support in Ada, for fault-tolerant systems,

and proposes a programming interface to support Software Transactional Memory. A goal is for this interface to be independent of an eventual STM implementation, so programs can, for instance, change the contention mechanism (the algorithm that manages conflicting operations), according to application specific characteristics.

The paper is structured as follows. Section II revisits previous work on transaction support in Ada for fault-tolerant and distributed systems. Afterwards, an introduction to the STM essential issues is given in Section III. In Section IV, we propose a general programming interface to access STM functionality. This paper terminates with conclusions and perspectives for further work in Section V.

II. Previous work on transactions in Ada

Transaction support in Ada has been a subject of research in the field of fault-tolerant systems. The concept of a transaction grouping a set of operations that appear to be executed atomically (with respect to other concurrent transactions) if it had success, or having no effect whatsoever on the state of the system if aborted, is quite appealing as a concurrent control mechanism for fault-tolerant and/or distributed systems. The ability to abort a transaction due to data access contention or to an unexpected error and, consequently, rolling back any state modifications, automatically preserves the system in a safe and consistent state. Safe consistent states can be stored in a durable medium, so they might be available even after a crash.

The loose parallelism provided by the isolation property of the transactional paradigm is appealing for systems based on multiple processors (either clustered or distributed) and the inherent backward recoverability mechanisms suit fault-tolerant concerns.

Two paradigmatic implementations of transaction support in Ada are the Transactional Drago [13], [14] and the OPTIMA framework [15].

Both proposals share many common attributes, aiming to support competitive concurrency (between transactions) and cooperative concurrency (inside a transaction). The two provide the essential interface to start, commit and abort transactions. Transactions can be multithreaded, i.e. multiple tasks can work on behalf of a single transaction. Both implementations support nested transactions and exception handling.

Despite the similarities, both implementations take different approaches.

Transactional Drago is an extension to the Ada language, so it can only be used with compilers that include this extension. Transactions are defined using the transactional block, an extension that resembles an Ada block statement, but identified with the keyword `transaction` [14]. The transactional block creates a new scope in which data, tasks and nested transactions can be defined. Data declared inside a transactional block is volatile and subject to concurrency control. Tasks inside a transactional block work cooperatively on behalf of the transaction and their results will dictate the outcome of the transaction.

The transactional block provides a clean syntax, defining clearly the limits of the transaction, without the need for an explicit abort statement. Aborts are triggered by the raising of unhandled exceptions from within the transactional block. The following code sample illustrates a transactional block.

```
transaction
  declare
    -- data declared here is subject to
        concurrency control
  begin
    -- sequence of statements
    -- can include tasks that work on behalf of
        transaction
    -- can include nested transactions
  exception
    -- handle possible exceptions here...
end transaction;
```

The OPTIMA framework is provided as a library, which does not modify the language. In this framework, a transaction is created with the command `Begin_Transacion` and, depending on the result of the computation, ends with either `Commit_Transaction` or `Abort_Transaction`. The OPTIMA framework supports open multithreaded transactions, so the additional command `Join_Transaction` allows one task to join and cooperate on an ongoing transaction. When a task calls `Begin_Transacion` or `Join_Transaction`, it becomes linked to the transaction via the `Ada.Task_Attributes` standard library unit. From that moment on, the transaction support is able to determine the transaction context for the task. The following code sample illustrates a task that starts a transaction.

```
begin
  Begin_Transaction;
  -- perform work
  Commit_Transaction;
exception
  when ...
    -- handle recoverable exceptions here...
    Commit_Transaction;
  when others =>
    Abort_Transaction;
    raise;
end;
```

This example shows how the use of the exceptions mechanism in this framework permits to define handlers for foreseen exceptional cases, thus allowing forward recovery in such cases. However, unexpected exceptions will abort the transaction, like in Transactional Drago.

III. Lightweight memory transactions

STM shares the basic principles of transactions from databases and fault-tolerant systems: transactions can execute speculatively in parallel, but their effects should appear as if they executed atomically in sequence. However, the

field of application has particular characteristics. Transactions are expected to be the size of typical critical sections, *i.e.* as short as possible. In the STM perspective, the transaction must conclude its work, so even if it aborts, the transaction should try again the necessary number of times until it is allowed to commit.

The STM mechanism keeps track of accesses to transactional objects that are exclusively memory locations (words or structures). Since memory accesses to transactional objects become indirect accesses, the STM mechanism must be light enough to avoid performance issues.

Unlike the previous work oriented to fault-tolerant systems, STM does not intend to store consistent states in a persistent medium, but simply to keep data in memory consistent. Furthermore, the multithreaded transaction concept does not apply: a transaction belongs exclusively to a task, *i.e.* it is part of the sequential code of the task.

In comparison with database transactions and fault-tolerant transactions, transactional memory transactions can be considered lightweight memory transactions [16].

Transactions can be divided in two classes, according to the transactional memory access pattern:

- **read-only transactions**, in which the transaction does not try to modify any transactional object, and
- **update transactions**, in which the transaction tries to modify at least one transactional object.

A conflict may occur when two or more transactions concurrently access one object and at least one of the accesses tries to modify the object: if the updating transaction commits before the contenders, the contenders will be working with outdated data and should abort.

Conflicts are typically solved by selecting one transaction that will commit and aborting the contenders. The performance of a STM is therefore dependent on the frequency of contention, which has direct effect on the transaction abort ratio. Thus, STM behaves very well in systems exhibiting a predominance of read-only transactions, short-running transactions and a low ratio of context switching during the execution of a transaction [17].

STM implementations may differ in multiple ways, namely [18]:

- **version management** – how tentative writes are performed;
- **conflict detection** – when conflicts are detected;
- **granularity of conflicts** – word, cache-line or object granularity.

Eager version management allows the object to be immediately modified, but requires that the object is unavailable to concurrent transactions until the updating transaction commits. If the updating transaction aborts, the value of the object is rolled-back to the previous version. *Lazy version management* defers the update of the object until the transaction commits, leaving the object available to other concurrent transactions. This implies that a transaction must work with an image of the object that will eventually replace the value of the transactional object.

Conflicts can be detected immediately, under *eager conflict detection*, or deferred until one transaction tries to commit under *lazy conflict detection*. Eager conflict detection inhibits a transaction to continue once it faces a conflict that will not win. Lazy conflict detection permits transactions to execute in total isolation and only when a transaction tries to commit, conflicts are detected and solved. Eager conflict detection better serves write-write conflicts, since one of the transactions will inevitably abort, but lazy conflict detection can be more efficient with read-write conflicts, as long as read-only transactions commit before update transactions [19].

The granularity of conflict detection determines the possibility of false conflict detection. Finer granularity means less false conflicts detected and lower transaction abort ratio, but at the expense of higher memory overheads.

Regardless which attributes are selected for an actual STM implementation, transactions will eventually be aborted, and some transactions may present characteristics (*e.g.* long running, low priority) that can potentially lead to starvation. In parallel systems literature, the main concern about STM is on system throughput, and the contention management policy has often the role to prevent livelock (a pair of transactions indefinitely aborting each other) and starvation (one transaction being constantly aborted by the contenders), so that each transaction will eventually conclude and the system will progress as a whole. In real-time systems, the guarantee that a transaction will *eventually* conclude is not sufficient to ensure the timing requirements that are critical to such type of systems: the *maximum time to commit* must be known. The verification of the schedulability of the task set requires that the WCET of each task is known, which can only be calculated if the maximum time used to commit the included transaction is known. As such, STM can be used in real-time systems as long as the employed contention management policy provides guarantees on the maximum number of retries associated with each transaction.

Recently, we have proposed new approaches to manage contention between conflicting transactions, using on-line information, with the purpose of reducing the overall number of retries, increasing responsiveness and reducing wasted processor utilization, while assuring deadlines are met [12]. With our proposed policy, conflicting transactions will commit according to the chronological order of arrival, except if an older contender is currently pre-empted. This approach is fair in the sense that no transaction will be chronically discriminated due to some innate characteristic. But most importantly, this approach is predictable, because the time overhead taken by a transaction until commit depends solely on the ongoing transactions at the moment the transaction arrives, being independent of future arrivals of other trans-

actions, except for conflicting transactions executing in the same processor that arrive after the job being pre-empted by another job with higher urgency.

IV. Providing STM support in Ada

Essential support to STM can be implemented in a library, without introducing modifications in the Ada programming language, easing the portability of an STM service, without the need to modify compilers and debuggers. The drawback is that the programmer must adhere to a somewhat less clean syntax.

The following code illustrates how a very simple transaction should be written.

```
-- we need a transaction identifier structure
My_Transaction : Transaction;

-- start an update transaction
My_Transaction.Start(Update);

loop
  -- read a value from a transactional object
  x := trans_object_1.Get(My_Transaction);

  -- write a value to a transactional object
  trans_object_2.Set(My_Transaction, y);

  -- try to commit transaction
  exit when My_Transaction.Commit;

exception
  -- handle possible exceptions here...
end loop;
```

This example shows how the initialisation of the transaction and the retry-until-commit loop have to be explicitly written.

Our STM perspective requires two key classes of objects: the *transactional object* and the *transaction identifier*.

The transactional object encapsulates a data structure with the transactional functionality. For instance, a write operation will not effectively modify the value of the object if the STM applies lazy version management.

The transaction identifier is a structure that stores the data required by the contention manager to apply the conflict solving policy chosen for the system.

A. Transactional object

A transactional object is a type of class that wraps a classical data structure with the transactional functionality. The interface provided is similar to the non-transactional version, but adds the operations required to maintain the consistency of the object, according to the implementation details of the STM.

Thus, for every access, the identification of the transaction is required, either to locate the transaction holding the object (case of eager version management) or track all transactions referring the object (case of lazy version management). In each case, the object must locate one or all accessing transactions, respectively, so under contention, transactions attributes are used to determine which transaction is allowed to proceed, according to the contention management policy.

Reading accesses can also be tailored for read-only transactions, if a multi-version STM is in use. Transparently, the transactional object can return the latest version to an update transaction, or a consistent previous version to a read-only transaction.

```
-- Transactional object
package Transactional_Objects is
  type Transactional_Object is tagged private;
  -- examples of transactional class methods
  procedure Set(T_Object: Transactional_Object;
                Transaction_ID : Transaction;
                Value : Object_Type);
  function Get(T_Object: Transactional_Object;
               Transaction_ID : Transaction)
           return Object_Type;
private
  type Transactional_Object is tagged
  record
    Current_Value : Object_Type;
    Accesses_Set : <list of pointers to
        transaction identifiers>
    -- some other relevant fields...
  end record;
end Transactional_Objects;
```

B. Transaction identifier

The transaction identifier provides the transactional services to a task, uniquely identifying a transaction. The essential interface of this class should provide the `Start`, `Commit` and `Abort` operations, and keep track of the accessed objects.

```
type Transaction_Type is (Read_Only, Update);

-- Transaction identifier
package Transactions is
  type Transaction is tagged private;
  procedure Start(T : Transaction;
                  TRX_Type : Transaction_Type);
  procedure Abort(T : Transaction);
  procedure Terminate(T : Transaction);
  function Commit(T : Transaction)
           return Boolean;

private
  type Transaction is tagged
  record
    Data_Set : List_Ref_Transactional_Objects;
end record;
end Transactions;
```

The `Start` procedure initialises the transaction environment. Starting an already active transaction is not allowed, and an exception should be raised.

The `Abort` procedure erases any possible effects of the transaction, but the transaction remains active and is allowed to undertake further execution attempts. Aborting an inactive transaction is not allowed, and an exception should be raised.

The `Terminate` procedure cancels the transaction, leaving the transaction inactive. Terminating an inactive transaction is not allowed, and an exception should be raised.

The last operation provided by this interface is the `Commit` function that validates accessed data and resolves possible conflicts. If the transaction is allowed to commit its updates, then this function will return the `True` value. Commiting an inactive transaction is not allowed, and an exception should be raised.

This class also stores the references to the transactional objects that were accessed in the context of the transaction. These data are required when trying to commit, to validate read and modification locations.

Specific STM implementations will, most likely, require modified operation functionality and additional attributes. For example, some STM algorithms require to know the instant the transaction started, the current status of the transaction [12], or the instant the current execution of the transaction began [20]–[22].

These attributes can be included in extensions of this class, deriving a new class for each implementation, as the following example illustrates.

```
type Transaction_Status is (Active,
                            Preempted,
                            Zombie,
                            Validating,
                            Committed);

-- Transaction identifier
package Transactions_Foo_STM is
  type Transaction_Foo_STM is new Transaction with
       private;
  overriding
  function Commit(T : Transaction_Foo_STM) return
      Boolean;

private
  type Transaction_Foo_STM is new Transaction with
  record
    -- implementation specific elements
    -- some examples below
    Type_of_Accesses : Transaction_Type;
    Time_Started : STM_Time;
    Time_Current_Begin : STM_Time;
    Status : Transaction_Status;
    -- some other relevant fields...
  end record;
end Transactions_Foo_STM;
```

Our current approach to STM assumes that a transaction is not able to abort ongoing concurrent transactions, as this could be very costly for the implementation. However, we intend to evaluate this in future work, and if considered feasible we will also evaluate the usefulness of the Asynchronous Transfer of Control (ATC) feature of the language to detect the request to abort a transaction and execute the roll-back operations.

V. Conclusions

Current and foreseen multi-core architectures have raised performance issues to classical concurrency control based on locks: either coarse-grained locking impairs parallelism or fine-grained locking increases the difficulty to safely develop concurrent software.

Transactions were already considered as a concurrency control mechanism able to maintain the consistency of the systems state in which parallel transactions could abort due to data contention or hardware/software errors. Currently, the same concept is being applied to manage concurrent access to shared data, known as transactional memory.

In this paper, we discuss the use of software transactional memory and we draft a common programming interface to STM in Ada. This interface is independent of a particular STM implementation, so different implementations can address different utilization patterns. It is also the goal of this work to allow the research on contention mechanisms for software transactional memory in real-time systems. Future work will address new contention mechanisms for these systems and evolve the STM support in Ada (*e.g.* using generics or Ada 2012 aspects as wrappers, and reflecting the real-time issues in the Ada proposal).

Acknowledgment

We would like to thank the anonymous reviewers for their valuable comments to improve the paper.

This work was supported by the VIPCORE project, ref. FCOMP-01-0124-FEDER-015006, funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness') and by National Funds (PT) through FCT - Portuguese Foundation for Science and Technology, and the ARTISTDESIGN - Network of Excellence on Embedded Systems Design, grant ref. ICT-FP7-214373.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, Sep. 2005.

[2] B. N. Bershad, "Practical considerations for non-blocking concurrent objects," in *Proceedings of the 13th International Conference on Distributed Computing Systems - ICDCS 1993*, May 1993, pp. 264–273.

[3] P. Tsigas and Y. Zhang, "Non-blocking data sharing in multiprocessor real-time systems," in *Proceedings of the 6th IEEE International Conference on Real-Time Computing Systems and Applications - RTCSA'99*, Dec. 1999, pp. 247–254.

[4] B. B. Brandenburg, J. M. Calandrino, A. Block, H. Leontyev, and J. H. Anderson, "Real-Time Synchronization on Multiprocessors: To Block or Not to Block, to Suspend or Spin?" in *Proceedings of the 14th IEEE Real-Time and Embedded Technology and Applications Symposium - RTAS '08*, Apr. 2008, pp. 342–353.

[5] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th anual International Symposium on Computer Architecture - ISCA '93*, May 1993, pp. 289–300.

[6] N. Shavit and D. Touitou, "Software transactional memory," in *Proceedings of the 14th annual ACM symposium on Principles of distributed computing - PODC '95*, Aug. 1995, pp. 204–213.

[7] A. Dragojevik, P. Felber, V. Gramoli, and R. Guerraoui, "Why STM can be more than a research toy," *Communications of the ACM*, vol. 54, no. 4, pp. 70–77, Apr. 2011.

[8] C. J. Rossbach, O. S. Hofmann, and E. Witchel, "Is transactional programming actually easier?" in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, Jan. 2010, pp. 47–56.

[9] S. F. Fahmy, B. Ravindran, and E. D. Jensen, "On Bounding Response Times under Software Transactional Memory in Distributed Multiprocessor Real-Time Systems," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition - DATE '09*, Apr. 2009, pp. 688–693.

[10] T. Sarni, A. Queudet, and P. Valduriez, "Real-Time Support for Software Transactional Memory," in *Proceedings of the 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications - RTCSA' 2009*, Aug. 2009, pp. 477–485.

[11] M. Schoeberl, F. Brandner, and J. Vitek, "RTTM: Real-Time Transactional Memory," in *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, Mar. 2010, pp. 326–333.

[12] A. Barros and L. M. Pinho, "Software transactional memory as a building block for parallel embedded real-time systems," in *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications - SEAA 2011*, Aug. 2011.

[13] M. Patiño-Martínez, R. Jiménez-Peris, and S. Arévalo, "Integrating groups and transactions: A fault-tolerant extension of Ada," in *Proceedings of the International Conference on Reliable Software Technologies - Ada-Europe '98*, Jun. 1998, pp. 78–89.

[14] M. Patiño-Martínez, R. Jiménez-Peris, and S.Arévalo, "Implementing Transactions using Ada Exceptions: Which Features are Missing?" *Ada Letters*, vol. XXI, no. 3, pp. 64–75, Sep. 2001.

[15] J. Kienzle, R. Jiménez-Peris, A. Romanovsky, and M. Patiño-Martínez, "Transaction Support for Ada," in *Proceedings of the 6th International Conference on Reliable Software Technologies - Ada-Europe 2001*, May 2001, pp. 290–304.

[16] T. Harris and K. Fraser, "Language support for lightweight transactions," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 388–402, Nov. 2003.

[17] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, "Scheduling support for transactional memory contention management," in *Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '10*, Jan. 2010, pp. 79–90.

[18] T. Harris, J. Larus, and R. Rajwar, "Transactional Memory, 2nd edition," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, Dec. 2010.

[19] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Conflict Detection and Validation Strategies for Software Transactional Memory," in *Proceedings of the 20th International Symposium on Distributed Computing - DISC 2006*, Sep. 2006, pp. 179–193.

[20] D. Dice, O. Shalev, and N. Shavit, "Transactional Locking II," in *Proceedings of the 20th International Symposium on Distributed Computing - DISC 2006*, Sep. 2006, pp. 194–208.

[21] T. Riegel, P. Felber, and C. Fetzer, "A Lazy Snapshot Algorithm with Eager Validation," in *Proceedings of the 20th International Symposium on Distributed Computing - DISC 2006*, Sep. 2006, pp. 284–298.

[22] D. Perelman and I. Keidar, "SMV: Selective Multi-Versioning STM," in *Proceedings of the 5th ACM SIGPLAN Workshop on Transactional Computing*, Apr. 2010.