Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

Transactional memory in a dynamic language[☆]

Lukas Renggli[*], Oscar Nierstrasz

Software Composition Group, University of Berne, 3012 Bern, Switzerland

ARTICLE INFO

Article history:
Received 16 November 2007
Received in revised form
30 May 2008
Accepted 3 June 2008

Keywords:
Transactional memory
Concurrent programming
Language constructs and features

ABSTRACT

Concurrency control is mostly based on locks and is therefore notoriously difficult to use. Even though some programming languages provide high-level constructs, these add complexity and potentially hard-to-detect bugs to the application. Transactional memory is an attractive mechanism that does not have the drawbacks of locks, however, the underlying implementation is often difficult to integrate into an existing language. In this paper we show how we have introduced transactional semantics into Smalltalk by using the reflective facilities of the language. Our approach is based on method annotations, incremental parse tree transformations and an optimistic commit protocol. The implementation does not depend on modifications to the virtual machine and therefore can be changed at the language level. We report on a practical case study, benchmarks and further and on-going work.

1. The need for transactions

Most dynamic programming languages have inherently weak support for concurrent programming and synchronization. While such languages relieve the programmer of the burden to allocate and free memory by using advanced garbage collection algorithms, they do not provide similar abstractions to ease concurrent programming [1].

We chose to build our prototype implementation in Smalltalk, because this dynamic language has excellent support for reflection [2] that goes beyond the level of objects and classes and allows us to easily reify aspects of method compilation. Smalltalk, and other dynamic languages such as Ruby, Python and Scheme, provide libraries to work with concurrent processes, but only provide little help to control and synchronize the access of shared data. Smalltalk-80 [3] offers semaphores as the only mean for synchronizing processes and guaranteeing mutual exclusion. The ANSI standard of Smalltalk [4] does not refer to synchronization at all.

Only a few current Smalltalk implementations provide more sophisticated synchronization support. VisualWorks Smalltalk provides a reentrant lock that allows the same process to reenter the lock multiple times. Other processes are blocked until the owning process leaves the critical section. Unfortunately lock-based approaches have their drawbacks and are notoriously difficult to use [5]:

Deadlocks: If there are cyclic dependencies between resources and processes, applications may deadlock. This problem can be avoided by acquiring resources in a fixed order, however, in practice this is often difficult to achieve.

Starvation: A process that never leaves a critical section, due to a bug in the software or an unforeseen error, will continue to hold the lock forever. Other processes that would like to enter the critical section starve.

[☆] This work is based on an earlier work: Transactional Memory for Smalltalk, in Proceedings of the 2007 International Conference on Dynamic Languages (ESUG/ICDL 2007), http://doi.acm.org/10.1145/1352678.1352692 © ACM, 2007.

[*] Corresponding author.
E-mail addresses: renggli@iam.unibe.ch (L. Renggli), oscar@iam.unibe.ch (O. Nierstrasz).
URL: http://www.lukas-renggli.ch (L. Renggli).

1477-8424/$ - see front matter © 2008 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2008.06.001

Priority inversion: Usually schedulers guarantee that processes receive CPU time according to their priority. However, if a low priority thread is within a critical section when a high priority process would like to enter, the high priority thread must wait.

Squeak Smalltalk [6] includes an implementation of monitors [7], a common approach to synchronize the use of shared data among different processes. In contrast to mutual exclusion with reentrant locks, with monitors, a process can wait inside its critical section for other resources while temporarily releasing the monitor. Although this avoids deadlock situations, the use of monitors is difficult and often requires additional code to specify guard conditions [8]. Moreover, if the process is preempted while holding the monitor, everybody else is blocked. Beginners are often overwhelmed by the complexity of using monitors as Squeak does not offer method synchronization as found in Java.

Transactional memory [9,10] provides a convenient way to access shared memory by concurrent processes, without the pitfalls of locks and the complexity of monitors. Transactional memory allows developers to declare that certain parts of the code should run atomically: this means the code is either executed as a whole or has no effect. Moreover transactions run in *isolation*, which means they do not affect and are not affected by changes going on in the system at the same time. Upon *commit* the changes of a transaction are applied atomically and become visible to other processes. Optimistic transactions do not lock anything, but rather conflicts are detected upon commit and either lead to an *abort* or *retry* of the transaction.

Most relational and object databases available in Smalltalk provide database transactions following the ACID properties: Atomicity, consistency, isolation, and durability. However, they all provide this functionality for persistent objects only, not as a general construct for concurrent programming. These implementations often rely on external implementations of transactional semantics. GemStone Smalltalk [11] is a commercially available object database, that directly runs Smalltalk code. As such, GemStone provides transactional semantics at the virtual machine (VM) level. Guerraoui et al. [12] developed GARF, a Smalltalk framework for distributed shared object environments. Their focus is not on local concurrency control, but on distributed object synchronization and message passing. They state that "*A transactional mechanism should, however, be integrated within group communication to support multi-server request atomicity.*" [13]. Jean-Pierre Briot proposed Actalk [14], an environment where Actors communicate concurrently with asynchronous message passing. The use of an Actor model is intrusive. It implies a shift of the programming paradigm to one where there is no global state and therefore no safety issues.

In this paper we present an implementation of transactions in Squeak based on parse-tree transformation. In this way most code is free of concurrency annotations, and transactional code is automatically generated only in the contexts where it is actually needed.

The specific contributions of this paper are:

- The implementation of transactional semantics in a dynamic language, using the reflective capabilities of the language without any changes to the low-level VM implementation.
- A mechanism to specify context-dependent code using method annotations, for example to intercept the evaluation of primitive methods.
- Incremental, on-the-fly parse tree transformation for different execution contexts.
- Efficient, context-dependent code execution using the execution mechanisms of a standard VM.

This article extends our previous work [15] as follows: (1) we describe how our approach applies to dynamically typed languages in general, not just the implementation language of our prototype, (2) we devote a section to related work, (3) we explain how nested transactions are handled, and (4) we enhance the validation section with results of additional benchmarks.

Section 2 presents some basic usage patterns of our implementation. Section 3 shows the implementation of transactions in Squeak without modifying the underlying VM. Section 4 validates our approach by running a collection of benchmarks and by applying the concept to a real world application. Section 5 compares our approach with other approaches that have been taken to integrate transactional memory in programming languages. Section 6 concludes this article with some remarks about ongoing and future work.

2. Programming with transactions

Transactions offer an intuitively simple mechanism for synchronization concurrent actions. They do not require users to declare specific locks or guard conditions that have to be fulfilled. Moreover transactions can be used without prior knowledge of the specific objects that might be modified. Transactions are global, yet multiple transactions can run in parallel. The commit protocol checks for conflicts and makes the changes visible to other processes atomically.

On the left side of Fig. 1 we see the traditional way of using a semaphore to ensure mutual exclusion on a tree data structure. The key problem is that *all* read and write accesses to the tree must be guarded using the same lock to guarantee safety. A thread-safe tree must be fully protected in all of its public methods. Furthermore, we cannot easily have a second, unprotected interface to the same tree for use in a single-threaded context.

On the right side of Fig. 1 we present the code that is needed to safely access the collection using a transaction: the write access is put into a block that tells the Smalltalk environment to execute its body within a transaction. The read access can happen without further concurrency control. As long as all write accesses occur within the context of a transaction, read accesses are guaranteed to be safe. The optimistic commit protocol of the transaction guarantees safety by (i) ensuring