Transactional Memory in a Dynamic Language ⋆

Lukas Renggli , Oscar Nierstrasz

Software Composition Group
University of Berne, Switzerland

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.

Keywords. Transactional Memory, Concurrent Programming, Language Constructs and Features

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].

⋆ 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

Email addresses: renggli@iam.unibe.ch (Lukas Renggli), oscar@iam.unibe.ch (Oscar Nierstrasz).

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.

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 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.

Outline. 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.

```
tree := BTree new.
lock := Semaphore forMutualExclusion.

" writing "
lock critical: [ tree at: #a put: 1 ].

" reading "
lock critical: [ tree at: #a ].
```

```
tree := BTree new.


" writing "
[ tree at: #a put: 1 ] atomic.

" reading "
tree at: #a.
```

Fig. 1. Lock-based vs. Transactional accesses of a shared data structure.

On the left side of Figure 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 Figure 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 that no write conflicts have occurred with respect to the previous saved state, and (ii) atomically updating the global object state.

To make the code using transactions as simple as possible we provide two methods for running code as part of a transaction. These methods are exten-

sions to the standard Smalltalk library, and do not affect the language syntax or runtime.

- sending #atomic causes the receiving block closure to run as a new transaction. Upon termination of the block, any changes are committed atomically. If a conflict is detected, all modifications are cancelled and a commit conflict exception is raised.
- sending #atomicIfConflict: causes the receiving block to run as a new transaction. Instead of raising an exception if a conflict occurs, the block argument is evaluated. This enables developers to take a specific action, such as retrying the transaction or exploring the conflicting changes.

Further convenience methods can easily be built out of these two methods, for example a method to retry a transaction up to fixed number of times, or only to enter a transaction if a certain condition holds.

3 Inside transactions

We introduce transactions to Smalltalk without modifying the underlying Virtual Machine (VM). Our approach is based on earlier proposals in which source code is automatically and transparently transformed to access optimistic transactional object memory, rather than directly accessing objects [16,17]. The key advantage of this approach is that most source code can be written without embedding any explicit concurrency control statements. Transactional code is automatically generated where it is needed. Furthermore, in contrast to the earlier approaches, we generate the needed transactional code dynamically where and when it is needed, and caching the generating code for future invocations.

In a nutshell, our approach works as follows:

- Every method in the system may be compiled to two versions: one to be executed in the normal execution context, and the other within a transactional context. Contrary to the other approaches we do this incrementally and on the fly using a compiler extension.
- State access in transactional methods is automatically transformed to use an indirection through the transaction context.
- We use method annotations to control the automatic code transformation or to provide different code. Unlike other implementations of transactional memory, we take into account the use of primitives, exception handling and file-system access by providing alternative code to be used in a transactional context.

- When entering a transactional context we record the transaction (an object) in the current process (also an object).
- During a transaction, snapshots of all touched objects are taken. Each snapshot consists of two copies of the original object: one that reflects the initial state and one that is altered during the transaction. For efficiency reasons immutable objects are excluded from snapshots. This strategy gives us repeatable reads, however since snapshots are made at different times a transaction might read inconsistent state and will abort when trying to commit the changes.
- Upon commit we check for conflicts by atomically comparing the state of the object at the beginning of the transaction to the current version in memory. If no conflict is detected, the changes are committed. In case of a conflict the system is left in the state as it was before the transaction and an exception is raised that provides information for further reflection, namely all the changes, the conflicting changes and the transaction object itself.

The key novelties of our approach lie in the use of annotations and reflection to lazily generate the transactional versions of methods, and the ability to provide alternative code to use in place of primitives during transactions.

In the following two sections we describe (1) the compilation to transactional code, and (2) the implementation of the transactional object model.

3.1 Compiling to transactional code

We transform methods by changing read and write accesses to make use of transactional object memory. Methods are transformed using the behavioral reflection framework Geppetto [18] which is based on sub-method reflection [19], allowing us to declaratively reify and transform an abstract syntax tree (AST) before compiling to byte-code.

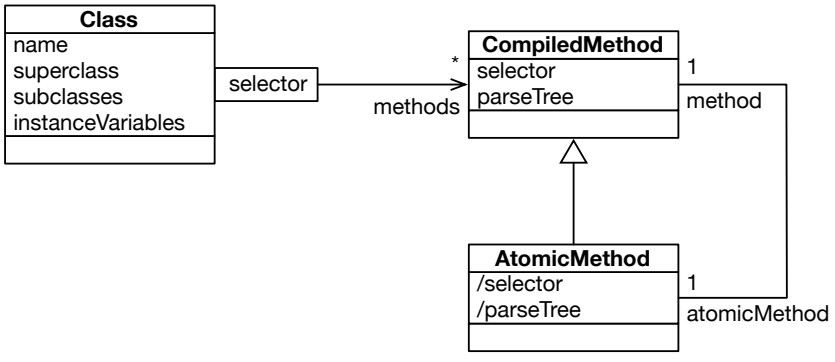

Fig. 2. Static Compilation Model

Whenever the source code of a method is compiled, as seen in Figure 2, our compiler plugin creates an additional compiled method that implements the

behaviour to be used within the context of a transaction. The following basic transformations are performed:

(1) Reading from instance variables and global variables is transformed to send the messages #atomicInstVarAt: or #atomicValue respectively. This allows us to implement these two messages to read the current value from within a transactional context instead of directly accessing the variables within the receiving object.
(2) Writing to instance and global variables is transformed to send the messages #atomicInstVarAt:put: or #atomicValue: respectively. Again this allows us to intercept state access and handle it from within the current transaction.
(3) Sending a message from inside a transactional method will actually send a different message name, namely we prepend #__atomic__ to the original selector name.

```
BTree≫at: aKey put: anObject
    | leaf |
    leaf := root leafForKey: aKey.

    leaf insertKey: aKey value: anObject.
    root := leaf root.
    ^ anObject
```

```
BTree≫__atomic__at: aKey put: anObject
    | leaf |
    leaf := (self atomicInstVarAt: 1)
        __atomic__leafForKey: aKey.
    leaf __atomic__insertKey: aKey value: anObject.
    self atomicInstVarAt: 1 put: leaf __atomic__root.
    ^ anObject
```

Fig. 3. Original vs. transformed code.

Having applied these three transformations to the code, the two compiled methods are stored in the method dictionary of their owning class. To tell the two methods apart, the atomic version of the method has #__atomic__ prepended to its name. These methods are hidden, and are only called from generated code within an atomic context. Transactional methods are filtered from the code editors, so they are not visible to the developer and development tools but only to the VM. On the left side of Figure 3 we present the code of a method as the developer implemented it, whereas on the right side we show the same method as it is compiled for the atomic context.

A transaction is created by sending the message #atomic to a block containing normal (non-transactional) Smalltalk code. The code within such a block is statically transformed to evaluate within a transactional context. We have seen an introductory example for such a call in Figure 1. Methods that send #atomic are special, because the code outside this block is compiled normally, whereas we apply the transformations as described above to the inside of the block closure.

Squeak includes a few primitive methods that access and modify state. The most prominent of these are #at: and #at:put: to access the elements of variable-sized objects. Moreover there are also some key collection and stream methods that are implemented within the VM for efficiency. As primitive op-

In presence of annotation	do transform	take source code from
(no annotation, default)	yes	method body
<atomic:>	yes	argument
<atomicDoNotTransform>	no	method body
<atomicDoNotTransform:>	no	argument

Fig. 4. Method annotations are used to control how the compiler transforms source code for the transactional context.

erations are written in C and statically compiled into the VM, we cannot use Geppetto to modify state-access. The only possibility to reify these methods is to replace them with non-primitive methods.

We make use of annotations to further control the way in which transactional code may be generated. Figure 4 summarizes the effect of the following annotations:

<atomicDoNotTransform> avoids doing any code transformation. This means the normal and the transactional method will be the same, so no transformation is needed. In the current implementation this is mostly used for exception handing, as this code should continue to work through the boundaries of transactions. It is also used for infrastructural code, as shown in the example below.

```
Transaction≫signalConflict
    "Signal a conflict within a transaction."

    <atomicDoNotTransform>
    CommitConflictException new
        transaction: self;
        signal
```

<atomic:> uses the method identified as its argument as the source for the code transformation. We use this for primitives that are implemented for efficiency reasons only. For example the method #replaceFrom:to:with:startingAt: in the class Array calls the primitive 105 and is used to copy elements from one collection to another one. With the method annotation we tell the compiler that it should instead transform and install the method #atomicReplaceFrom:-

to:with:startingAt: which has the same behavior but is implemented in Smalltalk and can therefore be transformed automatically.

```
Array≫replaceFrom: start to: stop with: repl startingAt: repStart
    "Primitive. This destructively replaces elements from start to stop in the
    receiver starting at index, repStart, in the collection, repl."

    <primitive: 105>
    <atomic: #atomicReplaceFrom:to:with:startingAt:>
    super replaceFrom: start to: stop with: repl startingAt: repStart

Array≫atomicReplaceFrom: start to: stop with: repl startingAt: repStart
    | index repOff |
    repOff := repStart - start.
    index := start - 1.
    [ (index := index + 1) <= stop ]
        whileTrue: [ self at: index put: (repl at: repOff + index) ]
```

<atomicDoNotTransform:> uses the method identified as its argument as untransformed atomic code. We use this mainly in infrastructural code to dispatch primitive requests that access state to the working copy of the receiver. For example indexed slot access is handled through primitives in Squeak. The method #at: in the class Object calls the primitive 60 to fetch the contents of an indexed element. The method annotation tells the compiler to use #atomicAt: instead. This method delegates the request to the current working copy of the object.

```
Object≫at: index
    "Primitive. Assumes receiver has indexed slots. Answer the value of an indexable
    element in the receiver. Fail if the argument index is not an Integer or is out
    of bounds."

    <primitive: 60>
    <atomicDoNotTransform: #atomicAt:>
    self primitiveFail

Object≫atomicAt: index
    ^ self workingCopy at: index'
```

Compiling all the methods of the system is costly both in time and memory. Most methods available in the system are never called from within a transactional context and therefore do not need to be translated. The dynamic nature of Smalltalk makes it difficult to determine statically the required set of transactional methods, however it allows us to compile methods lazily when they are about to be executed. This produces a slowdown the first time a method is executed within a transactional context, but subsequent invocations are dis-

patched using the normal mechanisms of the VM and therefore run at full speed.

Most transactional systems prohibit I/O or other side effects that cannot be undone (system calls, file system access) during transactions [20,21]. Our approach allows replacement code to be specified for use within a transactional context. For example, when deleting a file the action is recorded with a custom change object and atomically applied together with the other changes upon successfully committing the transaction:

```
FileDirectory≫deleteFile: aString
    <primitive: 'primitiveFileDelete' module: 'FilePlugin'>
    <atomicDoNotTransform: #atomicDeleteFile:>

FileDirectory≫atomicDeleteFile: aString
    Processor activeProcess currentTransaction
        addChange: (CustomChange onApply: [ self deleteFile: aString ])
```

Our model also allows exceptions to be thrown and handled inside the transaction boundaries. An exception that leaves the boundaries of a transaction causes that transaction to abort and the exception to be re-raised in the non-transactional context.

3.2 Transactions at runtime

When entering a transaction we create a new transaction object and store it in an instance variable of the current process, as depicted in Figure 5. When leaving a transaction we set the current transaction reference back to nil. In this way we can efficiently determine the current transaction from anywhere in our application. Moreover we capture an escape continuation upon entry, to be able to abort the current transaction by doing a non-local jump to the calling context.

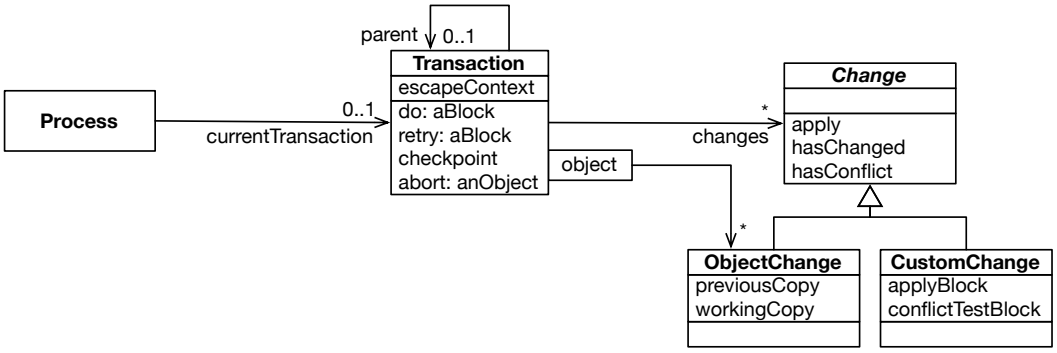

Fig. 5. Dynamic model of transactions at runtime

After having entered a transactional context, all the executed code is in its transformed form. This means that state access goes through special accessor methods and all message sends are redirected to their transactional counterparts. The execution of transactional code consequently works the same as normal code execution: it shares the same object memory but it uses a different access strategy to access state.

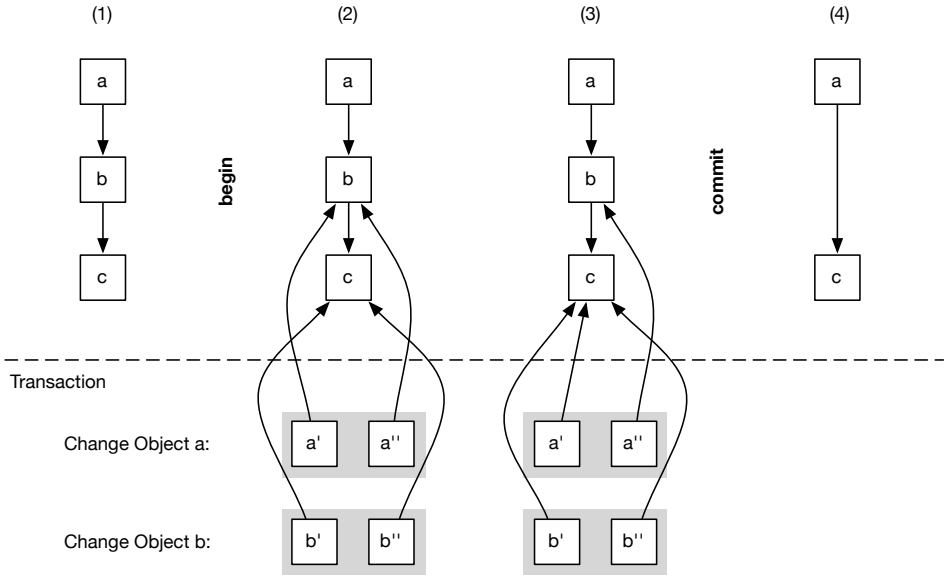

Fig. 6. Removing node b from a linked list. (1) The list before the transaction starts. (2) As the list is traversed, change objects for a and b are created, each containing a working copy a' and b' and a previous copy a'' and b''. (3) As b is removed from the list, the reference from a' to b is changed to point to c. At this point the change is not visible from outside the transaction, the original object a is left unchanged. (4) Upon commit, the previous copies a'' and b'' are compared with a and b to detect conflicts. The changes to a are copied from a' to a. As b isn't referenced anymore, it will eventually be garbage collected.

We adopt a conventional optimistic transaction protocol [8]. Whenever an object is touched within the context of a transaction for the first time (read or written), the transaction instantiates a new change object ObjectChange. This change object contains references to two copies of the object. The previousCopy contains an immutable copy for detecting conflicts. The workingCopy is a mutable copy of the object being used during the transaction. The change object knows if it has a conflict (the original object is not the same as the previous copy) and if it has changed (original object is not the same as the working copy). Both the previousCopy and the workingCopy are shallow copies, this means that the copies reference the same values as the original. See Figure 6 for a practical example clarifying the steps above.

We also provide a custom change object CustomChange that is used to record irreversible actions that should only be applied during the atomic commit phase if there are conflicts. We have seen the use of such a custom change in

Section 3.1, where we presented a possible solution for file-deletion within a transactional context.

At the end of the transaction we have to acquire a form of "global lock" on the object memory to be able to check for conflicts and commit the changes. We use #valueUnpreemptively implemented on block closures to ensure that no other process is running at the same time. As a first step we check if any of the changes we gathered during the transaction has a conflict and raise an exception if this is the case. Otherwise we copy the changes from the working copies to the original objects. The time required to hold the lock and to validate and apply the changes linearly depends on the number of objects involved in the transaction.

These are some important properties of our transactional model [22]:

Repeatable read. Reads are repeatable. Since data that is read within a transaction is copied, repeated reads from within a transaction are consistent. Changes outside the transaction are not visible after a first read.

Optimistic write. Our transactional memory writes optimistically [23]. The transaction boundaries are controlled by working on copies of the objects.

Lazy version management. We create copies of objects that are read and written within transactions. This requires an extra redirection for accessing the state and a considerable amount of memory and processing time for copying the involved objects. Aborting a transaction is cheap as no state has to be restored.

Lazy conflict detection. Assuming that conflicts are rare, conflicts are checked before committing data. This check happens atomically together with the commit.

Lazy conflict resolution. Conflicts are resolved by dropping (or retrying) the transaction that produces the conflict when committing. The changes are eventually collected by the garbage collector.

Real object references. Objects do not change their representation within a transaction — they are neither wrapped nor replaced with different objects. Identity comparisons work as one would expect from non-transactional code. The only thing that changes is their behavior of objects, namely access to state is reified trough the current transaction context (see Figure 6).

Nested transactions. Every transaction references its parent transaction or nil, if this is the outermost transaction. Upon commit of a nested transaction, the changes are tested for conflicts and then added to the change-list of the outer transaction. This enables opacity of library calls, even if these libraries themselves use transactional semantics.

4 Validation

First we assess the cost of transactions by means of benchmarks that compare the running time of actions performed with and without transactions. Then we compare the cost of thread-safety realized with semaphores to that of our implementation with transactional memory.

4.1 Micro benchmarks

We performed several micro benchmarks to establish the runtime cost of using our implementation of transactional memory for Smalltalk. Figure 7 shows the times and ratios of performing basic actions, such as invoking a method or accessing state. t_1 is the time required to perform the action 10^7 times outside a transactional context, and t_2 is the time required to perform the same action within a transactional context. The benchmarks were performed on an Apple MacBook Pro, 2.16 GHz Intel Core Duo in Squeak 3.9. The required transactional methods were compiled in advance.

Operation	t_1	t_2	ratio
Activation	2.75	85.27	31.03
Method invocation	1.98	1.98	1.00
Special method invocation	1.14	2.00	1.75
Instance variable read	1.03	20.72	20.08
Instance variable write	1.13	21.04	18.60
Indexed variable read	1.11	19.92	17.93
Indexed variable write	1.21	20.22	16.75
Global variable read	1.03	20.89	20.25
Global variable write	1.15	21.72	18.92

Fig. 7. t_1: time in seconds for 10^7 runs in a non-transactional context, t_2: time in seconds for 10^7 runs in a transactional context, ratio: t_2/t_1, the penalty when used in a transactional context.

The activation time is the time required to enter a transaction as compared to the time required to evaluate a block closure. The *ratio* indicates that entering a transaction is 31 times slower than entering a block closure. This results from the fact that entering a transactions requires several objects to be instantiated to track the changes of the transaction. Moreover the transaction is recorded

in the current process and an escape context must be captured to be able to abort a running transaction.

Normal method invocation does not show any speed penalty. In all the benchmarks we assume that the transactional methods are already compiled. For some common selectors, such as #+, #*, #=, #size, #at:put:, #new, #class, *etc.*, Squeak uses special byte codes to make the invocation about twice as fast as a normal message send. In a transactional context these byte codes cannot be used anymore and have to be replaced by normal message sends, resulting in a penalty for special method invocations.

State access within a transactional context is fairly expensive. For instance, indexed and global variable reads and writes produce very similar results: in the current implementation these are about 20 times slower than their non-transactional counterparts. As we have seen in Section 3.2, accessing state of an object requires to lookup the current transaction, the change object and to dispatch the state access to its working copy. This whole procedure involves several message sends that cannot be easily optimized in Smalltalk. Further improvements are possible by writing primitives (or introducing new byte-codes) that can more efficiently dispatch that kind of request.

Here we have been comparing the cost of thread-safe actions to unsafe actions. A fairer comparison would be that between thread-safe actions implemented with semaphores and thread-safe transactions. We discuss this in the following section.

4.2 Real world example using transactional memory

We applied our transactional model to Pier, a web-based content management system [24, Chapter 3]. Pier uses a tightly-connected graph of objects to represent pages and their content. Edit operations on pages use the Command design pattern. Many operations, such as adding or removing a page, require the system to walk through the whole object graph to invalidate links. We have two lock-based approaches for Pier: one acquiring a global lock while executing the command, and another one using a more fine-grained locking on the level of individual pages. While the first approach is much simpler in implementation, it potentially holds the lock for a longer time and blocks all other commands, even though edit operations rarely conflict with each other.

Before executing a command Pier checks for conflicts on the current page, to avoid changes of other users from being accidently overridden. It does not check for conflicts that could be caused by the need to update links in other parts of the object model. Pier normally does not lock read operations, such as browsing the web site, as they are very common and would introduce a major

bottleneck. In rare cases users could therefore encounter an inconsistent state of a particular page.

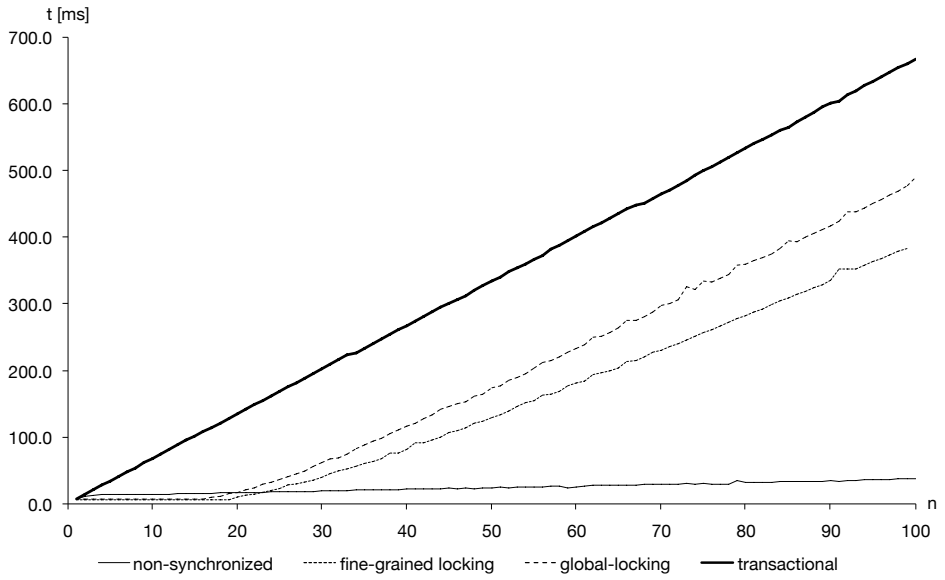

Fig. 8. Average execution time for non-synchronized, fine-grained locking, global-locking and transactional execution time t to complete $n = 1..100$ concurrent edit operations in Pier.

To assess the effectiveness of transactional memory for Smalltalk, we remove the global lock in Pier and wrap the execution of the command within a transactional context. This means that edit commands can now be evaluated concurrently while still ensuring consistency. Moreover we could remove the manual checks for conflicts as these are now detected and handled by the transaction in a complete manner. Page views are now guaranteed to see a consistent state of the web site, as all the changes are applied atomically through a transaction.

Figure 8 shows the average execution time of an edit command that changes the contents of a single page. Using a script we simulated $n = 1..100$ concurrent edit operations on different pages, so no conflicts could occur. Interestingly the overhead is consistently just over 100 ms for transactions over locks. The transactions are short, and involve only few objects and very little state access. Memory requirements are moderate: the edit operation touches 39 objects, for each of which the transaction requires 2 copies to track changes. In this particular use case, a single transaction consumes 2556 bytes of additional memory.

We believe that the transactional approach would be considerably faster than the lock-based one, if the Squeak VM would exploit multiple CPUs to process concurrent requests.

5 Related work

The use of software transactional memory is simple and straightforward. Implementors of transactional memory however have chosen a variety of different strategies, each with its own advantages and disadvantages:

Harris et al. [5] propose an implementation of software-based transactions in Java. They make use of a modified compiler and VM. A secondary method table is added to every class, holding the transformed code to be used within a transaction. Similar to our approach, transactional code is only generated on demand. The implementation keeps ownership tables and version numbers of objects, with the price of changing the internal memory management of Java. Conflicts are detected before a non-blocking commit phase. The implementation does not allow native methods to be executed within a transactional context, with the exception of a few special cases handled explicitly.

Hindman et al. [16] implemented atomicity in Java that neither modifies the compiler nor the VM. Their implementation is based on a source-to-source translation that happens as a pre-compilation step before the normal Java compilation. All instances are extended with an extra field holding the current owner of the object. Read and write accesses are rewritten to check the ownership and to acquire an object lock if necessary. Special measures are taken to avoid deadlocks. The drawback of this approach is that all sources must be available at compile time and that certain classes cannot be easily changed as their definition is assumed by the VM.

Scheme 48 [25] provides *proposals* for optimistic transactional semantics. During transactions a thread references a log of *provisional* read and write accesses. As there is no automatic code transformation, developers use special *provisional* accessor primitives to read and write state during a transaction. Writes are delayed until all read and write accesses are checked for consistency upon commit. Calling library code during a transaction is not possible. Kimball et al. [26] avoid the shortcomings of this approach by swapping out the bytecode dispatch table during the transaction. Write access is logged and directly performed on the involved objects. If the thread is preempted, the transaction is aborted, changes are undone and the scheduler is advised to give the transaction a longer uninterrupted time slice the next time. This approach is efficient, however it only works well for relatively short transactions and systems that use one operating system thread only.

6 Conclusion and future work

Smalltalk VMs traditionally offer poor support for concurrency control. Existing Smalltalk dialects provide only lock-based concurrency control, with the exception of GemStone Smalltalk, which provides transactions only for database code. In this paper we have presented an implementation of optimistic transactions for Squeak Smalltalk without modifying the underlying VM.

Our prototype implementation demonstrates that any Smalltalk can profit from having a transactional model. The implementation can be potentially ported to any of today's available Smalltalk platforms, as it is purely based on parse tree transformation of source code. The fact that the whole implementation is written in Smalltalk makes it an ideal platform to experiment with different transaction policies and implementation strategies. Changes to the transactional runtime system and transactional code can be applied and compiled on the fly, so there is no need to restart or rebuild the system.

Our approach works well with external libraries. New code that is loaded into the Smalltalk environment is transformed lazily within the context of a transaction. Primitive methods, filesystem I/O and exceptions work well together with transactions, as special transformation rules can be specified using method annotations. Contrary to other approaches our implementation integrates well with garbage collection, as the transactions are fully implemented in the object system of Smalltalk.

State access within a transaction is about 20 times slower than usual, which is a big penalty to pay. The integration of transactions with the object model at the VM level would certainly lead to much better performance, however we would also lose the flexibility to be able to quickly change the semantics of the transactional mechanisms. Code not using transactions continues to work exactly as before. The traditional mechanisms used for concurrency control can be even mixed with transactions.

As future work we would like to investigate how to further improve the speed of our model. We would like to investigate other areas of applicability, such as atomic loading of source code. In Smalltalk this is traditionally done in an incremental manner and poses certain problems, for example when the application is supposed to continue running while loading.

Furthermore we would like to see how to apply our approach to other dynamic programming languages, such as Python or Ruby. We expect the implementation in Ruby to be much more difficult than in Smalltalk, as this language has major parts of its library implemented in C. The lack of support to transform source code using high-level AST representations can be avoided in Python

by using PyPy, an implementation of Python in itself [27]. We expect it to be possible to introduce transactional semantics using the PyPy translator toolchain in a similar way, as we did for Smalltalk.

Our approach to implementing optimistic transactions in Smalltalk can be seen as a special case of *context-oriented programming* [28], a programming paradigm that supports context-dependent behaviour. Transactional behaviour is automatically dispatched whenever we enter a transactional context. We believe that this approach can be extended more generally to support other forms of context-dependent concurrency control: instead of littering code with explicit calls to specific concurrency mechanisms, one should be able to simply annotate code with the concurrency properties one would like to ensure, and depending on the runtime context the appropriate behaviour will be automatically selected. We also intend to explore more efficient approaches to implementing contextual behaviour, in particular the use of *scoped reflection* [29] to control the temporal and spatial context in which reflective behaviour is active.

Acknowledgments

We thank Tudor Gîrba for his careful review of a draft of this paper.

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Analyzing, capturing and taming software change" (SNF Project No. 200020-113342, Oct. 2006 - Sept. 2008).

References

[1] D. Grossman, The transactional memory / garbage collection analogy, SIGPLAN Notices 42 (10) (2007) 695–706.

[2] F. Rivard, Smalltalk: a reflective language, in: Proceedings of REFLECTION '96, 1996, pp. 21–38.

[3] A. Goldberg, D. Robson, Smalltalk 80: the Language and its Implementation, Addison Wesley, Reading, Mass., 1983.

[4] American National Standards Institute, Inc., Draft American National Standard for Information Systems — Programming Languages — Smalltalk, American National Standards Institute, 1997.

[5] T. Harris, K. Fraser, Language support for lightweight transactions, in: Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, New York, NY, USA, 2003, pp. 388–402.

[6] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, A. Kay, Back to the future: The story of Squeak, a practical Smalltalk written in itself, in: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97), ACM Press, 1997, pp. 318–326.

[7] P. B. Hansen, Monitors and Concurrent Pascal: a personal history, ACM Press, New York, NY, USA, 1996.

[8] D. Lea, Concurrent Programming in Java, Second Edition: Design principles and Patterns, 2nd Edition, The Java Series, Addison Wesley, 1999.

[9] M. P. Herlihy, Wait-free synchronization, ACM Transactions on Programming Languages and Systems 13 (1) (1991) 124–149.

[10] M. P. Herlihy, J. E. B. Moss, Transactional memory: Architectural support for lock-free data structures, in: Proceedings of the 20. Annual International Symposium on Computer Architecture, 1993, pp. 289–300.

[11] P. Butterworth, A. Otis, J. Stein, The GemStone object database management system, Commun. ACM 34 (10) (1991) 64–77.

[12] R. Guerraoui, B. Garbinato, K. R. Mazouni, The GARF library of DSM consistency models, in: EW 6: Proceedings of the 6th workshop on ACM SIGOPS European workshop, ACM Press, New York, NY, USA, 1994, pp. 51–56.

[13] R. Guerraoui, P. Felber, B. Garbinato, K. Mazouni, System support for object groups, in: OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, ACM Press, New York, NY, USA, 1998, pp. 244–258.

[14] J.-P. Briot, Actalk: A testbed for classifying and designing actor languages in the Smalltalk-80 environment, in: S. Cook (Ed.), Proceedings ECOOP '89, Cambridge University Press, Nottingham, 1989, pp. 109–129.

[15] L. Renggli, O. Nierstrasz, Transactional memory for Smalltalk, in: Proceedings of the 2007 International Conference on Dynamic Languages (ICDL 2007), ACM Digital Library, 2007, pp. 207–221.

[16] B. Hindman, D. Grossman, Atomicity via source-to-source translation, in: MSPC '06: Proceedings of the 2006 workshop on Memory system performance and correctness, ACM Press, New York, NY, USA, 2006, pp. 82–91.

[17] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, T. Shpeisman, Compiler and runtime support for efficient software transactional memory, in: PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, ACM Press, New York, NY, USA, 2006, pp. 26–37.

[18] D. Röthlisberger, M. Denker, É. Tanter, Unanticipated partial behavioral reflection: Adapting applications at runtime, Journal of Computer Languages, Systems and Structures 34 (2-3) (2008) 46–65.

[19] M. Denker, S. Ducasse, A. Lienhard, P. Marschall, Sub-method reflection, in: Proceedings of TOOLS Europe 2007, Vol. 6, ETH, 2007, pp. 231–251.

[20] S. Lie, Hardware support for unbounded transactional memory, Master's thesis, Massachusetts Institute of Technology (May 2004).

[21] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, LogTM: Log-based transactional memory, in: Proceedings of the 12th International Symposium on High-Performance Computer Architecture, IEEE Computer Society, 2006, pp. 254–265.

[22] J. Bobba, K. E. Moore, L. Yen, H. Volos, M. D. Hill, M. M. Swift, D. A. Wood, Performance pathologies in hardware transactional memory, in: Proceedings of the 34rd Annual International Symposium on Computer Architecture, International Symposium on Computer Architecture, 2007, pp. 81–91.

[23] H.-T. Kung, J. T. Robinson, On optimistic methods for concurrency control, ACM TODS 6 (2) (1981) 213–226.

[24] L. Renggli, Magritte — meta-described web application development, Master's thesis, University of Bern (Jun. 2006).

[25] R. Kelsey, J. Rees, M. Sperber, The incomplete Scheme 48 reference manual for release 1.8 (Feb. 2008).
URL `http://s48.org/`

[26] A. Kimball, D. Grossman, Software transactions meet first-class continuations, in: The 8th Annual Workshop on Scheme and Functional Programming, ACM SIGPLAN, 2007.

[27] PyPy, an implementation of Python in Python.
URL `http://codespeak.net/pypy`

[28] P. Costanza, R. Hirschfeld, Language constructs for context-oriented programming: An overview of ContextL, in: Proceedings of the Dynamic Languages Symposium (DLS) '05, co-organized with OOPSLA'05, ACM, New York, NY, USA, 2005, pp. 1–10.

[29] O. Nierstrasz, M. Denker, T. Gîrba, A. Lienhard, Analyzing, capturing and taming software change, in: Proceedings of the Workshop on Revival of Dynamic Languages (co-located with ECOOP'06), 2006.