Technical Report

UCAM-CL-TR-572
ISSN 1476-2986

Number 572

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Design choices for language-based transactions

Tim Harris

August 2003

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

Series editor: Markus Kuhn

ISSN 1476-2986

Design choices for language-based transactions

Tim Harris
University of Cambridge Computer Laboratory
15 JJ Thomson Avenue
Cambridge, UK, CB3 0FD
tim.harris@cl.cam.ac.uk

ABSTRACT

This report discusses two design choices which arose in our recent work on introducing a new `atomic` keyword as an extension to the Java programming language. We discuss the extent to which programs using atomic blocks should be provided with an explicit 'abort' operation to roll-back the effects of the current block. We also discuss mechanisms for supporting blocks that perform I/O operations or external database transactions.

1. INTRODUCTION

In recent work we have been investigating the use of Software Transactional Memory (STM) as a mechanism for implementing language-level concurrency control features [6]. In our system, developed as an extension to the Java programming language, we introduce a new keyword `atomic` which allows a group of statements to execute atomically with respect to the operation of other threads. We also allow `atomic` statements to be guarded by boolean conditions, with execution blocking until the condition is satisfied. For example, the core of a single-cell shared buffer could be implemented as:

```
class Buffer {
  private boolean full;
  private int value;

  public void put(int new_value) {
    atomic (!full) {
      full = true;
      value = new_value;
    }
  }

  public int get() {
    atomic (full) {
      full = false;
      return value;
    }
  }
}
```

The `atomic` construct build on designs for Conditional Critical Regions (CCRs) [8] and on the concurrency control features of languages such as DP [3], Edison [4], Lynx [10] and Argus [9].

Our intended semantics are that an `atomic` block without a guarding condition should behave equivalently to executing the block in single-threaded mode. Executing a block with a guard condition should be equivalent to waiting until the condition is known to be true and then executing the block in single-threaded mode. In either case, the order in which different blocks enter single-threaded mode will define a serialisable order for their execution.

As you may realise from this sketch, these informal definitions leave many design choices for exactly how this `atomic` construct should behave. Our OOPSLA paper describes the decisions taken in our prototype implementation [6]; the purpose of this paper is to elaborate on some of the more contentious points which have frequently come up in discussion.

In Section 2 we present a brief overview of our system, describing the interface exposed by the STM and showing how we implement the `atomic` keyword over it. In Section 3 we discuss the extent to which transactional semantics should be exposed to application programmers. In Section 4 we discuss the integration of I/O operations (and, more generally, operations with side effects). Section 5 concludes.

2. SYSTEM OVERVIEW

Our current implementation of the `atomic` is built in two layers:

- The lower level is a word-based software transactional memory (STM). This allows groups of memory accesses to be performed within transactions which commit atomically. The STM is implemented in C within the Java Virtual Machine. The heap formats used by the JVM remain unchanged.
- The higher level maps the `atomic` keyword onto a series of STM operations. For example, entering an atomic block requires a new transaction to be started, and accesses to shared fields within an `atomic` block require special STM read and write operations to be invoked. This translation is implemented in the source-to-bytecode compiler (for transaction management operations) and the bytecode-to-native compiler (for individual field accesses). The intermediate Java bytecode format is unchanged.

We describe the lower level interface in Section 2.1 and then show how we map `atomic` onto these operations in Section 2.2.

2.1 STM interface

The STM provides operations for non-nesting transactions accessing memory locations on a word-addressed basis. We define five operations for transaction management:

```
boolean done = false;
while (!done) {
  STMStart ();
  try {
    if (condition) {
      statements;
      done = STMCommit ();
    } else {
      STMWait();
    }
  } catch (Throwable t) {
    done = STMCommit ();
    if (done) {
      throw t;
    }
  }
}
```

Figure 1: A CCR of the most general form `atomic (condition) { statements; }` expressed in terms of transaction start/commit operations, assuming that `done` is an otherwise unused identifier. In practise exception propagation is complicated by the fact that the translated code should retain the expected `throws` clause.

Native transaction management
`void STMStart()`
`void STMAbort()`
`boolean STMCommit()`
`boolean STMValidate()`
`void STMWait()`

The first four of these have their usual meaning in transaction processing [1]. `STMStart` begins a new transaction within the executing thread. `STMAbort` aborts the transaction in progress by the executing thread. `STMCommit` attempts to commit the transaction in progress by the executing thread, returning `true` if this succeeds and `false` if it fails. `STMValidate` indicates whether the current transaction would be able to commit – that is, whether the values read represent a current and mutually consistent snapshot and whether any locations updated have been subject to conflicting updates by another transaction. It is an error to invoke `STMStart` if the current thread is already running a transaction. Similarly, it is an error to invoke the other operations unless there is a current transaction.

The fifth operation, `STMWait`, is one that we introduce for allowing threads to block on entry to a CCR. It ultimately has the effect of aborting the current transaction. However, before doing so, it can delay the caller until it may be worthwhile attempting the transaction again. In a basic implementation `STMWait` would be equivalent to `STMAbort`, leading to callers spin-waiting. In our implementation, `STMWait` blocks the caller until an update may have been committed to one of the locations that the transaction has accessed.

2.2 Construction of atomic statements

Figure 1 summarises how a non-nesting `atomic` block may be expressed in terms of these explicit transaction management operations. Nesting CCRs are implemented within the *same* transaction, counting the dynamic nesting depth and only invoking `STMCommit` when the top-level completes; they do not require support here.

Transaction A:

```
atomic {
  if (x != y)
    while (true) { }
}
```

Transaction B:

```
atomic {
  x ++;
  y ++;
}
```

Figure 2: Initially `x=y=0`. Unless the system performs periodic validation, transaction A may enter an endless loop if it reads `x` before Transaction B commits and then reads `y` after it does so.

Note how exceptions propagate – if an exception reaches the end of an `atomic` block then the operations made up to that point are attempted to be committed and, if that succeeds, the exception is propagated outside the block. If the update fails to commit then the `atomic` block is re-tried, leading to the intended exactly-once semantics.

Java bytecode provides separate operations for accessing different kinds of data: local variables, fields, and array-elements. This means that local variables can continue to be accessed directly. `STMRead` and `STMWrite` operations are only necessary when accessing fields or array-elements.

We add a second method table to each class. This holds references to transactional versions of its methods (compiled on demand) and is used by method invocations within transactional methods. The bytecode-to-native compiler is responsible for inserting `STMValidate` calls to detect internal looping in transactions that cannot commit. This can occur if a transaction has made unrepeatable reads – Figure 2 shows an example.

3. TRANSACTIONS VERSUS CCRS

In contemporary work, Herlihy *et al* developed an alternative STM with a Java-based interface [7]. They expose transactional operations directly, providing methods for transaction management and requiring that objects accessed by the transaction are 'opened' before use.

A frequent question, given our use of a general STM as an implementation mechanism, is whether we should also provide explicit transaction management operations to the application programmer. The ability to abort a running `atomic` block is of particular interest.

For illustration, consider a case in which a thread is attempting to move an object from one container to another. The two containers provide `remove` and `insert` operations and an `insert` operation can fail with an exception if the target container is discovered to be full. Using the `atomic` syntax, a move operation could be implemented as:

```
void move(Container s, Container d, int key) {
  atomic {
    Object o = s.remove(k); /*R1*/
    try {
      d.insert(k, o); /*I1*/
    } catch (ContainerFullExn e) {
      s.insert(k, o); /*I2*/
    }
  }
}
```

The code is far from elegant; the programmer must manually implement appropriate fix-up operations in the case

of discovering that the destination container is full. Furthermore, when `R1` has to be counteracted by `I2` the underlying software transaction may involve numerous updates even though the abstract state of the two containers is unchanged. It may even be necessary to consider exceptions raised by `I2`.

It is therefore tempting to consider variants of `atomic` which would allow allow a transactional abort to be invoked. Two options are immediately. The first is that exceptions which leave an `atomic` block should cause any updates made thus far to be abandoned – in this case an exception thrown by `I1` would cause any updates made by `R1` to be discarded. The second option is to provide an explicit 'abort transaction' operation as a method call.

In Section 3.1 we discuss the consequences of allowing any kind of abort operation within CCRs. In Section 3.2 we discuss the particular mechanism of using exceptions in this rôle.

3.1 Allowing abortable CCRs

Allowing a deliberate abort operation appears attractive, not least given our use of an STM (and given the titles of the papers in which we have presented this work). However, it is by no means clear that it is applicable in other implementations of `atomic`.

For instance, given our intended semantics, a simplistic implementation is that entering an `atomic` block will actually switch the system to single-threaded execution and leaving it will resume the ordinary scheduling discipline. This may be appropriate for low-end devices, or for applications which are actually single-threaded[1].

Of course, as well as explicit aborts, the implementation may have to abort transactions that have made unrepeatable reads (as we illustrated in Figure 2). However, note that the need to perform validation-triggered aborts can be avoided by using a CCR implementation which enforces strict isolation: if a transaction cannot fail validation then it cannot be required to abort.

Allowing explicit aborts places a stronger requirement in necessitating that the implementation of `atomic` retains the ability to roll-back updates, even if strict isolation is used in its implementation.

3.2 Aborting CCRs with exceptions

A separate problem, if exceptions were to be used to trigger aborts, is how to expose the abort to the code around the `atomic` block.

It is unreasonable to simply propagate the exception in question. Exceptions in Java are ordinary objects which extend a designated superclass; they can have fields and, although the usage '`throw new ContainerFullExn()`' would be common, the exception thrown could be an existing object. If the exception object was instantiated or modified within the CCR then retaining it outside is incompatible with rolling back the other modifications made. Indeed, in the general case, it would be unclear which modification to retain and which to lose if the exception object was interlinked with the data structure being updated.

Furthermore, using existing exceptions as triggers for roll-back could destroy invariants assumed by existing library code. For example, a library implementor may ensure that some particular kind of exception is only thrown after the data structure has reached some a given state. This guarantee would be broken if changes leading up to the exception were automatically rolled back.

[1]Such a scheme was, of course, used in earlier systems providing CCRs in non-preemptive environments. In the Java Virtual Machine a number of niggling problems would remain, e.g. finalizer execution and class initialization.

3.3 Discussion

The question of whether to provide an abort operation seems separate from the use of an STM as a concurrency control mechanism; in examples such as the `move` operation it could benefit programmers of single-threaded code as well as those developing concurrent systems. One could say that these problems are correctly avoided by designing an appropriate interface for the container – in this case one that allows a capacity check to be made before attempting an insert.

However, if an abort operation is to be provided then there seems to be a strong case that:

- As with our existing semantics, ordinary exceptions are not used to automatically trigger roll-back. This avoids the problems of exception-carried state and library invariants.
- CCRs which may be aborted should be able to be identified through a straightforward static analysis. This allows streamlined implementation in single-threaded code or in non-STM mechanisms which do not ordinary provide an abort operation.

One option, which we hope to explore in future work, is to introduce a single new exception class for triggering aborts. It would subclass `java.lang.Exception` and therefore – as a checked exception – would need to be declared in methods' `throws` clauses. This means that the existing analysis performed to track exception propagation would indicate which CCRs could possibly require roll-back. If the exception class is declared `final` then it cannot be sub-classed and so cannot be used to expose links to objects instantiated within the aborted CCR. Separate instances of it could be used to expose different reasons for aborted CCRs.

4. SIDE EFFECTS

The second subject which we consider in this paper is how to support `atomic` blocks with external side effects. In the system described in our OOPSLA paper we prohibited all `native` methods – that is, all methods that are not implemented in Java bytecode. This ultimately restricts the availability of most I/O operations. Fortuitously simple text-based output remains possible if the characters written are absorbed by buffering within the standard Java libraries. However, this is far from being a comprehensive solution to the general case.

4.1 Side effect visibility

There are some series of operations for which the JVM cannot guarantee atomicity. For example, the programmer may attempt to define an `atomic` block to swap the names of two files by a series of `renameTo` method invocations. There is nothing that the JVM can do to make these operations atomic unless the operating system provides such a facility; all that can reasonably be provided is atomicity with respect to the operations of other threads in the JVM. Again,

this is consistent with our intended 'as-if single threaded' semantics.

There is an interaction between the ability to explicitly abort transactions and the range of features that can be supported within them. As in Section 3 the problem is that aborting requires undo information to be available – in this case operations with external side effects may have been performed and so even if a 'balancing' operation is possible, other processes may have observed the intermediate state.

4.2 Implementation options

In the general case, there is not a clear solution to the problem of supporting arbitrary native methods with side effects. However, aside from a simple prohibition of operations with external side effects, a number of possible directions exist.

The first option, which would allow general native methods to be executed, is to actually use single-threaded execution of blocks that perform operations outside the STM. While unlikely to give good performance in concurrent programs, this option shows that a simple complete implementation remains possible.

A second possibility is to develop some kind of layer *below* the native methods' implementation in order to ensure serialisation of transactions at the system call level; perhaps dealing with a small number of system calls is more practical than dealing with an aribtrary number of native methods. Czajkowski and Daynès' work on running native methods in a separate process may be useful here [5], as is our own work on machine virtualization [2].

A third option is to prohibit the execution of native methods within CCRs but to provide separate mechanisms which library implementors can use where operations are required to have side effects. This is the option that we are currently following. It requires key I/O-related libraries to be re-implemented, but it would allow application software to be used without modification and it would not require the low-level intricacies of supporting arbitrary native methods. One direction which is particularly interesting is integrating JDBC-style database transactions within `atomic` blocks.

Our proposed scheme introduces a reflective interface to the STM. The general idea is that libraries implementing transactional output will temporarily exit the current transaction and use library-specific mechanisms to buffer operations with tentative side effects. When the transaction attempts to commit, a two-phase protocol is used to ensure that the commit is acceptable to each library involved. The side effects can then be exposed externally (or buffered data discarded). Input can be performed in a similar manner, except that data read must be buffered for subsequent rereading if the transaction aborts rather than consuming it.

The reflective interface defines the following operations:

Java transaction reflection

```
TransactionHandle currentTransaction()
void writeExit(TransactionHandle t)
void readExit(TransactionHandle t)
void readEnter(TransactionHandle t)
void writeEnter(TransactionHandle t)
void registerCallback(TransactionHandle t,
                      TransactionCallback cb)
```

The first operation, `currentTransaction` returns a handle representing the currently active transaction (or `null` if no transaction is active).

The next set of four operations are used to control whether write operations made by the calling thread are global or transactional and whether or not read operations will see updates made by the indicated transaction. In each case the supplied transaction handle must be the most recent transaction associated with the current thread and must not have committed or aborted.

Invoking `writeExit` causes subsequent writes by the caller to be made directly to the heap. Similarly, invoking the `writeEnter` operation causes subsequent writes to be made transactionally. A `readExit` call indicates that subsequent reads *need not* reflect updates made by the current transaction, while `readEnter` requires that they *must* – this pair of operations could be no-ops if `atomic` blocks are implemented by single-threaded execution, but can offer performance benefits if transactions are built over an STM. The intent is that a thread will `writeExit` the current transaction, copy data it requires access into globally-visible structures, then `readExit` the current transaction and perform appropriate buffering or external communication as appropriate.

The final operation `registerCallback` requests that a call-back is made when the specified transaction attempts to commit or to abort. The call-back object `cb` defines two operations:

Java call-back interface

```
boolean canCommit(TransactionHandle t)
void setResult(TransactionHandle t,
               boolean result)
```

These operations are used to perform voting on commit and to inform each registered call-back of the eventual result. Once more, the ability to perform roll-back is necessary if `canCommit` may ever return `false`.

It is hoped that this scheme is sufficiently flexible to accommodate a wide range of different forms of input and output. For instance, interaction with an external database would be possible by associating an external transaction identifier with the current transaction handle and deferring commit/abort of the external transaction until a decision is made about the CCR in which it is being performed. Of course, this requires a suitable interface to the external database that is compatible to the two-phase voting algorithm proposed here.

5. CONCLUSION

This paper has examined two particular aspects of our `atomic` construct.

The first is the attraction, in many situations, of being able to automatically roll-back updates made within an `atomic` block. This is straightforward to implement in our current scheme based on an STM, but as we have shown it may not have an effective implementation in other situations. We have also shown how it is important to consider the implications of the mechanism used to trigger roll-back.

The second area is that of performing operations with externally visible side effects – that is, operations which cannot be deferred within the STM. We have described our current ideas for providing I/O operations as part of `atomic` blocks.

In future work we will assess the implementation and performance consequences of these different options. A third area, which we have not dwelt on here, is the performance of long-running `atomic` blocks. As with long-running database

transactions, this is probably an area in which further design work is necessary when compared with the shorter-running statements which we have studied to date.

6. REFERENCES

[1] Bacon, J., and Harris, T. L. *Operating Systems: Concurrent and Distributed Software Design*, 3rd ed. Addison Wesley, 2003.

[2] Barham, P., Dragovic, B., Fraser, K., Hand, S., harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. In *Symposium on Operating Systems Principles (SOSP '03)* Oct. 2003.

[3] Brinch Hansen, P. Distributed processes: A concurrent programming concept. *Communications of the ACM 21*, 11 (Nov. 1978), 934–941.

[4] Brinch Hansen, P. Edison – a multiprocessor language. *Software – Practice and Experience 11*, 4 (Apr. 1981), 325–361.

[5] Czajkowski, G., and Daynès, L. Multitasking without compromise: a virtual machine evolution. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '01)* Nov. 2001.

[6] Harris, T., and Fraser, K. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Langauges & Applications (OOPSLA '03)* Oct. 2003.

[7] Herlihy, M., Luchangco, V., Moir, M., and Scherer, W. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing* (July 2003), pp. 92–101.

[8] Hoare, C. A. R. Towards a theory of parallel programming. In *Operating Systems Techniques* (1972), vol. 9 of *A.P.I.C. Studies in Data Processing*, Academic Press, pp. 61–71.

[9] Liskov, B., and Scheifler, R. Guardians and actions: linguistic aupport for robust, distributed programs. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381–404.

[10] Scott, M. L. Language support for loosely coupled distributed programs. *IEEE Transactions on Software Engineering SE-13*, 1 (Jan. 1987), 88–103.