J. Parallel Distrib. Comput. 73 (2013) 150–163

Contents lists available at SciVerse ScienceDirect

J. Parallel Distrib. Comput.

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

Software transactional memories for Scala

Daniel Goodman*, Behram Khan, Salman Khan, Mikel Luján, Ian Watson

School of Computer Science, The University of Manchester, Oxford Road, Manchester M13 9PL, UK

ARTICLE INFO

Article history:
Received 10 November 2011
Received in revised form
8 September 2012
Accepted 25 September 2012
Available online 5 October 2012

Keywords:
Scala
Transactional memory
Software transactional memory

ABSTRACT

Transactional memory is an alternative to locks for handling concurrency in multi-threaded environments. Instead of providing critical regions that only one thread can enter at a time, transactional memory records sufficient information to detect and correct for conflicts if they occur. This paper surveys the range of options for implementing software transactional memory in Scala. Where possible, we provide references to implementations that instantiate each technique. As part of this survey, we document for the first time several techniques developed in the implementation of Manchester University Transactions for Scala. We order the implementation techniques on a scale moving from the least to the most invasive in terms of modifications to the compilation and runtime environment. This shows that, while the less invasive options are easier to implement and more common, they are more verbose and invasive in the codes using them, often requiring changes to the syntax and program structure throughout the code.

1. Introduction

In this paper, we survey the range of options for adding software transactional memory (STM) [11] to Scala, examining the advantages and disadvantages of each approach from the view of both the application programmer and the transactional memory implementer. This survey includes the introduction of several thus far undocumented techniques developed during our work on Manchester University Transactions for Scala (MUTS) which complement our initial implementation [9], specifically the work within bytecode rewriting on the use of closures and annotations. Where possible, we provide references to implementations that use each surveyed technique and describe in detail the technique used. We will attempt to order the techniques on a scale moving from the least invasive through to the most invasive in terms of modifications to the compilation and runtime environment. This will show that, while the less invasive options are easier to implement and more common, they are also more verbose and invasive in the codes using them, often requiring changes to the syntax and program structure throughout the code. Overall, we show that, as with all tradeoffs, the optimum point is not at either end, but somewhere in the middle.

1.1. *Why Scala*

As multi-core and many-core processors continue to increase their prevalence and core count, there is a growing need to develop general-purpose programming models which can effectively allow programmers to harness these processors. Approaches which appear to be gaining traction are the addition of elements of functional programming to conventional languages and the use of transactional memory. Scala's possibly unique combination of functional programming and imperative/object-oriented programming means that, although every technique for implementing STM described in this paper is possible in other languages, the use of Scala allows the examination of the entire range within a single environment.

This work was carried out as part of the Teraflux project [17], which is examining possible future many-core architectures and programming models. This includes a high-level programming model based on Scala and containing transactional memory. After examining the currently available transactional memories for Scala and failing to find one that fulfilled our requirements, we decided to examine the possible options and tradeoffs of different strategies in order to implement our own transactional memory (MUTS) and facilitate the implementation of other transactional memories.

Before we examine the different implementation strategies for STMs, we will introduce in more detail transactional memory, software transactional memory, and Scala.

1.2. *Transactional memory*

To ensure that correct results are generated by programs performing concurrent access to shared state, it is necessary to control access to the shared state. Without such controls, errors can occur when threads interleave reads and writes. For example, consider a program that contains a counter to keep track of the number of completed threads. In this example, each thread at the end of its execution performs the assignment `counter` = `counter` + 1 to increment the counter. Unfortunately, if thread *A* reads the value of

* Corresponding author.
E-mail address: Daniel.Goodman@cs.man.ac.uk (D. Goodman).

0743-7315/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
doi:10.1016/j.jpdc.2012.09.015

proc 1	proc 2
Read counter	
Increment counter	Read counter
	Increment counter
	Write counter
Write counter	

Fig. 1. An example demonstrating the problem with uncontrolled concurrency. The execution of these two statements should increment counter by 2, but because of their interleaving they only increment the value by 1.

`counter`, then Thread *B* reads the value of `counter`, increments it and writes it back, when thread *A* writes back its result, because it read `counter` before *B* incremented the counter, the value of `counter` will be one less than it should be, and the computation may no longer behave correctly. An example of this interleaving can be seen in Fig. 1.

Traditionally, this issue is handled by some form of locking that restricts the access to the shared state to a single thread at a time. This approach has several complications.

1. The composition of functions that require a lock forces the encapsulation of the implementation of the composed functions to be broken. For example, if we have functions to deposit and withdraw money from bank accounts, each of these functions will require a lock to ensure that the balances remain correct. To now construct a function that atomically transfers money from one account to another so that all money can remain accounted for at all times, it is necessary to get the locks of both the sending account and the receiving account before the functions can be invoked. To achieve this, the locking mechanisms for the individual functions, and therefore information about their internal data structures, have to be made available to the programmer. This breach of encapsulation affects both programmability and maintainability.
2. Code that requires multiple locks to be acquired is prone to dead-lock or live-lock as competing functions may attempt to gain the same set of locks in a different order. If the set of locks required is known in advance, then a total order can be applied to the locks to overcome this, but for a large collection of interesting problems it is not possible to determine the set of locks required ahead of the computation, for example, exploring a graph in order to make modifications.
3. Locking is pessimistic: it assumes that there will be a conflict, and so restricts access on the basis of this. This pessimistic nature of the locking means that opportunities for concurrency are missed. As the world is forced to use parallel processing due to physical limits preventing the increase in speed of single-core processors, this is becoming a serious issue.

One solution to these problems is transactional memory [11]. Here, instead of taking locks, the code executes and records sufficient information such that conflicts can be detected and, in the event of a conflict, one of the conflicting transactions can be rolled back and restarted while the other commits. The underlying system is constructed so that, when the transaction has completed, it will attempt to commit, and if it succeeds, its changes to the system will appear atomically to all other threads; otherwise, the transaction will roll back such that, as far as all other threads are aware, it never executed. This means that the semantics of transactions are equivalent to having a single global lock that the transaction takes, while it executes, but because of the underlying implementation the possible concurrency can be far higher.

We will now describe how this alternative approach addresses the specific points raised about locks.

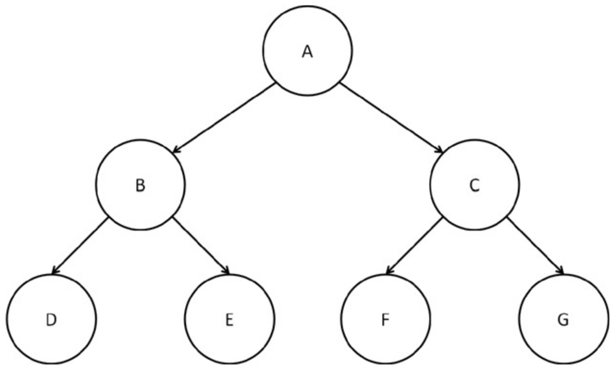


Fig. 2. A simple binary tree for demonstrating transactional memory.

1. Because transactions can be nested within one another, and the collection of data and the handling of collision detection is all dealt with by the transactional memory, the user does not need to be aware of any of the internal locking information of functions, so the composition of functions does not require the breaking of encapsulation.
2. As there are no specific locks, just logged information, there is no possibility of dead-locking. Live-lock is possible with some collision detection mechanisms, but the use of live-lock-free collision management resolves this.
3. Transactions do not have locks that prevent code from executing concurrently. As a result, they are optimistic, and concurrency is not restricted for non-conflicting actions.

1.2.1. Example

To demonstrate the behaviour of transactional memory, we will look at an example of access to a binary tree. A labelled representation of the tree can be seen in Fig. 2. For this example, each node contains some data and up to two child references. We will now consider the behaviour with transactional memory under several scenarios.

Two threads read F: Both threads will traverse the tree, reading the children of *A* and *C* before reading the content of *F*. As all these actions are reads, there are no conflicts, so the transactional memory system will allow both threads to progress in parallel and commit.

One threads reads F and another modifies G: Both threads will read from *A* and *C*; the first thread will read the content from *F* and the second thread will write to *G*'s content. As the second thread does not modify any of the values read by the first, again there is no conflict, and the transactional memory will allow the threads to run in parallel and commit. This would also be true if the first thread was modifying the content of *F* or, because the node children and their content are separate, if one of the threads modified *C*'s data.

Two threads modify F and G: In this scenario, a conflict is possible. If either thread completes before the other reaches *F* or *G*, then there is no conflict. However, if both threads attempt to modify the nodes at the same time, the transactional memory system will detect the conflict, and one of the threads will be reverted and retried. This reversion will result in the computation already performed by this thread being wasted.

Aside from the composability and dead-lock or live-lock freedom of transactions, the hope with transactional memory is that the wasted computation is less than the computation that would otherwise be required to manage fine-grained locking at a level which is able to support a similar level of parallelism.

Locking. If the above example had been implemented with locking, it would have to use fine-grained locking to allow any parallel execution. This locking could use a lock per node, acquiring the required node locks at the next level down before releasing the lock at the current node if this node is not to be modified. Alternatively, there could be separate read and write locks for each node. Both strategies are complex to implement correctly and include a large number of locking operations.

Currently optimized locking can outperform transactional memory; however, a straight performance comparison is unfair, as locking is able to take advantage of hardware support that is currently unavailable to transactional memory, but such support is becoming available [2,6,12].

1.3. Software transactional memory

Software transactional memory (STM) [16], as the name suggests, is a transactional memory implemented entirely in software, so without any specific hardware support. Conceptually, STMs consist of three parts.

1. A backend that records the actions within the transaction. This handles the logging of reads and writes, conflict detection, and the committing of the transaction.
2. A frontend that the user interacts with. This is responsible for providing a user interface through which the user selects the areas of code and variables within the code that are to be transactional. If features such as the automatic retrying of transactions in the event of failure are handled by the STM, this is the part of the STM that will perform that task.
3. A boundary interface. This is an interface that the backend will implement and the frontend will make calls to. The provision of this interface allows for different backends to be chosen according to the code or operating environment, and for multiple different frontends to be used at the same time.

This separation is not to say that there are not implementations that dispense with the boundary interface and attempt to merge the frontend and backend code, but that it is always possible to refactor such codes into these three components without loss of functionality. In principle, all sufficiently functional backends can be fitted to all frontends, and the backends are simply a library implementing one of the many well-documented and surveyed transactional memory algorithms [11]. Therefore, in this paper we will confine our survey to the available frontends and the user and implementer tradeoffs involved in using each of these.

Hardware transactional memory is starting to appear in commercial processors [2,12]; however, despite the performance benefits of such systems, they are unlikely in the near future to replace software transactional memory, as by their nature they are bounded to finite-sized transactions. Instead, hardware TM will probably be incorporated into the backend of software TM to improve the performance of software TM by providing a hybrid TM system [7]. For such a system the work on the frontend discussed in this paper is still relevant.

1.4. Scala

Scala [15] is a general-purpose programming language designed to smoothly integrate features of object-oriented [19] and functional languages [3]. By design, it supports seamless integration with Java, including existing compiled Java code and libraries. The compiler produces Java byte-code [14], meaning that you can call Scala from Java and you can call Java from Scala. Scala is a pure object-oriented language in the sense that every value is an object. The types and behaviour of objects are described by classes and traits which are extended by subclassing and a flexible mixin-based composition mechanism as a replacement for multiple inheritance. However, Scala is also a functional language in the sense that every function is a value. This is furthered through the provision of a lightweight syntax for defining anonymous functions, as shown in Fig. 3, and support for higher-order functions, also seen

```
def apply( f1:() => Int, f2:Int => Int) {
  println(f2(f1()))

......
......

apply(() => 42, (x:Int) => x*x)
}
```

Fig. 3. A snippet of Scala code demonstrating the construction of two anonymous functions and them being passed as arguments into a third conventional function. The first anonymous function takes no inputs and produces an output (42). The second anonymous function takes an integer as an input and returns this value squared. The function `apply` takes these two functions as arguments and applies the second function to the output of the first. It then prints the resulting value.

in Fig. 3, for the nesting of functions, and for currying. Scala's case classes and its built-in support for pattern matching algebraic types are equivalent to those used in many functional programming languages.

Scala is statically typed and equipped with a type system that statically enforces that abstractions are used in a safe and coherent manner. A local type inference mechanism means that the user is not required to annotate the program with redundant type information.

Scala supports two types of reference: immutable values, which are denoted `val`, and mutable variables, which are denoted `var`. While this has no effect on the overall behaviour of the transactional memory interfaces described, as it is good coding practice, where possible we will convert variables into values. An example of such an occasion is when the mutable state is encapsulated within an object such as a reference cell.

Finally, Scala provides a combination of language mechanisms that make it easy to smoothly add new language constructs in the form of libraries. Specifically, any method may be used as an infix or postfix operator, and closures are constructed automatically depending on the expected type (target typing). A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like metaprogramming facilities.

2. Evaluating Scala STMs

There is a wide range of techniques for implementing the frontends of STMs in Scala. These work through a combination of one or more of the following elements: library calls, annotations, byte-code rewriting, and compiler modifications. Also, Scala's ability to define anonymous functions and to pass them into other functions as arguments (closures) is important in many Scala STMs, and an example of the use of anonymous functions as function parameters can be seen in Fig. 3. Table 1 maps the different available techniques to the STMs that implement these techniques.

Before we look at the different ways of implementing these frontends, we will briefly consider some of the points that these should be rated against. Table 2 provides a summary for each implementation strategy of the properties that take a discrete value.

Java compatible: Scala is Java compatible; however, this does not mean that all frontends are automatically compatible with Java. Some require features of Scala that cannot be described in Java code without a high level of understanding of the workings of the Scala compiler. For example, using frontends that require the passing of compiler-inserted implicit parameters or the passing of functions as arguments to other functions is problematic in Java code. For techniques that are not Java compatible, we note if the technique can be combined with techniques that are Java compatible to allow transactions accessing the same shared state to be constructed in both Scala and Java.

Table 1
Summary of the different implementation strategies and software transactional memories that implement them.

Implementation style		Implementations
Pure libraries	Explicit library calls	–
	Reference cells	Multi-Verse [18] CCSTM [5] ScalaSTM [5] RadonSTM [4]
	Closures and reference cells	CCSTM [5] ScalaSTM [5]
Byte-code rewriting	Class annotations	Multi-Verse [18]
	Method annotations	MUTS [9]
	Closures	MUTS [9]
	Parser modifications	MUTS [9]
Compiler modifications		–

Supports strong isolation: If a TM supports strong isolation, no action of the programmer or user can result in transactional data being accessed from outside of a transaction. If instead only weak isolation is supported, it is possible to access data being manipulated in a transaction from outside of a transaction. This can then allow the non-committed state of a transaction to be observed and modified, potentially undermining the correctness of the code. Strong isolation can also be broken by the presence of non-transactional mutable data within a transaction; such data may exist as an optimization.

Supports legacy code: The use of code compiled by others is a routine part of writing programs; however, not all STMs are able to instrument this code. This means that it is not possible to include within transactions compiled methods that contain side effects. Examples of such methods are those contained within the Scala mutable collections and the Java Util libraries.

No runtime modifications: Some STMs form a simple library that can just be added to the classpath; others require the addition of more advanced JVM arguments, or the use of special compilers.

Clarity of code: Different methods of implementing transactional frontends require different syntaxes. These different syntaxes offer differing levels of complexity and verbosity. Excessive verboseness or complicated syntax can act as a barrier to the use of the transactional frontend, or result in difficulty reading and writing code that uses it.

No alternative syntax: A specific point relating to clarity of code is that while some STMs may be less verbose than others, some of these require that the code inside a transaction uses a different syntax to code outside a transaction. For example, an alternative assignment statement is required for some techniques. As the standard assignment operator can still be used, but may not be valid, such changes not only increase the application programmer workload, but also allow for the creation of subtle bugs.

No duplicate methods: To call functions from both inside and outside a transaction, some STMs required that the user constructs two copies of methods, one that is transactional and one that is not. Others are able to automatically construct a transactional and a non-transactional method from a single piece of user code. The ability to construct both methods from a single piece of code removes the opportunity for discrepancies between the two functions to be added when writing or maintaining the code, as well as reducing the workload for the programmer.

Guarantees correct transactions: Some STMs guarantee that all variables that need to be instrumented as part of a transaction will be, while others leave it to the programmer to ensure this. If the programmer fails to add all the required instrumentation, then the transaction may no longer be correct, and a race condition will have been introduced into the program.

2.1. Performance

The performance of the techniques can be affected by a range of issues, including the following.

- The backend used to support the STM.
- The level of optimization provided by the compiler.
- The level of code analysis implemented within the technique.
- The hardware/VM used to execute the program.
- The problem they are used to solve.

For example, techniques that introduce the indirection to the backend before compilation may prevent optimizations such as partial evaluation being applied. This optimization would be possible if the indirection were introduced after compilation. Other compilers may be more advanced and still introduce the optimization, or less advanced and not employ this optimization at all. Another example would be techniques that examine the byte-code, which may or may not do analysis to identify code that does not need instrumentation. This level of variation, coupled with the differing scopes of the integration addressed by the different techniques, means that their performance in their own right cannot be sensibly compared. For this reason we shall not attempt to compare the performance of the techniques, but would point the reader to the performance analysis provided by specific implementations for specific problems on specific hardware [11,5,13].

3. Scala STMs

We will now look in detail at how each of the different STM frontends can be implemented, appraising as we go the advantages and disadvantages of each. In order to keep these lists complete, many of the advantages and disadvantages will be duplicated for the different approaches. To demonstrate the use of the different options, we will use the example code shown in Fig. 4 for transferring money between accounts and keeping some simple logging. It consists of two classes: a class `Account` that contains a balance and methods for adding and removing money, and a class `Teller` that contains methods for performing operations on accounts.

Table 2
Summary of approaches and their properties.

	Pure libraries			Byte-code rewriting				Compiler modifications
	Explicit library calls	Reference cells	Closures and reference cells	Class annotations	Method annotations	Closures	Parser modifications	
Java compatible	●	●		●	●			
Combinable with Java compatible techniques	–	–	●	–	–	●	●	
Supports strong isolation				●				●
Supports legacy code				●	●	●	●	
No runtime modifications	●	●	●					●
Guarantees correct transactions				●	●	●	●	●
No alternative syntax	●			●	●	●	●	●
No duplicate methods				●	●	●	●	●

```
class Teller(log:Log) {
  var transactions = 0

  def transfer(from:Account, to:Account, amount:Int) {
    println("About to transfer " + amount)

    from.withdraw(amount)
    to.deposit(amount)
    log.value = log.value + amount
    transactions = transactions + 1

    println("Transferred " + amount)
  }
}

class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }

  def withdraw(amount:Int) {
    value = value - amount
  }
}
```

Fig. 4. Scala code for transferring money between accounts while keeping some basic logging. This contains two classes: a class `Account` that contains a balance and methods for adding and removing money, and a class `Teller` that contains methods for performing operations on accounts. This example will be used to demonstrate the syntax of the different STM frontends.

3.1. Pure libraries

In this section, we will examine the transactional memory implementations that can be provided as just a library without any supporting infrastructure. The principal advantage of this approach is that it can be used by any programmer without the use of other tools. However, this also means that it can only be applied to functions that they control the source code for. In addition, pure software libraries in Scala are unable to provide strong isolation. While libraries such as ScalaSTM claim to offer strong isolation [5], this property only applies if the programmer only writes code that obeys rules set down by ScalaSTM. It is a property of all correct transactional memory implementations that, if no action to breach strong isolation is taken, strong isolation is maintained. For strong isolation to be meaningful it has to be such that, no matter what the programmer or future user does, transactional data cannot be accessed from outside of a transaction.

3.1.1. Explicit library calls

Explicit library calls is the simplest form of STM to implement, and it is used to implement TinySTM [8] in C. The absence of pointers in JVM byte-code makes the implementation even more verbose in JVM-based environments. With this technique there is no frontend, and the user interacts directly with the boundary interface. This means that the user inserts library calls to explicitly start a transaction and create a context to hold the required logging information about the transaction. This is followed by the addition of library calls which use this context for every transactional read and write, and finally a library call to attempt to commit the transaction. In the event of the transaction failing, the user is responsible for constructing all the control logic to handle this failure. An example of the function in Fig. 4 written in this style can be seen in Fig. 5. In this example, the transaction is initialized and its context constructed through the call `c` = `TM.getContext()`. Transactional reads and writes are invoked through the context methods `get` and `set`, and the transaction is committed through the context method `commit`. The methods `get` and `set` require a reference to the object containing the data and the name of the data identifier. The method `set` also requires the value the data should be set to, and for efficiency reasons `get` takes the current value of the data. This example is still missing much of the code required to handle exceptions generated during the transaction and transactional aborts.

Advantages

- The principal advantage of this technique is its simplicity, which allows it to be implemented in almost any language, without any change to the runtime system, compiler, or associated code.
- This method also allows the user explicit control over which reads and which writes are made transactional, allowing for the potential for better optimization of transactions through the elimination of unnecessary recording of reads and writes. However, such optimizations make the code fragile, and bugs can be hard to find.

Disadvantages This technique has many disadvantages, including the following.

- The large volume of library calls obscures the code, making it hard to determine what the code is doing.
- There is no means by which the compiler can alert the user to the absence of a call for a read or write within a transaction. This can lead to values that should be part of the transaction being accessed non-transactionally.
- The user is required to add in all the control logic.
- There is no way that existing compiled code which contains side effects can be used transactionally within a transaction.
- If a function needs to be called from both transactional and non-transactional code, then two copies of the function are required. This makes the construction and maintenance of code significantly harder.

3.1.2. Reference cells

A partial solution to variables being accidentally modified in a non-transactional way from inside a transaction can be found by the construction of reference cells that hold the transactional data. These can then be used in conjunction with the type system to ensure that, as long as all accesses are directed to the reference cell, and not directly to the value it contains, the value will always be accessed transactionally. This access could be done via methods located in a separate object, but typically they are placed either in the reference cell object or the context object. This approach is applicable to almost all languages, and is supported in Scala by RadonSTM [4], Multi-Verse [18], CCSTM [5], and ScalaSTM [5]. The latter two, while able to support this approach, extend it with ScalaSTM using closures, as described in Section 3.1.3, and Multi-Verse extends it through byte-code rewriting, as described in Section 3.2.1. As Scala does not allow = to be overridden, it is necessary to create an alternative method, such as :=, which must be used for assigning new values to reference cells. An example of the code introduced in Fig. 4 can be seen in Fig. 6, modified to use reference cells instead.

Advantages

- The user is not forced to make all values within a transaction transactional; this provides the potential for user-implemented optimization.
- As long as no other references are kept to the value contained within the reference cell, the value within the cell will only be accessible from within a transaction. However, it should be noted that, as transactions may appear within libraries, this property is unenforceable in the general case.
- Potentially Java compatible.

```
import ...

class Teller(log:Log) {
  var transactions = 0

  def transfer(from:AccountTM, to:AccountTM, amount:Int) {
    println("About to transfer " + amount)

    do {
      val c = TM.getContext()
      from.withdraw(amount, c)
      to.deposit(amount, c)
      c.set(log, "value", c.get(log, "value", log.value) + amount)
      c.set(this, "transactions", c.get(this, "transactions", transactions) + 1)

    } while(!c.commit);

    println("Transferred " + amount)
  }
}

class AccountTM {
  var value = 0

  def deposit(amount:Int, c:Context) {
    c.set(this, "value", c.get(this, "value", value) + amount)
  }

  def withdraw(amount:Int, c:Context) {
    c.set(this, "value", c.get(this, "value", value) - amount)
  }

// Required for non-transactional environments
class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }

  def withdraw(amount:Int) {
    value = value - amount
  }
}
```

Fig. 5. An example of the code modification required when adding a transaction to the code in Fig. 4. This code is still missing control logic to handle exceptions and transactional aborts.

Disadvantages

- There is no way that existing compiled code that contains side effects can be used transactionally within a transaction.
- The user is required to construct the code to handle beginning, committing, and retrying transactions.
- If a function needs to be called from both transactional and non-transactional code, or a data structure needs to be used in both transactional and non-transactional code, then two copies are required. This makes the addition and maintenance of transactions in the code significantly harder.
- Values can be used in transactions without wrapping them in reference cells. This allows for side effects from transactions to be visible before committing and for variables to not be reset correctly in the event of a transactional roll back.
- The reference cells require access to a context in order to bind their actions to a transaction. All approaches we are aware of have problems.
  – If implicit parameters are used to hide the use of the context, then the reference cells are not compatible with Java.
  – If explicit references are passed, they obscure the code through too many references to the context.
  – If the cell internally references a global ThreadLocal context or uses some similar technique, then the repeated lookups will impede performance.
- Syntax differences for operations such as assignment exist between transactional and non-transactional code, making the addition or removal of transactions to existing code more labour intensive.

3.1.3. Closures and reference cells

Closures can be used to address the need for the user to explicitly manage the starting, committing, and retrying of transactions. This technique takes advantage of Scala's lightweight construction of anonymous functions, and its ability to pass functions as arguments to other functions. This restricts this technique to languages

```
import ...

class Teller(log:LogTM) {
  val transactions = new RefCell(0)

   def transfer(from:AccountTM, to:AccountTM, amount:Int) {
     println("About to transfer " + amount)

     do {
       val c = TM.getContext()
       from.withdraw(amount, c)
       to.deposit(amount, c)
       log.value := (log.value(c) + amount, c)
       transactions := (transactions(c) + 1, c)
     } while(!c.commit);

    println("Transferred " + amount)
  }
}

class AccountTM {
  val value = new RefCell(0)

  def deposit(amount:Int, c:Context) {
    value := (value(c) + amount, c)
  }

  def withdraw(amount:Int, c:Context) {
    value := (value(c) - amount, c)
  }

// Required for non-transactional environments
class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }

  def withdraw(amount:Int) {
    value = value - amount
  }
}
```

Fig. 6. The code modification required when adding a transaction to the code in Fig. 4 with `RefCell` objects acting as reference cells for the transactional data. Note the use of := instead of = within the transactions. This code is still missing control logic to handle exceptions etc.

```
import ...

class Teller(log:LogTM) {
  val transactions = new RefCell(0)

  def transfer(from:AccountTM, to:AccountTM, amount:Int) {
    println("About to transfer " + amount)

    atomic{ (implicit c:Context) => {
        from.withdraw(amount)
        to.deposit(amount)
        log.value := log.value + amount
        transactions := transactions + 1
      }
    }

    println("Transferred " + amount)
  }
}

class AccountTM {
  val value = new RefCell(0)

  def deposit(amount:Int)(implicit c:Context) {
    value := value + amount
  }

  def withdraw(amount:Int)(implicit c:Context)  {
    value := value - amount
  }

// Required for non-transactional environments
class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }

  def withdraw(amount:Int) {
    value = value - amount
  }
}
```

Fig. 7. The code modification required when adding a transaction to the code in Fig. 4 with reference cells and the control logic hidden in a function that accepts closures. Note the continuing use of := instead of = within the transactions, and the additional implicit parameters.

supporting these properties. With this option, instead of constructing all the control logic, the user passes a closure containing the code they would like to execute transactionally to a library function. This library function then handles the creation of the transaction context and the control logic for committing and retrying if a transaction fails. The use of closures can be combined with a range of options to form a frontend, and we will look at other forms in Section 3.2.3. However, in a pure library they are normally combined with reference cells. The general form is that the user provides a closure that takes a context as a parameter. The function receiving the closure will then execute the transaction, calling the closure at the appropriate moments. The context can be an implicit parameter, allowing it to be omitted from the rest of the closure, with the compiler adding it in again later. This is the technique used by ScalaSTM [5], CCSTM [5], and Multi-Verse [18], and an example can be seen in Fig. 7.

Advantages

- The user is not forced to make all values within a transaction transactional; this provides the potential for user-implemented optimization.
- As long as no other references are kept to the value contained within the reference cell, the value within the cell will only be accessed from within a transaction. However, it should be noted that, as transactions may appear within libraries, this property is essentially unenforceable in the general case.
- The library handles beginning, committing, and retrying of transactions.

Disadvantages

- There is no way that existing compiled code that contains side effects can be used transactionally within a transaction.
- If a function needs to be called from both transactional and non-transactional code, or a data structure needs to be used in both transactional and non-transactional code, then two copies are required. This makes the addition and maintenance of transactions in the code significantly harder.
- Values can be used in transactions without wrapping them in reference cells. This allows for side effects from transactions and the potential for variables to not be reset correctly in the event of a transactional collision.
- The use of closures means that it is not Java compatible.

- The reference cells require access to a context in order to bind their actions to a transaction. All approaches we are aware of have problems.
  - If implicit parameters are used to hide the use of the context, then the reference cells are not compatible with Java.
  - If explicit references are passed, they obscure the code through too many references to the context.
  - If the cell internally references a global ThreadLocal context or uses some similar technique, then the repeated lookups will impede performance.
- Syntax differences for operations such as assignment exist between transactional and non-transactional code, making the addition or removal of transactions to existing code more labour intensive.

3.2. Byte-code rewriting

If an STM is going to work with legacy code, or the code is to contain functions that can be used both from inside and outside of transactions, then it is necessary it use some form of modification to the JVM to allow code to be used in both transactional and non-transactional contexts. One such form of this is byte-code rewriting. The idea with byte-code rewriting is that, instead of the user adding in the instrumentation directly through reference cells or library calls, they mark the code that should be transactional and then the instrumentation is added by a rewrite of the code. This rewrite can either occur offline, or when the code is loaded into the JVM. As all transactional code must be present at the time of the rewrite, and this may include libraries in the JVM, performing the rewrite when the code is loaded into the JVM is usually the preferred option.

The rewriting of code in the JVM is achieved through the construction of a Java Agent that is provided to the JVM as an argument. The Java Agent is a program that receives the byte-code from the class loader, and returns it, having read it and if appropriate modified it. Java Agents can be used for gathering statistics about code, modifying its behaviour, or simply for instrumenting it to gather information for performance analysis or debugging.

As discussed above, to add transactions to existing code using a rewrite, the rewrite must insert additional code to perform the instrumentation. Typically, this will be the addition of calls such as those discussed for pure libraries, the difference being that now, instead of the user having to add the calls, the calls will be added automatically, with the user simply marking the scope of the transactions through one of the mechanisms discussed below. This hides complexity from the user and prevents the code they work with from being obscured by the calls. Within these different annotation mechanisms rewrites are one of two types: those that only make changes relative to the source code that contains the transaction, and those that make wider changes to allow the transaction to expand into other code that it interacts with. Before we introduce the different ways of marking transactions, we will discuss the implementation of both types.

Code local to the transaction: If the modifications are to be made only to the code local to the transaction, the mechanism is relatively simple. The Java Agent detects the marking for the transaction and then for the region marked instruments the code using either reference cells or library calls.

All code in the program: This second type is more interesting, as this is the type that allows transactions to span into unmarked and precompiled code. It is also the more involved option, as methods may appear in both transactional and non-transactional code, and in general there is no way of determining which situation the method will appear in when it is loaded, or deduce that the method will only be used in one context. For example, two objects can be constructed from the same class, and one may only be used within transactions, while the other is only used in non-transactional code, or indeed the use of an object may change over the course of a program's execution. This means that reference cells cannot be used, and any solution must revolve around the use of library calls when accessing object fields. The standard technique to overcome this is to construct for every non-transactional method a complementing transactional method. These methods are then differentiated from each other by their method signature, with the transactional methods taking a context which they can then use with the instrumenting library calls. In addition, the instrumented methods also use the context when making any other method calls. The net effect of this is that two independent sets of functions are created: one with no instrumentation, and the other where all accesses are instrumented. The system is then completed by the user marking sections that should be transactional and the STM inserting the appropriate code to start the transaction by creating a context and switching code bases, and then when the transaction is complete to switch back and commit.

The applicability of these techniques to other languages is dependent not only on the semantics of the language, but also on the level of support for code rewriting. We will now look at the different ways of marking the boundaries of transactions.

3.2.1. Class annotations

With this approach, the user adds annotations to the class definitions to state that modifications to this class should be transactional. This is then detected by the rewriter and used to instrument the class. For example, in Multi-Verse [18], the presence of this annotation results in the automatic insertion of reference cells replacing either the object's fields or the entire object, depending on the configuration of the STM. In addition, all the methods within the object are now made transactional. To allow multiple transactional methods to be nested within a single method to form a single transaction, it is necessary to indicate a start point for the transaction. This can be achieved either by the addition of library calls or, as in Multi-Verse, the addition of method annotations. This is applicable in most languages, and an example of this can be seen in Fig. 8.

Advantages

- The use of annotations on the objects simplifies the addition of transactions to programs.
- The user is not forced to make all values within a transaction transactional; this provides the potential for user-implemented optimization.
- Can be compatible with all JVM-based languages.
- Can provide strong separation of transactional and non-transactional data.

Disadvantages

- There is no way that existing compiled code which contains side effects can be used transactionally within a transaction.
- The automatic creation of transactions for all methods within an object can restrict the structure of programs, or negatively affect their performance through the execution of unnecessary transactions. These restrictions on the scope of the transactions can potentially cause problems using transactions to maintain data invariants. An example of this for implicit transactions on variable accesses can be seen in Fig. 9. At the method level, this can be overcome with additional annotations, but this further restricts the structure of the program by defining method boundaries. For example, now it is not possible to have a method that contains many lines of non-transactional code and a small transaction. Furthermore, the programmer can forget to include this additional annotation.
- Requires a Java Agent to be added to the JVM arguments.

```
import ...

@TransactionalObject
class Teller(log:Log) {
  var transactions = 0

  def transfer(from:Account, to:Account, amount:Int) {
    println("About to transfer " + amount)

    transferTM(from, to, amount)

    println("Transferred " + amount)
  }

  //Method factored out to constrain the scope of the transaction
  @TransactionalMethod
  private def transferTM(from:Account, to:Account, amount:Int) {
    from.withdraw(amount)
    to.deposit(amount)
    log.value = log.value + amount
    transactions = transactions + 1
  }
}

@TransactionalObject
class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }

  def withdraw(amount:Int) {
    value = value - amount
  }
}
```

Fig. 8. The additional annotations required for object-level transactions via annotations. Note that the duplicate classes and changes to the syntax are no longer required; however, refactoring the methods to constrain the scope of the transactions is required.

proc 1	proc 2
a=2*a	
	if(a==b)
	keep going
	else
	fail badly
b=2*b	

Fig. 9. Code that should maintain the invariant that $a == b$. While strong separation is maintained, the implicitly constructed transactions on the left allow the data invariant to be invalidated.

3.2.2. Method annotations

An alternative approach used in Deuce STM [13] for Java and extended to Scala by MUTS [9] is to mark methods which will be transactional. Methods that are called from this method are then implicitly transactional. To do this, the programmer adds an annotation to the method. This is applicable to most languages. In the case of Deuce STM the annotation is `@Atomic`. The Java Agent then constructs instrumented versions of all methods as described at the start of this section. When the atomic annotation is encountered, the annotated method is also replaced with a method that handles the context construction and control logic for the transactions. This method, having initialized the transaction, then calls the instrumented version of the original method, so starting the transactional code. Once this method completes, the transaction is committed or retried, accordingly. Example code including a transaction can be seen in Fig. 10.

Advantages

- Can use legacy code within transactions.
- Compatible with Java.
- Only one copy of each method is required, even if the method is used both inside and outside of transactions.
- Not possible to fail to instrument values within a transaction.
- Clean code.

Disadvantages

- Granularity of transactions is fixed at the method level. This can adversely affect the structure of the code; for example, if a method requires several small transactions, these all have to be refactored out as new methods.

```
import ...

class Teller(log:Log) {
  var transactions = 0

  def transfer(from:Account, to:Account, amount:Int) {
    println("About to transfer " + amount)

    transferTM(from, to, amount)

    println("Transferred " + amount)
  }

  //Method factored out to constrain the scope of the transaction
  @Atomic
  private def transferTM(from:Account, to:Account, amount:Int) {
    from.withdraw(amount)
    to.deposit(amount)
    log.value = log.value + amount
    transactions = transactions + 1
  }
}

class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }

  def withdraw(amount:Int) {
    value = value - amount
  }
}
```

Fig. 10. The additional annotations required for method-level transactions via annotations. Again note that the duplicate methods and changes to the syntax are no longer required; however, refactoring the methods to constrain the scope of the transactions is. With this setup there is only one annotation, so data invariants will be maintained, but strong isolation will not be provided.

- It is only capable of supporting weak isolation.
- Requires a Java Agent to be added to the JVM arguments.

3.2.3. Closures

The restrictions on the granularity of transactions displayed by method annotations can be addressed at the cost of direct compatibility with Java through the use of closures. Here, through the use of Scala's lightweight syntax for constructing functions, the user constructs a closure that is passed to a library function to be executed transactionally. This allows the construction of lightweight atomic blocks [10] without syntax changes. An example is given in Fig. 11. It is worth noting that this technique can coexist with method annotations, meaning that, although closures cannot easily be used in Java, transactions based on an STM that accepts closures can be.

The implementing library can be constructed by simply adding an atomic method annotation to a function the library provides and which executes the passed function within this function. This is demonstrated in Fig. 12. Alternatively, the library function can contain all the code required to construct the transaction context and its associated control logic, calling to the instrumented version of the passed function once the transaction has been initialized and committing it once the function completes. This will require manipulation of the byte-code when the library is compiled in order to pass the extra argument, but all of this will happen before the library is released to users. This second approach is slightly more efficient than the first, and it is easier to modify the control logic. An example of the required transformation can be seen in Fig. 13. Both approaches have been implemented as part of MUTS [9].[1]

Advantages

- Can use legacy code within transactions.
- Only one copy of each method is required, even if the method is used both inside and outside of transactions.
- Not possible to fail to instrument values within a transaction.
- Clean code.
- Unrestricted transaction granularity.

Disadvantages

- It is only capable of supporting weak isolation.
- Requires a Java Agent to be added to the JVM arguments.
- Not directly compatible with Java.

[1] http://apt.cs.man.ac.uk/projects/TERAFLUX/MUTS/.

```
import ...

class Teller(log:Log) {
  var transactions = 0

  def transfer(from:Account, to:Account, amount:Int) {
    println("About to transfer " + amount)
    atomic{
      from.withdraw(amount)
      to.deposit(amount)
      log.value = log.value + amount
      transactions = transactions + 1
    }
    println("Transferred " + amount)
  }
}

class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }

  def withdraw(amount:Int) {

    value = value - amount
  }
}
```

Fig. 11. The replacement of annotations with a closure taking the method to be executed atomically. While at the machine level the bounding of the transaction is still achieved by factoring out a method, at the code level the resulting fragmentation is no longer present.

```
@Atomic
def atomic(f: => Unit) {
  f()
}
```

Fig. 12. An example of a method that can be used with closures and the annotations provided by Deuce so that the user code does not have to be fragmented in order to assert finer granularities on methods. This works by taking the closure and then through the presence of the annotation, executing the closure in a transaction.

3.2.4. Parser modifications

In the case of some of the byte-code rewriting examples, the syntactic overhead is reduced to importing a library; however, to truly remove all the syntactic overhead it is necessary to modify the parser to accept transactions; for example, see Fig. 14. Having modified the parser to accept and detect atomic blocks, the contents of the block can then be encoded using existing elements of the language. Doing so, it is possible to add in the transformation shown in Fig. 15. This adds all the context creation, the control logic, and the instrumentation of the method variables without the need to further modify the compiler pipeline. Having done all of this, it is then necessary to mark the transaction so that it can be integrated with the instrumented method calls once they are created during the byte-code rewrite. As at this stage, the constructed abstract syntax tree is yet to go through the rest of the compiler and may be restructured: one of the most effective ways to mark the transactional code without the need to further modify the compiler is to wrap it in a try catch for a specially constructed class of exception. The range of the resulting exception handler will then be included in the compiled code and can be clearly read out once the code is compiled, regardless of any compile-time transformations. During the byte-code rewrite, this information can then be used to determine which method calls need to be modified to call their instrumented counterparts. At the same time, all references to the special exception can be removed. The inclusion of such an exception can be clearly seen in Fig. 15. Again, this technique is compatible with both the method annotations and the closure-based versions of TM, and all of these coexist in MUTS [9].

Advantages

- Can use legacy code within transactions.
- Only one copy of each method is required, even if the method is used both inside and outside of transactions.
- Not possible to fail to instrument values within a transaction.
- Complete control of the syntax.
- Unrestricted transaction granularity.

Disadvantages

- Is only capable of supporting weak isolation.
- Requires a Java Agent to be added to the JVM arguments.
- Not directly compatible with Java.
- Compilation must use the modified parser.
- Adds new keywords to the language.

3.3. Compiler modifications

The final and most invasive technique for introducing transactions into Scala is to enter them purely within the Scala compiler. This would allow complete control over the generated code, so allowing properties such as strong isolation to be enforced. However, the structure of the compiler means that this would be an extremely involved task, requiring modifications to many compilation phases. In addition, such a modification would still be restricted to adding transactions to code which is compiled using this compiler. As such, without support for such modifications from not just the Scala community, but also the Java community, this approach would be unlikely to be more effective than one of the byte-code rewriting approaches. As a result of this, we are unaware of any implementations that take this approach.

Advantages

- Is capable of supporting strong isolation.
- Only one copy of each method is required, even if the method is used both inside and outside of transactions.
- Not possible to fail to instrument values within a transaction.
- Complete control of the syntax.
- Unrestricted transaction granularity.
- Requires no modifications to the runtime environment.

Disadvantages

- Not compatible with Java.
- Cannot use legacy code within transactions.
- Compilation must use the modified compiler.
- Adds new keywords to the language.

4. Conclusions

In this paper, we have described all the available approaches to transactional memory in Scala that we are aware of, and where we are aware of implementations that use these approaches we have included references to them. This has included the documentation for the first time of a number of approaches implemented by the MUTS STM suite. We have then rated these approaches against a number of criteria, and ordered them from the least invasive

```
def atomic(block: => Unit) {
  var committed = false
  do {
    val context:Context = TM.beginTransaction()
    try {
      /* This line has been modified in the byte-code to add the
       * context as a function argument. In doing so the execution
       * passes from the non-instrumented code to the instrumented
       * code.
       */
      block.apply(context)
      committed = TM.commit(context)
    } catch {
      case e:TransactionException => context.rollback()
      case e:Exception => {
        committed = TM.commit(context)
        if(committed) throw e
      }
    }
  } while(!committed)
}
```

Fig. 13. An example of a more efficient means of starting transactions using closures. This technique also allows for easier modification of the control logic of the transactions as required. As commented in the code, the call to execute the closure is modified to take the context. This change facilitates the move from uninstrumented code to instrumented code. The exception TransactionException is used allow transactions to abort from within the function 'block' in the event of a collision. When it is thrown, it is caught by the managing code, and the transaction is rolled back and retried.

Table 3
Summary of the different implementation strategies and their relative usability and invasiveness tradeoffs.

Implementation style		Usability issues
Pure libraries	Explicit library calls	• Verbose code • Duplicate methods required • Unable to incorporate legacy code • Required calls can be omitted
	Reference cells	• Verbose code • Duplicate methods required • Unable to incorporate legacy code • Alternative syntax required • Required calls can be omitted
	Closures and reference cells	• Duplicate methods required • Unable to incorporate legacy code • Alternative syntax required • Required calls can be omitted
Byte-code rewriting	Class annotations Method annotations	• Refactoring required
	Closures	• None
	Parser modifications	• Requires modifications to the language specification
Compiler modifications		• Unable to incorporate legacy code • Requires modifications to the language specification
Implementation style		**Invasiveness**
Pure libraries	Explicit library calls Reference cells Closures and reference cells	• None
Byte-code rewriting	Class annotations Method annotations Closures	• Inclusion of a Java agent in the VM boot loader.
	Parser modifications	• Inclusion of a Java agent in the VM boot loader • Modifications to the parser of the Scala compiler.
Compiler modifications		• Extensive modifications to both the Scala and the Java compiler.

to the most invasive in terms of modifications to the compiler and runtime. In doing so, we have shown that there is a trade off between ease of construction and ease of use when programming. A summary of this can be seen in Table 3. As with most tradeoffs, the optimum point is not at either end of the spectrum, but somewhere in the middle. This is demonstrated by the lack of

```
class Teller(log:Log) {
  var transactions = 0

  def transfer(from:Account, to:Account, amount:Int) {
    println("About to transfer " + amount)
    atomic{
      from.withdraw(amount)
      to.deposit(amount)
      log.value = log.value + amount
      transactions = transactions + 1
    }
    println("Transferred " + amount)
  }
}

class Account {
  var value = 0

  def deposit(amount:Int) {
    value = value + amount
  }


  def withdraw(amount:Int) {
    value = value - amount
  }
}
```

Fig. 14. Our example using parser modifications. Finally there is complete control of the syntax and no need to import libraries, but this is at the expense of modifying the parser and adding new keywords into the language.

implementations at the end points. While we have an opinion on the location of the optimum point and have targeted our STM implementations accordingly, time will tell exactly where this point is.

Currently, the two most common transactional memories for Scala are Multi-Verse, which is included as part of the Akka platform [1], and ScalaSTM, which is being aimed at inclusion in the Scala standard library. These two STMs represent two differing philosophies to implementing transactional memory: on the one hand, modifying the runtime environment for clean code and strong semantics, and on the other, accepting the weaknesses and restricted use of transactions in exchange for the ability to operate without modification to the runtime environment. The remaining STMs have a relatively small uptake; however, Deuce STM for Java, which MUTS is based on, is relatively well used. The third philosophy, to modify the language and both the Java and Scala compilers, thus far has no traction, and as discussed is unlikely to, as it is both more invasive and less user friendly than the byte-code rewriting approach.

It is the authors' opinion that, while ScalaSTM is relatively popular, it is too constrained to act as a general-purpose transactional memory for Scala in the long term. For this reason, we expect that extensions to the runtime system such as those offered by Multi-Verse and MUTS will provide transactions for Scala in the long term. However, we would also expect that support for transactions in the compiler and language directly, while unlikely to happen in many existing languages, is likely in new languages as transactional memory becomes more ubiquitous.

Acknowledgments

The authors would like to thank the European Community's Seventh Framework Programme (FP7/2007-2013) for funding this work under grant agreement no 249013 (Teraflux-project). Dr. Luján is supported by a Royal Society University Research Fellowship.

```
                       var context$TM:Context = null;
                       var transaction_Variables_Backup = transaction_Variables
                       var committed$TM = false;

                       atomic_retry$1() {
                         context$TM = TM.beginTransaction();
                         try {
atomic
{                          Body
  Body    ⟹               committed$TM = TM.commit(context$TM)
}                        } catch {
                           case e:TransactionArea => ()
                           case e:TransactionException => ()
                           case e:Exception => { committed$TM =
                                                   TM.commit(context$TM);
                             if (committed$TM) throw e
                           }
                         }
                       };

                       if (!committed$TM) {
                         Transaction_Variables = transaction_Variables_Backup
                         atomic_retry$1()
                       }
```

Fig. 15. Transformation of a transaction block into existing Scala abstract syntax tree constructs. The code in italics is dependent on the environment that the specific block is called from.

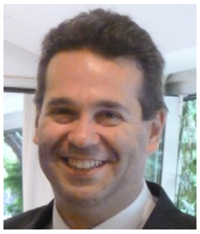
References

[1] Scalable Solutions AB, Akka project, February 2011. http://akka.io/.
[2] IBM BlueGene Team, The blue gene/*q* compute chip, in: HotChips 2011, 2011.
[3] R. Bird, Introduction to Functional Programming Using Haskell, second ed., Prentice Hall, 1998.
[4] F.W. Brasil, RadonSTM 2011. https://github.com/fwbrasil/radon-stm.
[5] N.G. Bronson, H. Chafi, K. Olukotun, CCSTM: a library-based STM for Scala, in: The First Annual Scala Workshop at Scala Days 2010, 2010.
[6] J. Chung, L. Yen, S. Diestelhorst, M. Pohlack, M. Hohmuth, D. Christie, D. Grossman, ASF: AMD64 extension for lock-free data structures and transactional memory, in: Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO'43, IEEE Computer Society, Washington, DC, USA, 2010, pp. 39–50.
[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, D. Nussbaum, Hybrid transactional memory, in: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS-XII, ACM, San Jose, California, USA, 2006, pp. 336–346.
[8] P. Felber, C. Fetzer, T. Riegel, Dynamic performance tuning of word-based software transactional memory, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP, 2008.
[9] D. Goodman, B. Khan, S. Khan, C. Kirkham, M. Luján, I. Watson, MUTS: Native scala constructs for software transactional memory, in: Scala Days 2011, 2011.
[10] T. Harris, K. Fraser, Language support for lightweight transactions, in: Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications, OOPSLA'03, ACM, New York, NY, USA, 2003, pp. 388–402.
[11] T. Harris, J.R. Larus, R. Rajwar, Transactional Memory, second ed., in: Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2010.
[12] Intel, Intel architecture instruction set extensions programming reference, 2012.
[13] G. Korland, N. Shavit, P. Felber, Noninvasive concurrency with Java STM, in: Third Workshop on Programmability Issues for Multi-Core Computers, MULTIPROG-3, Pisa, Italy, 2010.
[14] T. Lindholm, F. Yellin, Java Virtual Machine Specification, second ed., Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
[15] M. Odersky, L. Spoon, B. Venners, Programming in Scala: [A Comprehensive Step-by-Step Guide], first ed., Artima Incorporation, USA, 2008.
[16] N. Shavit, D. Touitou, Software transactional memory, in: Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, PODC'95, ACM, New York, NY, USA, 1995, pp. 204–213.
[17] The TERAFLUX project, 2010. http://www.teraflux.org.
[18] P. Veentjer, A. Philips, Multiverse 2010. http://multiverse.codehaus.org.
[19] C.T. Wu, An Introduction to Object-Oriented Programming with Java, second ed., McGraw-Hill, Inc., New York, NY, USA, 2000.

Daniel Goodman is a researcher at the University of Manchester, where he is investigating programming models for high-productivity languages on chips containing up to 1000 cores. His interests include programming abstractions for multi-core and heterogeneous processors, and GPU computing. Prior to this role, he developed tools and libraries to simplify the programming of novel hardware ranging from heterogeneous CPU–GPU systems to Japan's K supercomputer. He has a B.A. and Doctorate in Computer Science from the University of Oxford, where he developed the Martlet workflow language for distributed data analysis. His work can be seen at http://apt.cs.man.ac.uk/people/dgoodman.

Behram Khan is a Research Associate in the School of Computer Science at the University of Manchester. He gained his Ph.D. in Computer Science in 2009 from The University of Manchester. Behram's research interests include many-core architectures, and Software and Hardware Transactional Memory. His contact email address is khanb@cs.man.ac.uk.

Salman Khan received his Ph.D. degree in 2010 from the University of Edinburgh for work on improving the energy efficiency of thread-level speculative parallelization. He is now a postdoctoral Research Associate at the School of Computer Science at the University of Manchester. His research interests include parallel architectures, optimizing compilers, parallel programming, and the application of machine learning techniques to these areas. Information about his current research activities can be found at www.cs.man.ac.uk/~khansa.



Mikel Luján is a Royal Society University Research Fellow in the School of Computer Science at the University of Manchester. His research interests include managed runtime environments and virtualization, many-core architectures, and application-specific systems and optimizations. Luján has a Ph.D. in computer science from the University of Manchester. Contact him at mikel.lujan@manchester.ac.uk.

Ian Watson is Professor of Computer Science at the University of Manchester. His major interest is in the architecture of general-purpose parallel computers with particular emphasis on the development of machine structures from an understanding of the requirements of language and computational model issues.

His recent interests are in the area of multi-core systems, with particular emphasis on approaches to achieve high extensibility using a combination of Dataflow techniques together with transactions to provide a clean approach to introducing the handling of shared state.