Nesoi: Static checking of transactional coverage in parallel programs

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

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

Abstract

In this paper we describe our implementation of Nesoi, a tool for static checking the transactional requirements of a program. Nesoi categorizes the fields of each instance of an object in the program and reports missing and unrequired transactions at compile time. As transactional requirements are detected at the level of object fields in independent object instances the fields that need to be considered for possible collisions in a transaction can be cleanly identified, reducing the possibility of false collisions. Running against a set of benchmarks these fields account for just 2.5% of reads and 17-31% of writes within a transaction. Nesoi is constructed as a plugin for the Scala compiler and is integrated with the dataflow libraries used in the Teraflux project to providing support both for conventional programming modes and the dataflow + transactions model of the Teraflux project.

Keywords: Scala, Static Checking, Transactional Memory, Dataflow

1. Introduction

The move from single threaded processors to multi-core processors means that constructing concurrent programs has become a problem all programmers must face. Furthermore the move from conventional multi-threaded HPC programs to general purpose parallel programming means that the nature of

*Corresponding author
Email address: `manchester@djgoodman.co.uk` (Daniel Goodman)

the programs that must be constructed has become more complex. Many approaches have been proposed to help address this; one of these is to use transactional memory [1] to provide protection against race conditions. The use of TM only works if all the mutable shared state is protected by transactions. At the same time transactions have a cost and it is important to keep their use to a minimum.

In this paper we describe Nesoi, a plugin to the Scala compiler for performing compile time static analysis of the source code of Scala [2] programs to determine both where it is necessary to include transactions, and which values within a transaction must be treated as transactional. We also describe how Nesoi has been integrated with the dataflow programming model as part of the Teraflux project [3]. Nesoi's analysis serves three purposes:

- Firstly it informs the programmer if they have failed to include transactional protection where it is required. An example of how some required transactions can be hard to spot is shown in Figure 1.
- Secondly it allows the detection of transactions that are surplus to requirement.
- Thirdly it allows the transactional mechanism to only check for conflicts on the variables that could have conflicts with other transactions, while still recording information on the remaining variables in the transaction in case it is necessary to roll back the transaction. This can reduce the number of read and writes that have to be tested against by 97.5% and 69-82% respectively, reducing false conflicts and/or inter process communication.

As Nesoi performs its analysis as part of the compilation process of a Scala program its output is independent of the transactional memory being used. As such it can be used to inform both software transactional memories (STM) and hardware transactional memories (HTM).

1.1. Contributions

This paper contributes the following:

```
class NodeFactory {
  Node getNode(int id) {
    Node n = new Node();
    n.name = id; // This is an error
  }
}

class Node {
  static Set[Node] nodes = new HashSet();

  int name;

  Node() {
    atomic {
      nodes.add(this);
    }
  }
}
```

Figure 1: Method `getNode` is an example of a piece of code that at first glance looks to only be handling local data, but in practice requires protect against race conditions for the assignment `n.name = id;` because the constructor makes all instances of Node visible to all threads. Java is used for this example instead of Scala to aid reader familiarity.

- A detailed description of the implementation of Nesoi.
- A detailed description of how this is integrated with the Dataflow + TM model of the Teraflux project.
- An examination of how Nesoi combined with the Dataflow + TM model provides the potential to automatically insert transactional protection.
- An evaluation of Nesoi on a selection of test cases.

Nesoi currently provides the detection of errors, the detection of unnecessary transactions and outputs a list of transactional variables. That is variable that can conflict with other transactions. There are multiple possible uses of this additional data ranging from automatically inserted transactions in Dataflow + TM models, to improved performance in TM through the removal of unnecessary logging. However, the implementation and evaluation of tools and libraries that take advantage of this third output is outside the scope of this work for three reasons: Firstly it is unnecessary for Nesoi to perform its primary functions of error detection and field identification meaning it would only serve as a distraction from Nesoi; Secondly the set of possible uses is not exhaustively known, so any exploration would be incomplete; Thirdly, the performance effects will depend on the specifics of the application being used as benchmarks and the implementation of the TM, both of which are orthogonal to the design and implementation of Nesoi.

The rest of this paper is structured as follows: We will briefly describe what Transactional Memory is; We then describe the classes Nesoi categorizes memory into before describing Nesoi in detail; Next we introduce the dataflow and transactional memory model developed as part of the Teraflux project and describe how Nesoi handles this extension; We then discuss Nesois limitations and the potential to provide not only static checking of correctness, but also provide productivity and performance gains. Finally we conclude by discussing future and related work.

1.2. Transactional Memory

Transactional memory [1] is an alternative to conventional locking. Instead of using locks, segments of code that must occur atomically are marked as transactions. These transactions are then executed optimistically, that is they are executed in the hope that they will not be executed at the same time as another transaction that conflicts in its accesses or modifications to state. Such overlapping behaviour between transactions is known as a collision. To protect against collisions with other transaction, sufficient information is recorded by the transactional memory to allow one of the transactions to be rolled back and retried if a collision occurs. An example of this can be seen in Figure 2.

Transactions memory systems are implemented so that only the thread executing a transaction is able to observe any of the effects of the transaction prior to the transaction successfully completing, and at the point the transaction does complete all the modifications to the system state made by the transaction will be applied to the system as a single atomic update. In the event that a trans-

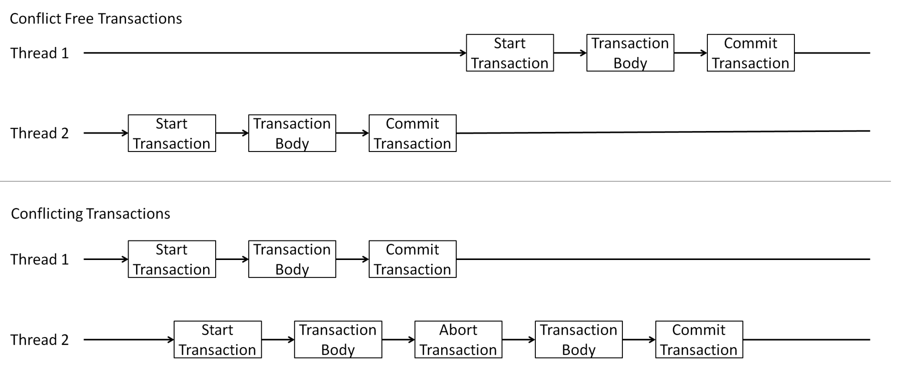

Figure 2: An example of the two threads executing a transaction on the same piece of data. The transaction is made of three parts, a start, a body where the transaction is computed, and a commit where any changes are saved back to the system or, if the transaction has been conflicted with, an abort where the changes are discarded and the transaction restarted. In the top example the transactions do not overlap and so both commit successfully. In the lower example there is a conflict and causing the second thread's transaction to abort and retry.

action conflicts and is rolled back, there is no mechanism by which any thread can observe that the transaction previously attempted to execute.

Transactional memory provides three basic constraints from which transactions can be reasoned about:

- **Atomicity** Transactions appear to other transactions as if they occur in a single instant, or do not occur at all.
- **Consistency** Transactions take the system from one consistent state to another.
- **Isolation** Transactions act in isolation of each other.

2. Description of Memory Model

For this analysis performed by Nesoi we assign one of three states to each piece of memory in the program: Local, Global, and Transactional. These states form a total order with Local < Global < Transactional and are defined as follows:

Local Memory is memory that is only accessible to the thread that allocated it. For example objects that are created as method variables.

Global Memory is memory that has escaped its allocating thread and can be accessed by multiple threads. This can occur by a globally visible object being assigned a reference to the memory, or by the memory being passed to another l thread.

Transactional Memory is globally accessible memory that is modified after it has become globally accessible.

The initial category of each piece of memory and the points where the memory changes type can be statically determined by Nesoi. We can further trace all the possible paths that the memory could take from its creation and determine if it traverses any of the points where a state change occurs. Once a piece of memory has changed to a higher state it cannot revert back to a lower state, and only once memory has become globally accessible can it become transactional.

3. Description of Nesoi

Nesoi is implemented in Scala [2] as a plugin for the Scala compiler. Any errors detected by Nesoi are reported to the compiler which will then return a compilation error. The output of Nesois analysis can be passed to further compilation phases, and can also be output from the compiler to be used by later tools if required.

3.1. Scala

Scala is a general purpose programming language designed to smoothly integrate features of object-oriented [4] and functional languages [5]. By design it supports seamless integration with Java, including existing compiled Java code and libraries. The compiler produces Java byte-code [6], meaning that Scala can be called from Java and vice-versa. Scala is a pure object-oriented language in the sense that every value is an object. Types and behaviour of objects are described by classes and traits, and classes are extended by sub-classing 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 power is furthered through the provision of a lightweight syntax for defining anonymous functions, support for higher-order functions, the nesting of functions, and for currying. Scala's case classes and its built-in support for pattern matching and algebraic types is equivalent to those used in many functional programming languages.

Scala is statically typed and equipped with a type system that 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.

The Scala compiler is pluggable and can be extended by the construction of modules that can be added to the compiler between any two existing modules in the compilation path. Using this functionality Nesoi extends the Scala compiler to identify and statically check for the correct use of transactional state and transactions.

3.2. Plugin Structure

Nesois execution is split into two phases: The first phase parses the abstract syntax trees (ASTs) and builds up descriptions of all the classes and methods in the program; The second phase detects any main methods in the program and using these as starting points, traverses the possible executions of the method ASTs to determine if there are any reads or writes to data that need to be protected by transactions, or any transactions that are not required. An example of these phases and the resulting data structures can be seen in Figure 3.

3.2.1. Class AST Traversal

The Scala compiler constructs an independent AST for each class it compiles. The part of these that represents the structure of the class and its contained methods is traversed by Nesoi's first phase to create a set of data structures that represent the classes of the program. These data structures are:

Object Description. An object description is a structure to describe a class of object so that *Object Instances* of the class of object can be constructed in

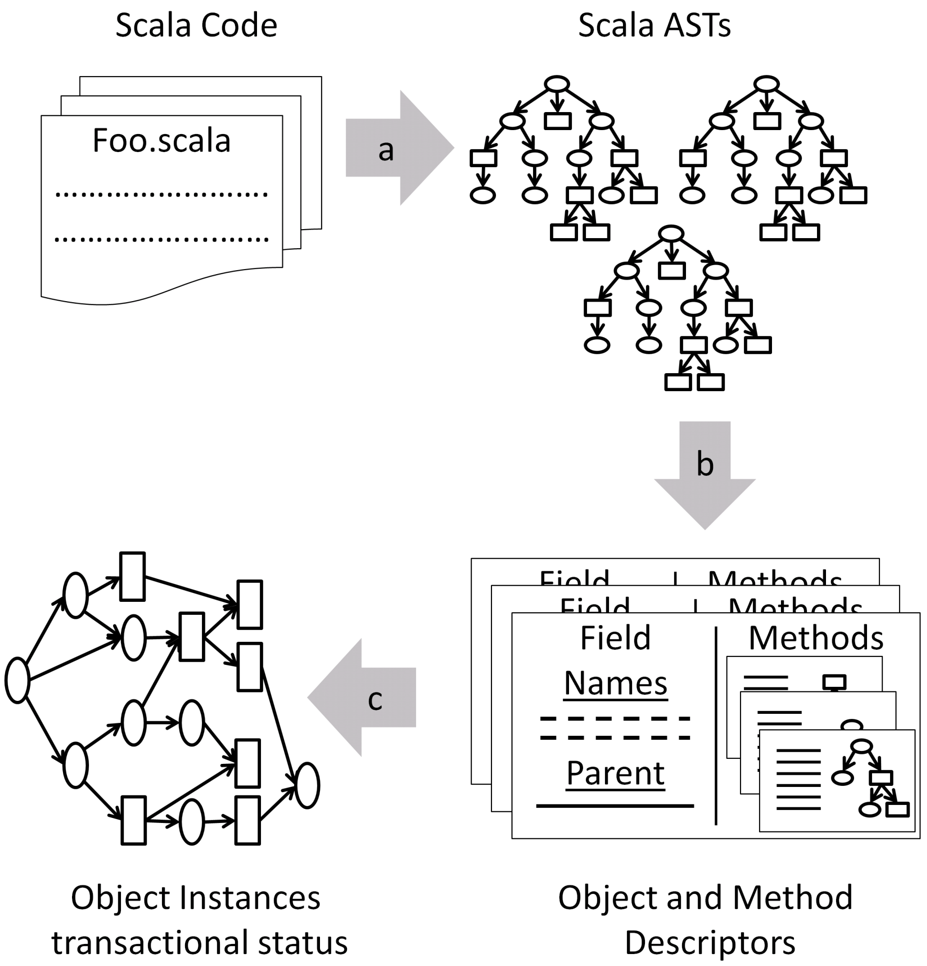
Scala Code
Scala ASTs
Foo.scala
a
b
c
Field
Names
Parent
Methods
Object Instances transactional status
Object and Method Descriptors

Figure 3: An outline of the different phases of the analysis. a) The first part of the Scala compiler turns the input files into ASTs and performs type checking etc. b) Phase one of Nesoi parses the trees and constructs class and method descriptors. c) Phase two statically analyses the methods in the class descriptors to detect which state needs to be transactional by generating a DAG of object relations. From this process it produces a list of errors and warnings.

the second phase. The object descriptor contains a list of the fields instances must contain, a set of method descriptions that represent the methods that the object contains, and a reference to the description of the objects parent. Type information is known to be correct as this phase is after the type checker, so is ignored with the exception of the method calls and signatures where it is required for correct matching and the type of the object which is required for initial construction.

Method Descriptions. Each method in the ASTs has a method description constructed to represent it. This contains the name of the method, the name of each of its arguments, and the AST for the method itself. This information is used in the second phase to construct *Method Instances* which are used to evaluate each call of the method. In addition to method descriptions constructed from the AST, hard coded versions are used to add some library methods as discussed in Section 3.6

3.2.2. Method AST Traversal

Having constructed a complete description of the classes and packaged up the ASTs of the methods, the second phase determines for every new object created by a program which of its fields are transactional, and when these fields are accessed. From this it is possible to detect if transactions are correctly placed and record which fields need to be monitored within a transaction.

To do this the first class descriptors that contain static main methods are identified and these methods are used as a starting point to walk all the ASTs of all the methods that can be reached from the main methods. Each time a method is to be evaluated a method instance is constructed to hold all the relevant data.

Method instances come in two forms, hard coded method instances used to add some specific library functions as described in Section 3.6 and Method instances that contain an AST from the Scala compiler. Both types contain an index between the input argument names and the passed object instances, and a reference to the object instance to which the method belongs, the `this` object.

For evaluation the first of these contains some custom code that determines the response of the function call, the second traverses the AST constructing new object instance, accessing local object instances, fields and global state as required by the nodes in the tree. On meeting calls to other functions, these are evaluated and resulting reference nodes returned before this method invocation proceeds.

The behaviour of the method instance is dependent on the methods in any objects instances it is able to reach through its arguments. Such passed objects include all object instances reachable from the argument. This is a potentially huge set making generating an index based on type information impossible for sensible space and time constraints. As a result it is necessary to traverse each function each time a call site is encountered In theory the worst case time complexity of evaluating each function every time it appears is exponential with the number of call sites in the program, however in practice programs are rarely constructed in anything approaching this case and the execution time is a low order polynomial.

It is important to note that this walk of the tree is done at compile time and is not the same as a walk done by an interpreter because it does not store values or evaluate expressions. So for example a while loop may execute multiple times, but will only have its body traversed once in this walk, and branches have both the if and the else sub trees traversed. As such a single object instance created during the walk represents the set of possible objects that can be constructed from a given statement in a given calling context.

By walking the ASTs Nesoi constructs a directed acyclic graph (DAG) of object instances and reference nodes that represents the possible linking between object instances in the program. This DAG allows changes to the visibility of one object to be propagated to other object instances as shown in Figure 4.

Object Instances. Object instances are constructed from the object descriptions and represent the set of objects constructed at a given location in a given function call. They contain a set of reference nodes that represent the fields of the specific instance of the object, and meta-data marking the object instances visibility and for instances of dynamic objects recording where it was created.

For dynamic objects multiple instances can be constructed throughout the program, for static objects a singleton instance of the object is automatically constructed the first time the object is accessed and is stored for all further accesses. While there can be multiple instances of the same type of dynamic object instance, only one instance is created per new statement in a method per time the method is evaluated.

Reference Nodes. Reference nodes are the data structures used to represent references in the program, they also contain meta-data about the reference describing if it can only be used in a transaction. However unlike a program execution where a reference can only point to a single location at a time, reference nodes address the set of object instances and other reference nodes that collectively represent the set of object instances that a reference could possibly address. Examples of why reference nodes may reference more than one object instance include the reference being a global static value in a multi-threaded program, in which case all the references assigned to the global value must be maintained as the order of assignments and reads is unknown, and braches, where as we are unable to determine the value of the branch condition, we evaluate both the if and the else branches, both of which may assign different values to the reference. In addition to being used for fields in classes, reference nodes are also used to represent local values in methods, and the arguments passed to and returned by methods.

3.2.3. Arrays

As Nesoi is not an interpreter we do not compute the value of expressions. This means that array indexes are not known when accessing arrays. Instead a cumulative set of all the object instances assigned to the array is stored, and returned when the array is accessed. This is not typically a large number of instances as arrays are typically populated by multiple iterations of a loop, and Nesoi will only evaluate each loop body once per method evaluation.

3.3. Detecting Transactions

There are many ways of adding transactions to Scala [7], but all of them either add in method calls or AST nodes that can be detected. We used the software TM MUTS [7] and have also implemented a stub version of this that integrates with the Teraflux hardware TM [3]. Every time a begin transaction is encountered, it is detected and a counter is incremented and when a commit transaction is detected the counter is decremented. If the counter is non zero then the evaluation is currently in a transaction. A counter is used instead of a Boolean value to allow for nested transactions.

3.4. Determining Memory Types

All object instances are either global or local memory, with the singleton static object instances initialized as globally visible and the dynamic objects instances initialized as local. Objects instances with a memory state of local become global if an object instance with and an existing reference to it becomes global, or a field in an object instance with global state has is updated to reference it, examples of this can be seen in Figure 4.

Each field within an object instance inherits the visibility of its encompassing instance, once its parent instance has become globally visible, the individual fields will become transactional if they are subsequently written to. As the traversal is aware if it is currently in a transaction it is able to detect if a write to a globally visible field occurs outside of a transaction. If this occurs an error is signalled for the compiler to report.

Handling of reads is slightly more complicated as reading data must only occur inside a transaction if the data being read is transactional, i.e. is modified elsewhere in the program after it has become globally visible. As several reads could have occurred before this happens, fields are required to keep a record of all reads that occur outside of transactions after becoming globally visible. In the event that a write then marks the field as transactional all these recorded reads will be reported as errors. If this were to take too much memory keeping

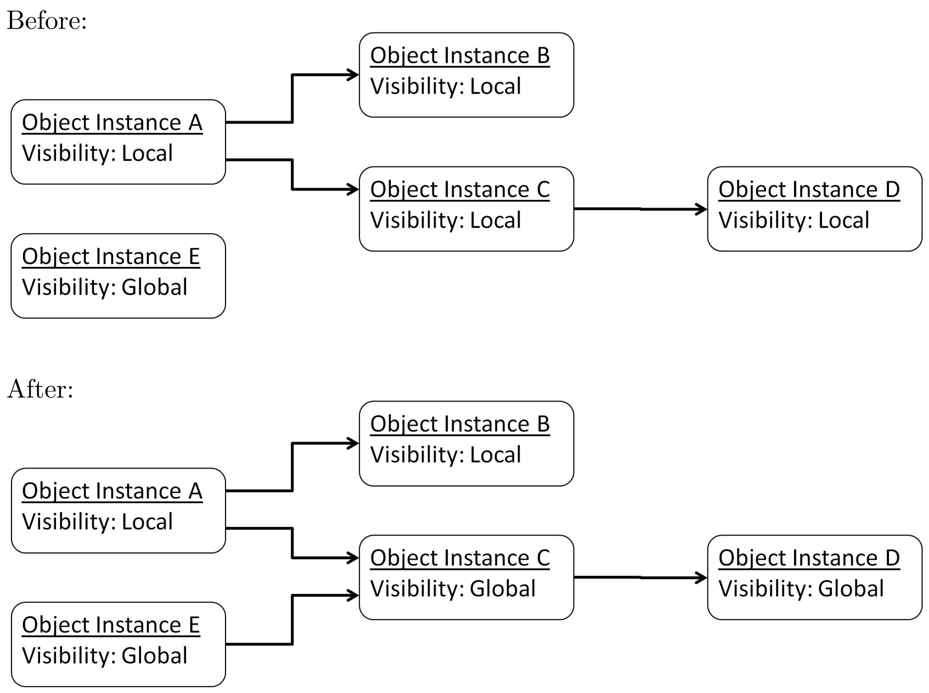

Figure 4: An example of how visibility propagates between object instances. Initially instances A, B, C and D are local, then a field in object instance E is linked to instance C. This linking causes instance C to change to globally visible, this change is then propagated to instance D. As there is not a path from a globally visible instance to either instance A or instance B these remain locally visible.

just one read would be sufficient, but would mean that only one error per field could be reported.

It is important to note that while global and local visibility is handled at the object level, transactional errors are detected at the granularity of individual fields. Handling the transactional state of fields individually instead of at the object level prevents transactions propagating to locations where they are not required. For example, if a program has a global object containing a large amount of constant data describing the configuration of the application, or the input data, and also includes a single field acting as a shared counter, then the counter must be transactional. If the object was marked as transactional then all reads of the large quantity of constant data would also be forced to be transactional despite the data being constant.

3.5. Recursion

As Nesoi explores all possible paths through the code without evaluating guards, a simple program containing recursion such as the naive calculation of Fibonacci could prevent the checker from terminating.

```
def fib(n) = if(n<2) 1 else fib(n-1) + fib(n-2)
```

To address this we test for recursions in two places, one to detect recursions in conventional code, and one to detect recursions in threads which we discuss in Section 3.7.1.

For conventional code we keep a stack of all the function calls in the current walk, complete with the object instances for each of the arguments. If the walk encounters an instance of a function called from the same location as an existing function, taking a subset of the object instances given to the pervious function as arguments then we deem this to be a recursion, and do not evaluate the method. If a return type is required, a new object instance is created and returned to satisfy this. To determine if one set of object instances is a subset of another set an equivalence operation is required, this is implemented by comparing the type of the object instance and the location where each object instance was constructed.

3.6. Library Functions

Any real program will include library calls and these must be handled. The simplest suggestion would be to include the libraries code in the compilation, however aside from the size of the libraries potentially being orders of magnitude bigger than the program being compiled there are some more challenging complications. Scala can call into Java which is not parsed by the Scala compiler, yet Scala's libraries make extensive use of Java libraries. The Java libraries then in turn use JNI to call into native compiled libraries. In addition, base types in Scala are represented as objects and library calls, these are then converted to native instructions in the final stage of compilation, and as such no bodies exist to compliment these method calls. To add in the libraries two approaches were taken.

First to address the Scala and Java libraries, stub codes were created. These codes are generally very simple as they do not have to work out specific values that should be returned or stored, but instead simply return all possible values. Often this can simply be achieved by returning either a new object instance, or in the case of operations on collections all the previously seen object instances. Because of the semantics of arrays, the use of a single cell array in the stub code suffices for this. Once the compilation has been completed, any classes generated by the stub code can be discarded, causing the resulting user code to default back to using the standard Scala libraries.

Second, to handle operations on base types, method descriptions and complimentary method instances for some operations such as addition on `Int` are added to the set of method descriptions when the tool initializes. The resultant method instances are typically extremely simple as the returned object from an addition on base types is a new instance of a base type.

3.7. Handling Threaded Code

To handle conventional threading we detect any object instance that extends `Runable` or `Thread`, and if `start` is called on this object instance, or it is passed into a thread pool, the instance is marked as global and the `run` method is scheduled for analysis.

3.7.1. Recursion

Recursion in threads occurs when a thread is able to spawn new threads to run the same code. The detection mechanism is very similar to that of standard recursion, though instead of keeping a stack of function calls in the current evaluation, we keep a history of all threads executed at anytime. We also replace the function call site with the thread creation site. If the creation of a thread is determined to be part of an infinite recursion we just mark the thread not to be analyzed. While effective this approach is cleaner when working with dataflow threads for which the arguments to the thread are well defined, as discussed in the next section.

As discussed in Section 5.1 Nesoi is unable to detect points where no conflicts exist because the results of evaluated expressions guarantee separation. Conventional threading can include complex interactions with complex synchronization relationships between threads. These are not present in the Dataflow and Transactions model presented by the Teraflux project even with the presence of transactions, as such dataflow programs are more amenable to analysis. We will now look at the dataflow plus transactions model before describing how this is integrated into Nesoi.

4. Dataflow

Dataflow programming is a means of splitting a program into a number of deterministic side effect free pieces. The execution of the program is orchestrated through the construction of a directed acyclic graph where the nodes are the sections of computation and the vertices are the data dependencies between sections. An example of this can be seen in Figure 5. Once all the parents of a given node have been computed all the inputs to the node will have been generated and the node can be scheduled for execution. As such the execution of the program is controlled by the flow of data through the graph. This differs from imperative programs where control is explicitly passed to threads requiring the programmer to determine when all the inputs will have been constructed and the code is safe to execute. To enforce this, once a piece of data has been constructed it remains immutable for the lifetime of the program. Depending on the granularity of the program the nodes vary in size from a single instruction to whole functions including calls to other functions.

Dataflow programming has been shown to be very effective at exposing parallelism as the side-effect-free nature means nodes whose inputs have been generated can be executed at anytime, regardless of what is happening elsewhere in the graph, and without affecting the nodes result which is deterministic. Dataflow programs are by definition free from deadlocks and race conditions, which simplifies the task of constructing parallel programs.

The explicit description of when results must be passed allows for a weakening of the cache coherency requirements of the system. Instead of maintaining coherence at the instruction level it is now only necessary to ensure that results from one node are written back to main memory before the nodes that depend on these results begin executing. The determinism of dataflow is both an advantage and a limitation which has to be overcome with carefully applied shared mutable state. For example without shared mutable state parallel interactive systems such as a seat reservation system cannot be implemented. To overcome this Teraflux has combined transactional memory with dataflow as the property of a dataflow task to execute as an atomic piece whose inputs are immutable during its execution makes dataflow tasks similar in appearance to transaction to memory. Both have a constant view of the world while executing and only make data visible after completion. As such a transaction in a dataflow task can be seen as three dataflow tasks to the memory system as shown in Figure 6.

4.1. Supporting Dataflow

Nesoi was developed as part of the Teraflux project to assist in the construction of dataflow programs with transactional memory. We'll now describe how Nesoi handles dataflow threading and is able to take advantage of the clearer

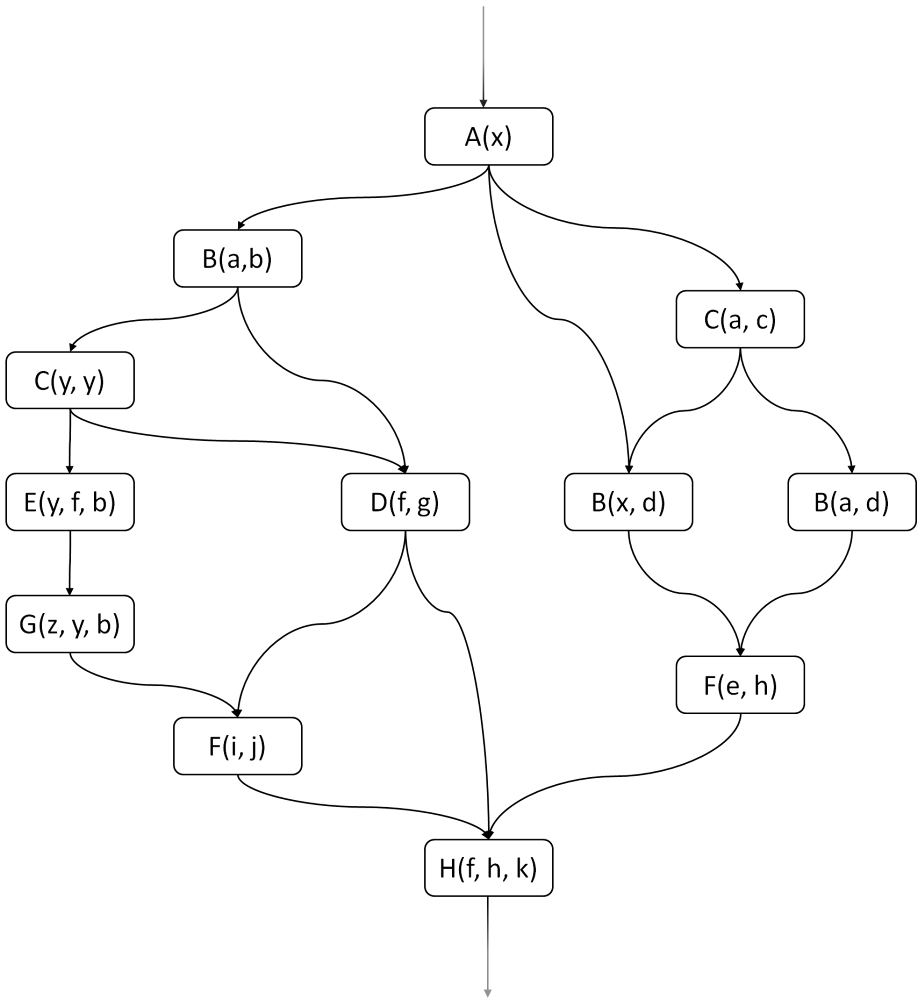

Figure 5: An abstract example of a pure dataflow graph demonstrating the functional nodes and the dependencies between them.

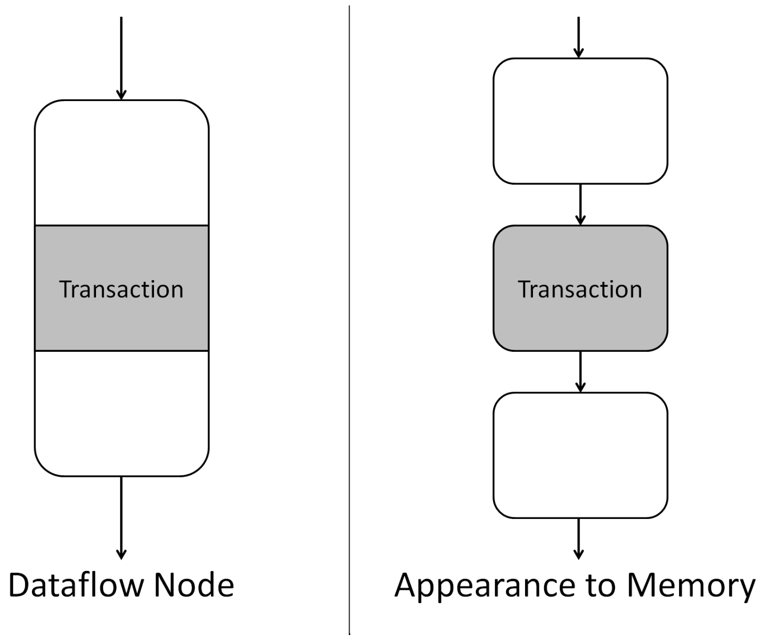

Figure 6: On the left a task in the dataflow graph containing a transaction, on the right how this can be viewed as three tasks by the memory system, with all inputs remaining constant from the start of the task and all outputs only appearing as the end of the task.

semantics of memory passing and thread interaction.

Dataflow Threading. Nesoi detects the creation of threads when it is walking methods and queues these to be examined after it has finished with the current thread. As data can only move between threads through explicit passing or transactions, the order in which threads are examined does not affect the result.

Dataflow threading provides a new way for an object to achieve global visibility. When an object is passed to a dataflow thread as an argument it becomes globally visible at the point that the thread starts. As such when we start to examine a thread we mark all its arguments as global. In addition, because closures are being used to describe the function being executed we also mark the object representing the closure as global to prevent object instances leaking between threads unmarked through this mechanism.

Dataflow execution is provide through the use of the software dataflow library DFScala [8]. The project ultimately aims for a compilation route through to native hardware, and this change will result in the library being replaced by a library of inline-able functions that maintain the same interface, but take advantage of the new capabilities. As such the techniques described here are valid regardless of the execution environment, and indeed the DFScala instance that the code is compiling against is just stub code as described in Section 3.6.

As dataflow threading is handled through library calls, it is possible to identify all invocations of these calls and add in the required extra logic to extend Nesoi. This logic falls into two categories, the creation of object instances to represent the threads, and the detection of passing arguments to threads.

When functions to create threads are detected, the number of arguments, and the resultant thread object instance are recorded in a lookup table. This table is then used to store the object instances that represent the possible arguments to the thread as they arrive. These argument values are set by detecting the calls to set the threads arguments. Once sufficient arguments to analyze a thread have been collected it is marked for evaluation, and placed in a queue to be evaluated at some point after the Nesoi has finished walking the current thread.

	Evaluated					Stored				
1)	Thread X	A,B	D	G	H	Thread X	A,B	D	G	H
2)	Thread X	A,B	D	E,F	H	Thread X	A,B	D	E,F,G	H
3)	Thread X	A,B	C	E,F,G	H	Thread X	A,B	C,D	E,F,G	H

Figure 7: An example of the re-analysis of a dataflow thread after the addition of a new argument value. Thread X takes four arguments. 1) is the first analysis after receiving a value for each of those four arguments. Note that the first argument receives two different objects instances. At 2) two more object instances are detected for the third argument and the threads function is reanalyzed using the existing instances for arguments 1, 2, and 4, and the new object instances are used for argument 3. The super sets of all these arguments are then stored for future analysis and recursion checking. At 3) a new object instance is passed for argument 2 and the thread is analyzed again, using the previous object instances for arguments 1,3, and 4, including the object instances that arrived at 2).

As all paths are being explored, it is common for a given dataflow argument to be set multiple times. When this occurs, the thread in question is looked up, and if it is yet to be analyzed the extra arguments are just added to the ones it is already has. If the thread has already been analyzed then it is set for a further analysis, taking all the object instances referenced for the other arguments, and just the new object instances for the argument that has had the addition. This addition is then added to the previous arguments in the lookup, so that is will be included in any further analysis if other arguments are assigned to a second time. An example of this can be seen in Figure 7.

5. Evaluation

Nesoi is able to divide all read and write accesses into categories of transactional or non-transactional. We will now consider its limitations and scalability, as well as how its output can be extended beyond ensuring the correctness of programs to enable better performance and simpler programming models.

5.1. Limitations

The main limitations of this approach are that as guards and index expressions are not evaluated Nesoi is unable to detect situations where threads are accessing the same array, but program logic maintains a separation between the threads accesses. This can be overcome by the addition of user annotations allowing them to indicate that a transaction is not required at a specific location. We favour this being on a location by location basis as this would allow the mode of access to an object to change through the program without weakening the effectiveness of the tool. However we anticipate that such instances would be rare outside of library code such as Scala Parallel Collections [9].

In the same vein, when working with conventional threads it is not possible to determine the synchronization relationships between the threads, and the protection this may offer to shared state. This is less true for dataflow threads as they have better defined interactions.

State that is shared between dataflow threads may be modified in a dataflow thread that through the construction of the program is guaranteed to be the sole thread accessing this area of memory at that time. This is a concept known as Owner Writable Memory (OWM) [3]. This problem is more addressable as our analysis includes detecting calls to pass arguments between threads and which thread the argument is passed to. However, it is not possible to address this in all cases as it is relatively easy to construct an example where only one thread will be created and executed, but without evaluating the guards on branches it is not possible to determine this. This can then be furthered to make the code depend on invariants of the input data meaning that when adding OWM to programs annotation of the code not further evaluation would be required.

5.2. Scalabillity

As discussed in section 3.2.2, because each method AST is evaluated every time it is met, if a program is constructed such that method n only calls method $n-1$ x times for all the methods in the program then the time complexity of Nesoi is $O(x^N)$ where N is the number of methods in the program. However, as real programs are almost never constructed like this the time complexity for real programs is manageable polynomial. This is especially true when it is remembered that this is a one off cost at compile time, not a runtime cost.

5.3. Performance Gain

Running Nesoi against a selection of test cases including a dataflow version of Lee's Algorithm [10] we measured that only 2.5% of the reads and 31-18% of the writes encountered in transactions were transactional with the remainder being either local or global constants such as loop indices, and multi dimensional arrays where only the cells in the lowest level array are modified. While these numbers do not correlate directly into percentages of reads and writes when the program is executed as Nesoi evaluates the body of a loop exactly once, while a real application may iterate round for many iterations, or not at all. However, they do show that this technique offers significant opportunities to reduce the number of values checked when determining if a transaction can commit. Reducing this reduces the number of entries in Bloom filters used to keep track of the reads and writes performed by the transaction. This reduces the chances of a false conflict, and/or the amount of inter-process communication. Both of these should equate to an improvement in the overall performance of the application. As such performance gains are highly dependent on the implementation of the transactional memory in question, and the applications being used, surveying the breath of this effect across the range of applications and transactional memories is outside of the scope of this paper.

5.4. Automatically Inserted Transactions

Given that all transactional state can now be automatically detected the question that has to be asked is why not use this to automatically add in transactions. With conventional multi-threaded programs this is not possible as transactions have to span more than a single access to have any utility at all, indeed transactions that only perform a single read or write of transactional data can be removed by the compiler as an optimization. Given the transaction has to be larger the question then becomes how much larger, and this is the question that it is not possible to answer in conventional parallel programming. However with dataflow programming, the programmer provides additional information in the form of the dataflow threads, and it would be perfectly possible to automatically insert a single transaction into threads that access transactional state to cover all the transactional accesses within the thread.

However, while this approach has the potential to greatly simplify the problem of adding shared state, it also has the potential to effect efficiency. For example consider a loop updating every value in a large shared array that is being concurrently used by a large number of other threads. If each iteration of the loop is independent, then they could be placed in separate transactions, however with this strategy they would all be fused into a single transaction that would have a high contention and would form a bottleneck in the system. To overcome this, the loop could be broken into separate dataflow threads, but this would result in a large number of small threads which can turn the scheduler into a bottle neck instead. One approach to overcome this is to automatically insert transactions at the granularity of the thread unless the user has added transactions that divide this single transaction. In this cases then additional transactions would be automatically added to either side of the transaction to cover any remaining transactional state. An example of this can be seen in Figure 8.

6. Future Work

Our future aims are to add in detection required for OWM to the extent described in Section 5.1, and to integrate the output of Nesoi with the different compilation routes developed as part of the Teraflux [3] project. Specifically this will involve compiling down to the simulated Teraflux hardware and integrating with MUTS [7] to improve efficiency. Finally we would like to extend this work to experiment with the automatic insertion of transactions. described in Section 5.4.

7. Related Work

This work was initially inspired by work on compile time escape analysis in Java [11]. In this work the compiler constructs graphs of a methods execution

```
.....
int count = 0;
//Auto inserted transaction starts
for(int i = 0; i < a.length; i++)
{
  count += a[i];
  a[i] = 0;
}
GlobalState.progress += count;
//Auto inserted transaction ends
......
```

a)

```
.....
int count = 0;
for(int i = 0; i < a.length; i++)
{
  atomic {
    count += a[i];
    a[i] = 0;
  }
}

//Auto inserted transaction starts
GlobalState.progress += count;
//Auto inserted transaction ends
......
```

b)

Figure 8: An example of how inefficient cases with automatically added transactions could be overcome by the user adding in transactions at specific bottlenecks. a) The transactional segment of a thread which as part of its work reads, sums, and zeros an array of progress counters to update a global progress counter. As every other thread will also be using the array to report its progress, this transaction will be highly contentious. This could be overcome by allowing the programmer to provide additional detail about the bounds of the transaction to improve efficiency, and so lift some of the information out of the auto inserted transaction. An example of this is shown in b).

to determine if objects escape their constructing method or their constructing thread. Objects which do not can be allocated onto the stack instead of the head or have their concurrency protection removed as optimizations. Our work extended this to handling transactional memory and adds understanding of dataflow graphs as well as addressing issues with inheritance caused by the single pass of the graph performed in this work. More recent work in this vein addresses issues such as the need for all the library code to be available [12], formalization of the properties of such methods [13] and the use of such analysis to improve garbage collection performance on multi-threaded systems [14].

Tools such as T-Rex [15] provided runtime checking of race conditions in transactional code written in C and C++ code. These have the advantage that they avoid false positives caused by Nesois static checking as they can evaluate expressions. However, this only works if the data that they are running with is truly representative, so they can be used for debugging, but are less effective at guaranteeing the absence of bugs. They also have the disadvantage that they are only run for debugging due to the overhead of running them with the program.

Previous work has also looked at the use of user defined separation of memory instead of automated separation. Such separations can be used to help ensure the correctness of transactional programs. In these models users explicitly state when a variable is going to be accessed without protection, and when the transactional region starts again [16]. This work has much in common with the Owner Writable Memory developed as part of the Teraflux project [3] and our intention to encompass this case through the use of annotations.

8. Summary

In this paper we have presented Nesoi a tool for analyzing Scala programs constructed as part of the Teraflux project to perform static checking of the placement of transactions in both dataflow and conventional code. This tool is able to detect which state in a program is transactional and if it is being correctly accessed, providing compiler errors and warnings when transactions fail to be included or are included unnecessarily respectively.

In addition ensuring transitionally correct programs, having generated this data Nesoi is able to pass the identification of transactional reads and writes to other applications. We discussed some ways that this data could be used. These uses ranged from improving the performance of transactions by reducing the number of potentially conflicting reads and writes that have to be recorded to automatically inserting transactions into dataflow code. The latter being something that has the potential to greatly simplify parallel programming. Programmer inserted transactions would then present a means to overcome any specific performance bottle neck resulting from this, not a necessity for correct code. As a result bugs from missing transactions would affect performance, not the program generating a correct result.

Acknolagements

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.

References

[1] T. Harris, J. R. Larus, R. Rajwar, Transactional Memory, 2nd edition, Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2010.

[2] M. Odersky, L. Spoon, B. Venners, Programming in Scala: [a comprehensive step-by-step guide], 1st Edition, Artima Incorporation, USA, 2008.

[3] M. Solinas, R. M. Badia, F. Bodin, A. Cohen, P. Evripidou, B. F. Paolo Faraboschi, G. R. Gao, A. Garbade, S. Girbal, D. Goodman, B. Khan, S. Koliai, F. Li, M. Lujan, L. Morin, A. Mendelson, N. Navarro,

A. Pop, P. Trancoso, T. Ungerer, M. Valero, S. Weis, I. Watson, S. Zuckermann, R. Giorgi, The teraflux project: Exploiting the dataflow paradigm in next generation teradevices, in: Proceedings of the 2013 Euromicro Conference on Digital System Design, 2013, pp. 272–279.

[4] C. T. Wu, An Introduction to Object-Oriented Programming with Java, 2nd Edition, McGraw-Hill, Inc., New York, NY, USA, 2000.

[5] R. Bird, Introduction to Functional Programming using Haskell, 2nd Edition, Prentice Hall, 1998.

[6] T. Lindholm, F. Yellin, Java Virtual Machine Specification, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[7] D. Goodman, B. Khan, S. Khan, M. Luján, I. Watson, Software transactional memories for Scala, Journal of Parallel and Distributed Computing 73 (2) (2013) 150–163. doi:10.1016/j.jpdc.2012.09.015.
URL `http://dx.doi.org/10.1016/j.jpdc.2012.09.015`

[8] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Luján, I. Watson, DFScala: High level dataflow support for Scala, in: 2nd International Workshop on Data-Flow Models For Extreme Scale Computing (DFM), 2012.

[9] A. Prokopec, P. Bagwell, T. Rompf, M. Odersky, A generic parallel collection framework, in: Proceedings of the 17th International Conference on Parallel Processing - Volume Part II, Euro-Par'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 136–147.
URL `http://dl.acm.org/citation.cfm?id=2033408.2033425`

[10] M. Ansari, C. Kotselidis, K. Jarvis, M. Luján, C. Kirkham, I. Watson, Lee-tm: A non-trivial benchmark for transactional memory, in: ICA3PP '08: Proceedings of the 7th International Conference on Algorithms and Architectures for Parallel Processing, LNCS, Springer, 2008.

[11] J.-D. Choi, M. Gupta, M. Serrano, V. C. Sreedhar, S. Midkiff, Escape analysis for java, SIGPLAN Not. 34 (10) (1999) 1–19. doi:10.1145/320385.320386.
URL `http://doi.acm.org/10.1145/320385.320386`

[12] A. Herz, K. Apinis, Class-modular, class-escape and points-to analysis for object-oriented languages, in: NASA Formal Methods, Springer, 2012, pp. 106–119.

[13] B. Blanchet, Escape analysis for java: Theory and practice, ACM Trans. Program. Lang. Syst. 25 (6) (2003) 713–775. doi:10.1145/945885.945886.
URL `http://doi.acm.org/10.1145/945885.945886`

[14] B. Steensgaard, Thread-specific heaps for multi-threaded programs, in: Proceedings of the 2Nd International Symposium on Memory Management, ISMM '00, ACM, New York, NY, USA, 2000, pp. 18–24. doi:10.1145/362422.362432.
URL `http://doi.acm.org/10.1145/362422.362432`

[15] G. Kestor, O. Unsal, A. Cristal, S. Tasiran, T-Rex: A Dynamic Race Detection Tool for C/C++ Transactional Memory Applications, in: Proceedings of Eurosys 2014, ACM, 2014.

[16] M. Abadi, A. Birrell, T. Harris, J. Hsieh, M. Isard, Implementation and use of transactional memory with dynamic separation, in: O. Moor, M. Schwartzbach (Eds.), Compiler Construction, Vol. 5501 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2009, pp. 63–77.