Detection of Transactional Memory Anomalies using Static Analysis

Bruno Teixeira
CITI / Dep. de Informática
FCT / Univ. Nova de Lisboa
2829-516 Caparica Portugal
bct18897@fct.unl.pt

João Lourenço
CITI / Dep. de Informática
FCT / Univ. Nova de Lisboa
2829-516 Caparica Portugal
Joao.Lourenco@di.fct.unl.pt

Eitan Farchi
IBM HRL
Haifa, Israel
farchi@il.ibm.com

Ricardo Dias
CITI / Dep. de Informática
FCT / Univ. Nova de Lisboa
2829-516 Caparica Portugal
rjfd@di.fct.unl.pt

Diogo Sousa
CITI / Dep. de Informática
FCT / Univ. Nova de Lisboa
2829-516 Caparica Portugal
dm.sousa@fct.unl.pt

ABSTRACT

Transactional Memory allows programmers to reduce the number of synchronization errors introduced in concurrent programs, but does not ensures its complete elimination. This paper proposes a pattern matching based approach to the static detection of atomicity violation, based on a path-sensitive symbolic execution method to model four anomalies that may affect Transactional Memory programs. The proposed technique may be used to to bring to programmer's attention pairs of transactions that the programmer has mis-specified, and should have been combined into a single transaction. The algorithm first traverses the AST tree, removing all the non-transactional blocks and generating a trace tree in the path sensitive manner for each thread. The trace tree is a Trie like data structure, where each path from root to a leaf is a list of transactions. For each pair of threads, erroneous patterns involving two consecutive transactions are then checked in the trace tree. Results allow to conclude that the proposed technique, although triggering a moderate number of false positives, can be successfully applied to Java programs, correctly identifying the vast majority of the relevant erroneous patterns.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging—*Diagnostics*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Algorithms, Experimentation, Languages, Reliability, Verification

Keywords

Static Analysis, Testing, Verification, Concurrency, Software Transactional Memory

1. INTRODUCTION

Transactional Memory [9, 13] (TM) is a new approach to concurrent programming, promising both, a more efficient usage of parallelism and a more powerful semantics for constraining concurrency. Transactional Memory applies the concept of transactions, widely known from the Databases community, into the management of data in main memory. TM promises to ease the development of scalable parallel applications with performance close to finer grain threading but with the simplicity of coarse grain threading.

While TM may reduce the amount of observed concurrency errors, its usage does not by itself imply the correctness of the program. In lock-based programming, failure to use the appropriate synchronization mechanism to protect a critical region will allow an invalid access from a thread to shared data, thus triggering a synchronization error (*datarace*). In the case of TM, a critical region which is not encapsulated in a transaction may also trigger a synchronization error in TM frameworks providing a weak isolation memory model [5]. This anomaly is analogous to the classical datarace anomaly, and will be addressed in this paper as *low-level datarace*.

A program that is free of low-level dataraces is guaranteed not to have corrupted data. The values of all variables should correspond to those of a specific serial execution of all synchronized (with locks or transactions) code blocks. However, experience shows that in many programs this guarantee does not suffice to ensure a correct execution, and although not corrupted, no assumptions can be made about data consistency. The programer may intent to have two transactional blocks, and the non-transactional code between them, to run atomically, but mistakenly believe it was sufficient to ensure the atomicity of the two individual transactional blocks, leading to a *high-level datarace*.

Unlike *low-level dataraces*, *high-level dataraces* do not result from unsynchronized accesses to a variable, but rather from multiple synchronized accesses that may lead to an incorrect behavior if ran in a specific ordering. A simple example is a set of related variables that should be handled together in order to preserve some invariant, but are incorrectly handled in separate transactions, such as illustrated in Figure 1.

```
private synchronized boolean hasSpaceLeft() {
    return (this.list.size() < MAX_SIZE);
}

private synchronized void store(Object obj) {
    this.list.add(obj);
}

public void attemptToStore(Object obj) {
  if (this.hasSpaceLeft()) {
      // list may be full!
    this.store(obj);
  }
}
```

Figure 1: Example of an atomicity violation

In this Figure, a bounded data structure is implemented by wrapping a `java.lang.List` object, whose size should not go beyond `MAX_SIZE` in order to maintain a consistent state. Before the client code asks for an item to be stored in the data structure, it politely checks if there is room available in the list. All accesses to the `list` field are safely enclosed inside transactions, and therefore no low-level datarace may exist. Due to the interleaving of two threads A and B, both running the same code, when thread A is between checking the size of the list and storing the new node in the list, thread B can fully store a new node in the list, making it full, and thus making the list unable to receive the node from thread A. The problem with this code resides in the fact that both calls to `hasSpaceLeft()` and `store()` are executed as separate transactions when they should have executed as a single one. However, in the comfort of knowing that the methods from this library are atomic and safe, a programmer could easily fail to notice this issue. This is a classic example of an *atomicity violation* [15], a kind of concurrency error that is subsumed by the more broader class of *high-level anomalies* [2].

Similarly to low-level dataraces, this anomaly was also caused by incorrect assumptions of concurrency. The code in this example wrongly assumes that the state observed in a transaction is still true when the following transaction is executed.

For another example, consider the code in Figure 2, adapted from [2]. A cartesian coordinate pair is encapsulated in an object. This object allows access to each of the coordinates separately (to be used when only one of the coordinates is needed) and also to both coordinates at the same time, atomically. In this example, a thread reads each value one at a time, and tries to assert a relation among them.

We see that the reading of an object state is broken into two individually atomic operations. If other threads are allowed to change the same object meanwhile, the result is that this reading will reflect parts of different global states. Again, the previous example shows a problem resulting from wrongly assuming the preservation of a value during the full length of an operation.

Lets see a final example that shows a different situation. The example of Figure 3 shows the reverse problem, where a

```
class Coord {
  double x, y;
  public Coord(double px, double py)
        { x = px; y = py; }
  synchronized double getX()
        { return x; }
  synchronized double getY()
        { return y; }
  synchronized Coord getXY()
        { return new Coord(x, y); }
}

// ...

void examine (Coord c) {
  double x = c.getX(); // atomic
  double y = c.getY(); // atomic
  check(x == y); // values are from
                 //     different states!
}
```

Figure 2: Global reading at different stages

thread makes a global update to a shared object, but breaks this operation into individually atomic sub-operations.

Consider again the example of a coordinate pair. This time, a new operation `reset()` sets both coordinates to their initial state. However, because this operation is not atomic as a whole but rather composed by two individual transactions, other threads could observe the intermediate state. For example, if method `swap()` is executed in parallel, and its effects take place between the two transactions of `reset()`, then the resulting state could likely to be unintended. The pair would finish with just one of its coordinates reset.

```
void swap() {
  atomic {
    int oldX = coord.x;
    coord.x = coord.y; // swap X
    coord.y = oldX; // swap Y
  }
}

void reset() {
  atomic {
    coord.x = 0;
  }
  // inconsistent state (0, y)
  atomic {
    coord.y = 0;
  }
}
```

Figure 3: Example of an atomicity failure

We have seen a few scenarios where a program that is free of low-level dataraces might still present an anomalous behavior. In the following sections we will discuss a possible categorizations of these high-level anomalies, and propose a methodology to detect them in Java programs.

The rest of the paper is organized as follows. Section 2 describes some relevant work and motivates out own approach; Section 3 introduces the transactional memory anomalies we are envisaging; Section 4 describes our approach; Section 5 gives some details on the status of the current prototype;

Section 6 describes the evaluation of our approach against well known test cases; and Section 7 synthesizes the overall work and discusses some open issues and future work.

2. RELATED WORK

Two different trends addressing anomalies that are visible even if there are no low-level dataraces will now be discussed. They will be used to provide context and starting point for our own definition of high-level anomalies in TM, presented in Section 3. First, we examine atomicity violations based on the work by Wang and Stoller [15], which try to verify very strict criteria about the parallel execution of a program. Then, we analyze the notion of high-level dataraces by Artho et. al [2], which provides a less restrictive and arguably more precise approach, though it leaves out many important anomalies and reports anomalies in data accesses that are not at all problematic, exhibiting both false negatives and false positives. Finally, in Section 2.3, we make a comparison between them, see how each classifies according to each other, take conclusions and provide hints to help us create our definition in Section 3.

2.1 Atomicity Violations

Wang and Stoller [15] define an *event* as a single data access, and a *transaction* as a sequence of events performed in order by a single thread, starting and finishing at arbitrary points in the program. The analysis by Wang and Stoller intends to assess the *atomicity* of a set of transactions (*thread atomicity*). In this context, a set of transactions is atomic if all of their traces are serializable, i.e., if all possible orderings of events are equivalent to some sequential and serial execution of all transactions. If a set of transactions is not serializable, then it contains an *atomicity violation*, a concurrency defect comparable to low-level dataraces or deadlocks.

Thread atomicity may be a too restrictive requirement for concurrent programs, inhibiting some valid concurrent computations. However, it also encompasses all high-level anomalies we are detecting in TM programs. If all the executions of a concurrent program, where each thread runs until the end without interference from the others, are equivalent to one of its sequential executions, then one may consider the program as being free from concurrency anomalies. Thus, it seems safe to declare that a *thread atomic* program is free from concurrency anomalies.

On the other hand, if thread atomicity may not be inferred for the program, then there is the risk of existing atomicity violations. In [15], two algorithms are presented that dynamically detect these anomalies. Their work serves as basis for a number of other authors, from which we provide two relevant examples. First, Flanagan and Freund [8] perform a dynamic check for method atomicity. Their approach is based on the reduction-based algorithm in [15]. This algorithm makes use of left and right movers algorithm [11], which attempt to determine the equivalence of two specific orderings of execution. The second example is the one from Beckman et. al [4], who implemented a type system for atomicity. Although this is a static approach, it requires the annotation of code with type information before a program may be subject to static analysis.

2.2 High-Level Dataraces

The program correctness requirement of thread atomicity, as seen in the previous Section, can sometimes be to strict. Even by checking the more flexible *method atomicity*, as most attempts do, a lot of false positive anomalies may be triggered and reported. Many correct programs are then regarded as suffering from non-existing anomalies.

Artho et. al [2] address this issue by employing the concept of *high-level dataraces*. A high-level datarace is an anomaly that occurs when a set of shared variables is meant to be accessed atomically, but at least one thread it fails to do so.

As an example, consider again the previous example with a bounded buffer. This time we added an access counter, as illustrated in Figure 4. Each time an item is stored to (or retrieved from) the list, the counter should be incremented. A new operation `clean()` deactivates the list and frees its resources. The counter should be reset each time the list is cleaned. As can be depicted from the example, the `clean()` method first empties the list in one atomic block, and then handles the counter in another atomic operation. Other threads could observe the inconsistency of a counter indicating a positive number of accesses when the list is already deactivated.

```
// ...

public void clean() throws IOException {
  synchronized {
    this.list.clear();
    this.list = null;
  }
  this.updateCounter.reset(); // atomic op
}

public void store(Object obj)
            throws IOException {
  synchronized {
    this.list.add(obj);
    this.updateCounter.increment();
  }
}

// ...
```

Figure 4: Example of a high-level datarace

The detection of high-level dataraces is supported by the concept of *View Consistency* [2]. This property is based on chains between access sets from different threads. The *View* from a synchronized block is the set of shared variables that are accessed (read or written) in the scope of that block. The *Maximal Views* of a thread are those views which are not totally enclosed in another view from the same thread. These maximal views are an inference of what variable sets are meant to be used atomically. If a thread makes several separate accesses to variables in this set then there is a violation of the view consistency. A program that violates the view consistency is stated as containing a *High-Level Datarace*.

To provide a practical example of these concepts, consider the set of threads shown in Figure 5, taken from [2]. Thread 1 performs a safe access to both fields x and y of a shared object inside the same critical region. Thread 4 makes an access solely to x, but it does also access both fields inside another transaction. Therefore, the first synchronized block in Thread 4 is likely an access that does not need y, and this thread is regarded as safe. The views and maximal views for Thread 1 and Thread 4 are, respectively, $V_1 = M_1 = \{\{x, y\}\}$; $V_4 = \{\{x\}, \{x, y\}\}$ and $M_4 = \{\{x, y\}\}$.

Thread t1

```
synchronized(c){
  access(x);
  access(y);
}
```

Thread t2

```
synchronized(c){
  access(x);
}
```

Thread t3

```
synchronized(c){
  access(x);
}
synchronized(c){
  access(y);
}
```

Thread t4

```
synchronized(c){
  access(x);
}
synchronized(c){
  access(x);
  access(y);
}
```

Figure 5: Four threads that access shared fields. Thread 3 may cause an anomaly.

Thread 2 only accesses x, so $V_2 = M_2 = \{\{x\}\}$, which is compatible with the remaining threads. However, Thread 3 accesses both x and y in separate transactions, so $V_3 = M_3 = \{\{x\}, \{y\}\}$. One can note that $\{x, y\} \in M_1$ intersects with the elements in V_3 as $I = \{\{x\}, \{y\}\}$, and the elements in I do not form a chain, i.e. $\{x\}$ does not contain $\{y\}$ and vice-versa. Therefore, Thread 3 is reported as containing an anomaly, since it may violate the inferred assumption that those fields are related and should be handled together.

This approach may contain both false positives and false negatives. The differences between the two criteria have been well addressed in [2,15]. Authors in [2] claim that view consistency provides more precise results. In fact, many false anomalies reported with atomicity are no longer reported with view consistency. However, false negatives appear with view consistency. Praun and Gross [14] follow the work from [2] and define the concept of *method consistency*, which is much similar to that of view consistency, aiming at finding atomicity violations.

Artho et. al developed a new approach that addresses the problem of stale-value errors [3], a specific class of anomalies that are not detected with view consistency. Stale-value errors result from data privatization, which occurs when a private copy is made of a shared variable, and this copy is later used to update that same shared variable.

2.3 Comparison with Atomicity Violations and High-Level Dataraces

Informally, the notions of *atomicity* and *view consistency* are related and appear similar. However, they have different definitions and target different issues. Atomicity and view consistency offer different guarantees, and neither implies the other. We present two examples [15] that illustrate this difference. In Figure 6, two threads concurrently read shared values. Because no thread updates the shared state, the outcome is invariably the same, regardless of the execution scheduling of the operations. Hence, these threads are atomic, but view inconsistent. Figure 7 illustrates the opposite scenario. If the Thread 1 runs between the two code blocks of Thread 2, it will produce an execution which is not serializable, and therefore not atomic. However, because only one single variable is accessed in all code blocks, this set is necessarily view consistent.

Thread t1

```
synchronized {
  read(x);
  read(y);
}
```

Thread t2

```
synchronized {
  read(x);
}
synchronized {
  read(y);
}
```

Figure 6: A set of transactions that are atomic, but view inconsistent

Thread t1

```
synchronized {
  write(x);
}
```

Thread t2

```
synchronized {
  read(x);
}
synchronized {
  write(x);
}
```

Figure 7: A set of transactions that are not atomic, but view consistent

The notion of atomicity is expressed taking into consideration possible execution orderings and schedulings. View consistency, on the contrary, analyzes only data sets, independently of executions.

It is also worth noticing that the approach of view consistency works by inference. The sets of variables which are meant to be used together is inferred from the analysis. The maximal view of each thread states which variables are accessed together, not which ones *should* be related. Therefore, at least one correct usage is required in order to detect any inconsistencies at all, e.g., if x and y should always be accessed together and by mistake are always accessed separately, the anomaly will never be detected.

Atomicity checkers [4,14] tend to present a very high rate of false positives. The work on high-level dataraces aim at providing an alternative correctness criteria, with fewer false warnings. High-level datarace detection and atomicity checking address problem sets there are incomparable, even if some scenarios are correctly considered anomalies according to both criteria. High-level datarace detection creates a new class of false positives which describe scenarios that would not at all be expected to be considered anomalous computations. Furthermore, this approach based in the detection of high-level dataraces may also allows trigger false negatives.

In this paper we propose a different compromise. We depart from the notion of atomicity and attempt to detect only some specific cases of atomicity violations, which present real and serious anomalies. The cases we consider were heuristically defined by analyzing the anomalies present in a set of well known examples of buggy concurrent programs. Our approach is therefore comparable to the detection of high-level dataraces, because it attempts to detect a specific class of problems,by refining an approach that is already known as effective in the detection of concurrency anomalies.

3. HIGH-LEVEL ANOMALIES IN TRANSACTIONAL MEMORY

Thread atomicity, which ensures that the execution of a concurrent program is necessarily equivalent to some sequential execution of all its threads, is a strong guarantee that facilitates the understanding of programs by reducing its analysis to the analysis of a serial equivalent. Atomicity checking encompass all possible concurrency anomalies, but may also inhibit many concurrent computations that would be valid otherwise.

Instead of pursuing thread serializability, we intend that serializability is established only among consecutive transactions in a thread. This is justified by the intuition that most errors will come from two consecutive atomic segments in the same thread, which should be merged into a single one. This option represents a compromise between allowing some non-probable false negatives while reducing significantly the number of false positives.

More specifically, the anomalies that are to be detected happen when a thread T_1 executes transactions A and B, without running any other transactions between these two, and another thread T_2 concurrently executes another transaction C between the runs of A and B, such that the resulting scheduling of the three transactions is not serializable. This is the basis for our heuristic and all examples we have seen so far in the literature fit this scenario.

In order to facilitate reasoning about these conditions, we shall first list the possible scenarios fitting to our heuristic assumptions, in a similar way to the listing of single-variable unserializable patterns presented in [15]. We will then compare this criteria to full thread-atomicity, and then review the patterns most likely associated with anomalies.

3.1 Anomaly Patterns in Transactional Memory

Figure 8 illustrates the possible unserializable schedulings involving two threads and two transactions.

Read—Read A thread performs a transaction that writes a and a subsequent transaction that reads b, while another thread changed both of those values between the readings. An example is presented in Figure 8(a).

Read—Write A thread performs a transaction that reads a and a subsequent transaction that writes b. Between these two transactions, another thread writes a, and also reads or writes b. An example is presented in Figure 8(b).

Write—Read A thread performs a transaction that writes a and a subsequent transaction that reads b. Between these two transactions, another thread reads or writes a, and also writes b. An example is presented in Figure 8(c).

Write—Write A thread performs a transaction that writes a and a subsequent transaction that writes b. Between these two transactions, another thread reads or writes a, and also reads or writes b. An example is presented in Figure 8(d). Note that if a and b are the same, then a reading of a by another thread is both sufficient and required to trigger an anomaly. A writing of a by the second thread does not change the atomicity of the set.

3.2 Difference to Thread-Atomicity

Because we are only considering pairs of two consecutive transactions at a time, there can be the case that non-atomic schedulings involving at least three or more transactions in one of the threads are not detected as anomalies by our technique. The example in Figure 9 shows a scenario involving two threads, each executing three transactions, in

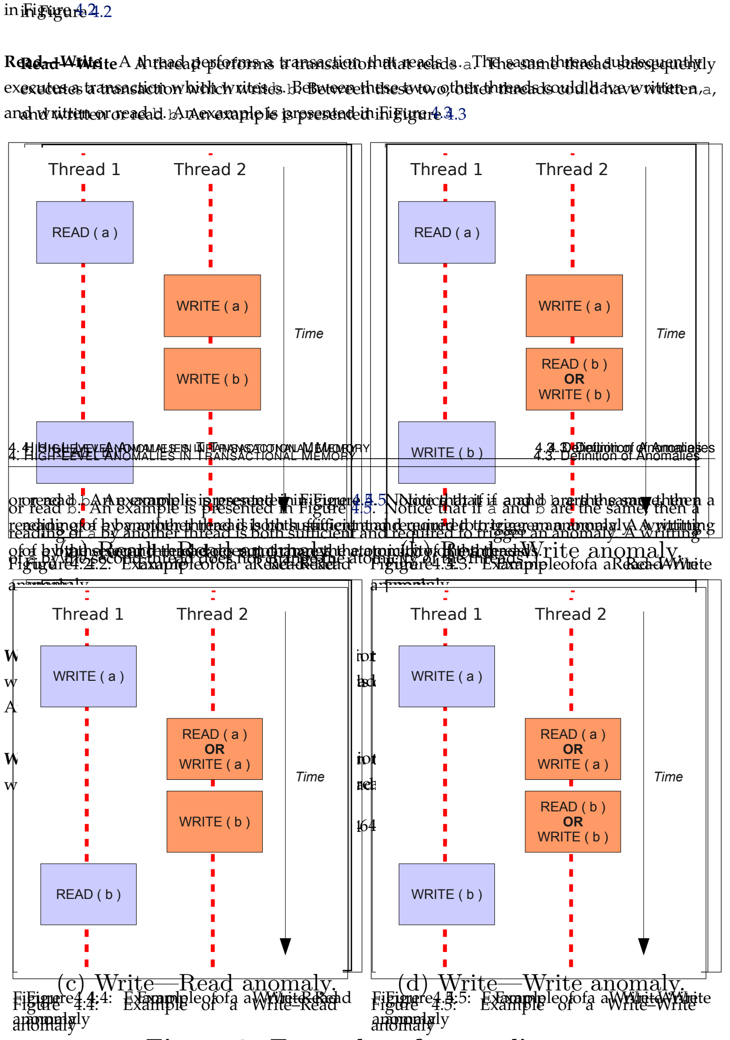

Figure 8: Examples of anomalies.

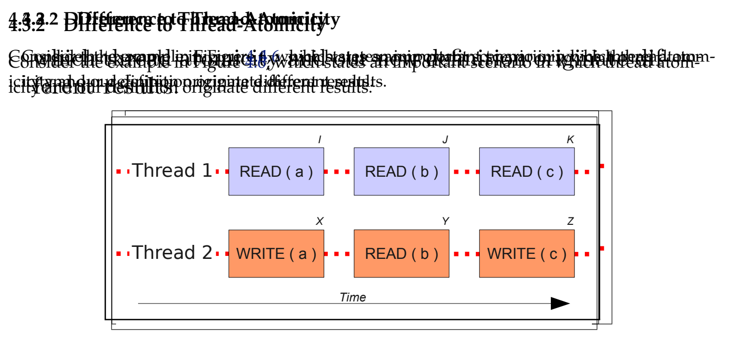

Figure 9: A set of non atomic threads, with no anomalies between each pair of consecutive transactions.

According to our systems, an anomaly would be triggered only if i) between transactions I and K of Thread 1, Thread 2 updates b (which it does not); or ii) between transactions X and Z of Thread 2, Thread 1 updates b (which also it does not). Since none of the previous conditions apply, this execution is free of anomalies according to our patterns. On the other hand, this execution is not atomic not globally serializable, because the execution of the three transactions of Thread 2 between transactions I and K could cause Thread 1 to observe two different global states.

3.3 Anomaly Patterns

We assume the run-time of the TM framework will deal with the mutual exclusion when and where necessary, freeing the analysis procedure of the burden of keeping information on which data is protected by which locks in each critical code block. However, the essence of anomaly patterns described in this Section is not limited to Transactional Memory and this approach could be adapted to the more classical locks-based model.

We can estimate that the full detection of all the patterns listed in Section 3.1 will still yield many false positives, much like thread serializability would. Furthermore, it is intuitive that many of these patterns will seldom or never be involved in an anomaly. Thus, some of these patterns will be discarded when performing an analysis, in order to produce a more manageable result. For some patterns, it may not be totally clear whether or not they should be flagged as anomalous. Therefore, we propose to address only the most intuitively anomalous patterns, and developed a validation tool that allows to interactively enable or disable the reporting of each pattern. By analyzing many examples from the literature, all the frequent high-level concurrency errors seem to fit into only three patterns, as described below.

Non-atomic Global Read (Read–write–Read/RwR) A thread reads a global state in two or more separate transactions. Therefore, it can make wrong assumptions based on a state that is composed of parts of different global states, since another thread may have modified any of the values read. The perceived global state may even be inconsistent. An example is presented in Figure 10.

```
Thread 1                   Thread 2
atomic {                   atomic {
  read (x);                  write (x);
}                            write (y);
atomic {                   }
  read (y);
}
if (x == y) {
  ...
}
```

Figure 10: Example of an RwR anomaly

Non-atomic Global Write (Write–read–Write/WrW) A thread changes the global shared state, but it breaks the update into two or more transactions. Another thread may read the global state meanwhile and observe several partial states, resulting in an inconsistency. An example is presented in Figure 11.

```
Thread 1                   Thread 2
atomic {                   atomic {
  write (x=x+1);             read (x);
}                            read (y);
atomic {                     assert (x==y);
  write (y=y+1);           }
}
```

Figure 11: Example of a WrW anomaly

Non-atomic Compare-and-Swap (Read–write–Write/RwW) A thread reads a value in a transaction and updates that same value in the following transaction. If there is a dependency between the read and stored values and meanwhile another thread also updates that same value, the initial update possibly does not make sense anymore. An example is presented in Figure 12.

```
Thread 1                   Thread 2
int a;                     atomic {
atomic {                     write (x = 42);
  a = read (x);            }
}
a = a + 1;
atomic {
  write (x = a);
}
```

Figure 12: Example of an RwW anomaly

These patterns are taken as hypothesis for our detection framework. Our real target is the occurrence of two consecutive transactions that are logically related and should be executed as a single transaction. To this end, all possible scenarios described previously are considered. Throughout Sections 4 and 5 we describe a static approach that checks for the above patterns, aiming at detecting high-level transactional memory anomalies, and a tool that implements this approach. The tool may be configured on a per-case basis, to have each pattern occurrence being considered as benign or triggering an anomaly.

4. DETECTION APPROACH

In order to evaluate the atomicity of a program, it is necessary to know the set of transactions that are executed, as well as the set of variables that are read and written by each transaction. However, it is not possible to have this information before running the program, since control flow changes as transactions are executed, also changing how many times each transaction is executed and what data is accessed inside each one.

Our approach will perform a sort of symbolic execution, starting from the bootstrap point of each thread, and assume that every transactional block is executed at its point. Similarly, we will assume that inside each transaction every read and write operation that is stated in the source code will be performed. This approach allow to obtain a set of conservative execution traces, that represents all possible executions of this thread. Each trace will be composed by a sequence of atomic blocks executed within that trace. From this trace, one can extract all pairs of consecutive transactions that may be carried out, as well as the point in code where they will be performed.

Since we are only concerned with detecting high-level concurrency anomalies in Transactional Memory, we may only trace transactional accesses to shared data. All other statements may be discarded from the trace, including statements in transactions that access only local data, as well as non-transactional accesses to shared variables.

Having set the guidelines of our method, in the remaining of this section we will discuss in further detail the relevant aspects that must be considered when defining this approach.

4.1 Transaction Nesting

In the case of atomic blocks containing sub-transactions in their code, both the top-level transaction and its sub-transactions will be handled as a single one. It should be taken into consideration that some TM systems actually provide different semantics for nested blocks. For now, however, we will take this more simplistic approach by assuming flat nesting. Therefore, the lower level atomic blocks can be discarded. More precisely, if we trace an atomic block while already in a transactional context, this block will be replaced by its sub-statements.

4.2 Method Calls

When tracing the execution method of a thread, whenever a call statement is reached, it is replaced with the statements inside the target method, in a process which will be called *inlining*. Inlining enables the viewing of all sequences of transactions performed by a thread, as though the execution was composed of one single method. If in turn this method calls other methods, then the process is repeated.

Care must be taken in order to avoid infinite inlining calls, such as in cases when two methods simultaneously call each other. In this case, the inlining should stop before the third time that a method is in the call stack. If it stops at the second time, then some execution scenarios might not be covered. Considering again the example on Figure 13, if method `B()` had not been expanded a second time, then the tracer would not foresee the possible anomaly resulting from `Transaction2()` being followed by `Transaction1()`.

```
void A() {
  Transaction1();
  B();
}

void B() {
  Transaction2();
  if(...)
    A();
}

void Main() {
  A();
}
```

```
void Main_Expanded () {
  Transaction1(); // A()
  Transaction2(); // B()
  if(...) {
    Transaction1(); // A'
    Transaction2(); // B'
    if(...) {
    // do not expand A again
    }
      // return B'
    // return A'
  }
  // return A
}
```

Figure 13: Method inlining. Care must be taken in aborting recursive inlining calls.

4.3 Alternative Execution Statements

Lets now consider *disjunction* points in a program. Disjunctions are control structures such as an *if* or *case* statement. They provide two or more alternative paths of execution, from which only one should be taken. After executing any of the branches in the disjunction, the execution path should again a single one, independently of the previous choice. This looks much like a fork and join. Disjunction nodes bring the problem of not knowing in advance which branch will be executed. However, we know that one and only one of the branches in a disjunction will be executed, and where, relatively to the previous and following points in the trace, it will take place. Therefore, we can have a special node in the trace that is not a transaction, but rather a disjunction. This will be represented by a set of sequences of transactions, from which exactly one will be taken. This allows to expose anomalies with nodes preceding and following the disjunction, while not raising anomalies between different branches.

4.4 Loop Statements

Loops are also typically available in most languages and in many variants. They always hold a code block that is executed a certain a number of times, frequently not known at compile time. We take into consideration the interactions that two consecutive executions of the loop code block may have with itself.

If we consider zero (when possible), one and two iterations of each loop, we can detect any anomaly resulting from the interaction of the code blocks preceding and succeeding the loop among themselves and with the loop code block, as well as of the the loop code block with itself.

4.5 Other Control Structures

Other control structures should be evaluated on a per-case basis, but most times they may be reduced to one of the previous cases. If we take chained if's, or else-if statements, they can be replaced with a proper disjunction node. Exception throwing and handling, try-catch, may be replaced by a sequence of disjunctions, like a code block whose execution could stop between each pair of statements. The same may go for loops that end prematurely, such as by the use of a *break* statement or similar.

4.6 Discussion of the Approach

The correctness of this analysis is subject to the correctness of the criteria determined for anomalies, i.e., such as the three common anomalies seen at the end Section 3.3. Because this approach is based in static analysis, it will not trigger false negatives with respect to those conditions. Any anomaly that is not detected is due to not matching any of the three patterns under consideration, and could be eliminated by refining those patterns or adding new ones.

In order to obtain a useful perception of the running of the program, two other analysis must be considered. These are not strictly necessary to analyze the program, but highly boost the usefulness of the process if available. Each of them may default to a conservative evaluation if not available. In both cases, their unavailability will impact the precision of the analysis, raising the chance of more false positives, although never spawning false negatives.

4.6.1 Happens-in-Parallel Analysis

Besides knowing the number and type of threads, it is important to know the time at which each of them will be performing which operation. *Start* and *Join*, as well as *Wait* and *Notify*, may alter execution flow in such a way that makes it impossible for certain interleavings to happen. The May-Happen-in-Parallel (MHP) analysis [7] can be used to achieve this purpose. To determine that an operation from a thread may interfere between two transactions from another thread, it is imperative to guarantee that this interleaving is actually possible at runtime. If a thread is guaranteed to start only after another one has finished, then there is no way there can be an anomaly between them. If this analysis is not available, then we may assume that there are no guarantees on when transactions may run, i.e., each transaction from a thread may occur between any other two consecutive transactions from another thread.

4.6.2 *Points-to Analysis*

In order to determine an anomaly, it is necessary to know that transactions are accessing the same object, and for this, the Points-to Analysis [1] can be used to identify which pointer is pointing to which variable in the program. If all transactions involved in a possible anomaly access completely disjoint sets of data, then there is no possible anomaly between them. Knowing that only fields of shared objects, and cells of shared arrays may be subject of an anomaly, we may discard accesses to local variables. Knowing whether two thread objects are the same would also help refining the static threaded control-flow and happens-in-parallel analysis. If this analysis is not available, we may assume that all accesses to a field of objects of the same class are made to the same object, i.e., that there is only one instance of each type/class, and that is is shared between all threads. Similarly, it may be assumed that all references to a thread type mean the same thread object.

5. IMPLEMENTATION

In order to implement the approach described in Section 4, we need a static analysis framework. Our choice went to the Polyglot framework [12] for extensions to the Java language. We also use an available Java language extension [6], which already recognizes atomic blocks and makes them available for further processing.

Having a program representation in the form of an AST, including nodes for atomic blocks, we extract the trace information from this. Remember that these traces are themselves trees, rather then a simple sequence of transactions, because of disjunction nodes. Because the AST produced by Polyglot is not an appropriate representation for our analysis, we designed our own tree class, with our own nodes. There are six main kinds of nodes in the new trace tree:

Access nodes represent an access to a field of a shared object. There are subtypes of nodes for read access, write access, and multiple accesses in the same statement.

Atom nodes represent an atomic block, a transaction.

Call nodes represent a call to a method, which shall be replaced by the statements of that method with the inlining process.

If nodes for binary disjunctions. A fork in the code with two possible branches of execution. The alternative branch may be empty. More disjunctive paths may be added by chaining several *If nodes*.

Loop nodes represent loops, which usually shall be replaced by three branches, one with an empty loop body, another with one iteration of the loop, and another with two iterations.

Sequence nodes represent a compound statement. Generally, the body of an if or of a method is a compound statement, unless it consists of a single statement.

A visitor pass, implemented is in accordance with the polyglot framework, transforms the original AST generated by Polyglot into our own specific format. A new set of visitors, matching our own specification of the AST, were also defined as described below.

The first pass will traverse the (modified) AST and generate an equivalent trace tree for each method. This process will also gather information on all the possibly available methods. The next pass is inlining. Each thread starting method is traversed, and every method call inside it is replaced by the tree of that method, in an iterative fashion. A sort of call stack and a counter keep track of the methods that are being expanded at a time, in order to avoid infinite inlining for recursive calls. The next iteration on the trace trees flattens a set nested atomic blocks into a single one, and generates the list of accesses of each single atomic block. This list will be used to compare data accesses. For example, many atoms contain multiple accesses to the same fields. This list will have the list of fields read, written, accessed in general, available and iterable in a number of ways. At this point, the data is gathered in a form that is suitable for analysis. An analyzer receives pairs of traces that run in parallel, and iteratively checks each pair of subsequent transactions against interference of each transaction of the second trace. If an anomalous pattern matches the analyzed transactions, then an alert is added to the report.

In the current prototype, neither static-thread, nor may-happens-in-parallel, nor points-to analysis are performed at the moment. For now, this approach conservatively assumes that there are exactly two instances of each thread type being executed in parallel, and that all transactions may be performed at an unknown time. Finally, only the object types and field names are compared in order to determine a simultaneous access; the analyzer assumes that all accesses are performed on the same object. This raises the maximum possible amount of conflicts, although we expect this number to drop significantly when those analysis steps are included in the prototype.

6. EXPERIMENTS AND RESULTS

We evaluated our static-analysis approach by using our tool in a set of examples of high-level anomalies, all but one well known from the literature. The exception was developed specifically for this thesis. None of those well known examples used Transactional Memory, but rather some lock-based concurrency control constructs, so these examples were adapted and manually rewritten to make use of TM constructs and still keep the original semantics and anomalies. Because the set of examples analyzed was quite large, we will focus in three of those examples and discuss the obtained results in detail. We will also provide some statistics on the usage of the tool on the full set of testing examples.

6.1 Testing Examples

From the set of test cases used with our application, we will now discuss in detail three of them which are representative of the set. The NASA remote agent is an instance of a critical real-world scenario in which a runtime anomaly was detected before deployment. The Coordinates test case shows an common error which is easy to make in a daily development environment. The Counter example was chosen because it contains an anomaly that some high-level anomaly checkers fail to detect, but ours does.

6.1.1 *NASA Remote Agent*

This example was adapted from [2]. The system state of a spacecraft is a table of variables (each pertaining a specific sensor), and their current values. An executing task may require that certain properties of the system state hold for the duration of the task. Once the property has been achieved, the task marks its *achieved* flag to true. A daemon thread monitors the execution, waking up at regular intervals to verify system invariants. A code snippet containing the problematic points of the program is shown in Figure 14.

```
// Task
atomic {
  table[N].value = v;
}
/* achieve property */
atomic {
  table[N].achieved = true;
}

// Daemon
while (true) {
  atomic {
    if (table[N].achieved &&
        system_state[N] != table[N].value)
      issueWarning();
  }
}
```

Figure 14: Code snippet for NASA test

The anomaly is as follows. A task has just achieved the property, and is about to set the flag *achieved*. In the mean time, the property is destroyed by some unexpected exterior event. The daemon that periodically wakes up and performs a consistency check, verifies that the property does not hold, and the achieved flag is false, adequately assuming the system is consistent at this time. If the task now resumes and sets the flag to true, a property will be marked as achieved when it is not, and the daemon has missed a possibly critical anomaly.

When analyzing this program, our tool correctly reports the anomaly. As we can see, the Task code issues two transactions, altering different fields of the same cell in the array. These modifications are actually dependent on each other. Our tool correctly detects these two consecutive transactions, as well as a possible parallel execution of the Daemon thread, which reads these two values. Therefore, a WrW anomaly is reported.

6.1.2 Coordinates

This simple test was adapted from [3]. The pair of coordinates this time has two additional operations: `swap()` to exchange the values of both fields, and `reset()` which resets both coordinates to the default value. The implementation of these two operations is shown in Figure 15.

As the code shows, the `reset()` operation is not atomic, and writes the coordinates in separate transactions. If a thread is scheduled to run `swap()` while another thread is in between both transactions of `reset()`, it would be swapping inconsistent values, therefore resulting in an inconsistent state even after `reset()` has resumed.

The anomaly has been correctly reported as a WrW. For the implementation of this test, the `main()` method spawns two threads. As each thread performs a swap operation, followed by a reset operation, additional anomalies have been reported, since the two consecutive operations access the same fields. These are false positives, and three were reported: one WrW, because `swap()` writes y, `reset()` writes x, and y could have been read in between by another instance of swap; and two RwW, because swap reads x, reset writes x, and in between another instance of swap or reset could have changed x.

```
public void swap() {
  int oldX;
  atomic {
    oldX = coord.x;
    coord.x = coord.y; // swap X
    coord.y = oldX; // swap Y
  }
}

public void reset() {
  atomic {
    coord.x = 0;
  } // inconsistent state (0, y)
  atomic {
    coord.y = 0;
  }
}
```

Figure 15: Code snippet for Coordinates test

6.1.3 Counter

Praun and Gross [14] present this program as an example of an anomaly that their approach is unable to detect. A counter is implemented as a wrapper for an integer variable, with one single operation that increments the counter with the provided argument, and returns the new value of the counter. An idiom is developed which uses this functionality to duplicate the present value. The counter is incremented by zero, returning the current unaltered value. This value is then provided as an increment argument, in order to duplicate it. However, because the value could have changed between these two operations, there is the chance that an increment will not double the value, and result in an inconsistent state. The code is presented in Figure 16.

```
public class Counter {
  int i;
  int inc(int a) {
    atomic {
      i = i + a;
      return i;
    }
  }
}

// Thread code
public void run() {
  int i = c.inc(0);
    // value could have changed
  c.inc(i);
}
```

Figure 16: Code snippet for the Counter test

For the implementation, we made a program that spawns a number of threads, with each thread running the code illustrated in the Figure. We can see that each thread issues two transactions. Our approach has correctly flagged the anomaly.

6.2 Testing Summary

Beside the three test cases already presented, we ran some additional similar tests, all adapted from examples in the literature [2, 3, 10, 14]. A total of 12 test cases were used until now to evaluate the behavior of our approach.

From a total of 11 anomalies known in these programs, 9 anomalies were correctly pointed out: 3 RwR anomalies, 3 WrW, and 3 RwW. The remaining 2 anomalies have not been detected. These failures were not due to imprecision of the anomaly patterns, but rather to data accesses which are not available. Many anomalies involve standard methods from the JRE which may possibly read or update internal data. In each of these missed anomalies, one of the conflicting operations was performed by a standard Java method. Since this code is not available, these methods are ignored, thus resulting in imprecision. As a possible way to resolve this issue, methods for which the source code is unavailable could be assumed to both read and modify the object to which the call is made.

In addition to these correct results, there were also 18 false positives (67% of total results). Out of these 18 false warnings, 5 were due to redundant reading operations. For example, in a read operation `object.field`, two readings are actually being performed, one to the object itself (`object`) and another to its `field`. Because we assume that there are always two instances of each thread running, a simple analysis may detect here a RwR anomaly. However, it makes no sense for two instances of this statement to be involved in such a conflict, so further analysis is needed to avoid these false positives. It would be possible to eliminate these false positives if one considers that both readings as a single one, thus accessing the same data.

Of the remaining false positives, 8 of them could be eliminated by refining the definition of the "common patterns", with alterations that are actually intuitive. For example, an RwR could be ignored if the second transaction would write both values involved. However, these alterations should be made with care, as they could harm the overall behavior in other tests, possibly triggering false negatives.

Finally, the remaining 5 false positives are related to interleavings which are not atomic, even though they are correct. Additional semantic information would have to be provided or inferred in order to correctly evaluate these cases. Most of these would need another level of analysis, such as a model checker, and a single one would be solved by providing *points-to* analysis.

7. CONCLUSIONS AND FUTURE WORK

In this paper we described and evaluated a new approach to detect high-level anomalies in transactional memory programs. From the obtained results we can conclude that the static analysis of programs, although limited in many aspects, is still a valid and effective approach to identify potential concurrency anomalies in programs. Because the complexity of a program grows exponentially with the number of concurrent control flows (threads), one must find a compact and efficient in-memory representation for the program, for further processing by the anomaly detection module tool. Our approach, with a conservative representation for program traces meets these requirements. An anomaly detection procedure based in evaluating pairs of transactions instead of pairs of (whole) threads has a fair precision, leading to results at least as good as the others reported in the literature.

For future work, we plan to proceed in two areas: improve the patterns and algorithms, and improve the transactional memory anomalies detection tool. In simple terms, we may say we want to keep testing and evaluating the anomaly patterns and the tool, refining both as necessary to increase anomaly detection accuracy. We plan to implement and evaluate the effect of the strategies suggested in Section 6.2 in the accuracy of the tool. We also want to test the proposed anomaly patterns in a set of larger tests, aiming at obtaining more meaningful statistical data and have some insight on the adequacy of the defined high-level transactional memory anomaly patterns. Concerning the tool itself, we strongly believe its accuracy would dramatically increase with the support for happens-in-parallel and points-to analysis, as referred in Sections 4.6.1 and 4.6.2. Currently, methods with no source-code available are assumed to neither write or read data, which is not true in most cases. For example, the methods for sending data in `java.lang.Socket`, such as the `socket.write(bytes)` method should be assumed to change the `Socket` object, and this scenario is currently not being considered. Likewise, references passed as arguments to methods with no source-code available may also change the referenced object. In both cases, assuming that the objects are modified by such methods will solve this problem, although it may lead to a considerable increase of false positives. The tools could also be improved with support for user-defined patterns and the option to enable and disable them as required.

Acknowledgments

This work was partially supported by Sun Microsystems and Sun Microsystems Portugal under the "Sun Worldwide Marketing Loaner Agreement #11497", by the Centro de Informática e Tecnologias da Informação (CITI) and by the Fundação para a Ciência e Tecnologia (FCT/MCTES) in the Byzantium research project PTDC/EIA/74325/2006 and research grant SFRH/BD/41765/2007.

8. REFERENCES

[1] Lars Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[2] C. Artho, K. Havelund, and A. Biere. High-level data races, 2003.

[3] Cyrille Artho, Klaus Havelund, and Armin Biere. Using block-local atomicity to detect stale-value concurrency errors. In Farn Wang, editor, *ATVA*, volume 3299 of *Lecture Notes in Computer Science*, pages 150–164. Springer, 2004.

[4] Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, 2008.

[5] Colin Blundell, E. Christopher Lewis, and Milo M. K. Martin. Deconstructing transactional semantics: The subtleties of atomicity. In *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*, June 2005.

[6] Ricardo Dias. Source-to-source java stm framework compiler. Technical report, Departamento de Informática FCT/UNL, April 2009.

[7] Evelyn Duesterwald and Mary Lou Soffa. Concurrency analysis in the presence of procedures using a data-flow framework. In *TAV4: Proceedings of the symposium on Testing, analysis, and verification*, pages 36–48, New York, NY, USA, 1991. ACM.

[8] Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 256–267, New York, NY, USA, 2004. ACM.

[9] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[10] IBM's Concurrency Testing Repository.

[11] Richard J. Lipton. Reduction: a method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.

[12] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, pages 138–152, 2003.

[13] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 204–213, New York, NY, USA, 1995. ACM.

[14] Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology*, page 2004, 2003.

[15] Liqiang Wang and Scott D. Stoller. Run-time analysis for atomicity. *Electronic Notes in Theoretical Computer Science*, 89(2):191–209, 2003. RV '2003, Run-time Verification (Satellite Workshop of CAV '03).