UDC, DOI: N/A

Detecting Concurrency Anomalies in Transactional Memory Programs

João Lourenço, Diogo Sousa, Bruno Teixeira and Ricardo Dias*

CITI / Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
2829-516 Caparica, Portugal

{joao.lourenco, rjfd}@di.fct.unl.pt
{dm.sousa, bct18897}@fct.unl.pt

Abstract. Concurrent programs may suffer from concurrency anomalies that may lead to erroneous and unpredictable program behaviors. To ensure program correctness, these anomalies must be diagnosed and corrected. This paper addresses the detection of both low- and high-level anomalies in the Transactional Memory setting. We propose a static analysis procedure and a framework to address Transactional Memory anomalies. We start by dealing with the classic case of *low-level dataraces*, identifying concurrent accesses to shared memory cells that are not protected within the scope of a memory transaction. Then, we address the case of *high-level dataraces*, bringing the programmer's attention to pairs of memory transactions that were misspecified and should have been combined into a single transaction. Our framework was applied to a set of programs, collected form different sources, containing well known low- and high-level anomalies. The framework demonstrated to be accurate, confirming the effectiveness of using static analysis techniques to precisely identify concurrency anomalies in Transactional Memory programs.

Keywords: Testing, Verification, Concurrency, Software Transactional Memory, Static Analysis.

1. Introduction

Concurrent programming is inherently hard. The fact that more than one ordering of events may take place at runtime leads to an exponential growth in the number of both valid and invalid program states. Programs with concurrency errors may reach invalid states and expose unpredicted and anomalous behaviors. Thus, more than just convenient, tool based approaches tackling the automatic verification and validation of programs are essential building-blocks in the

* This work was partially supported by Sun Microsystems 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 research projects PTDC/EIA/74325/2006, PTDC/EIA-EIA/108963/2008, and PTDC/EIA-EIA/113613/2009, and research grant SFRH/BD/41765/2007.

process of developing correct concurrent programs. This paper addresses the identification of both low- and high-level dataraces in the Transactional Memory [12, 18, 20, 21] setting.

Data races are among the most notorious concurrency errors. A program suffers from a *low-level datarace*, or simply *datarace*, when two threads concurrently access a shared variable with no concurrency control enforced, and at least one of those accesses is an update. Low-level dataraces may be avoided by synchronizing the conflicting threads, e.g., using locks, thus enforcing that critical sections, program code blocks that are mutually exclusive, will not be executed concurrently.

A program free from low-level dataraces may still exhibit concurrency anomalies resulting from the a scope misspecification, where two or more correctly synchronized critical sections should be merged into a single one to ensure the program's correctness. We shall call these errors *high-level dataraces* or *high-level anomalies*. Likewise, low-level dataraces are also referred in this paper as *low-level anomalies*.

Transactional Memory (TM) [11,19] is a promising approach that offers multiple advantages for concurrent programming. In contrast to locks, which enforces mutual exclusion, TM is neutral concerning the execution model, resorting in a transactional monitor to establish the transactional properties at run-time. The transactional monitor may opt to enforce mutual exclusion, as with locks, or to allow transactions to execute concurrently, optimistically assuming they will not conflict, and later aborting and restarting those that do conflict.

TM is inherently immune to some of the concurrency anomalies that are common in lock-based programs, such as deadlocks. Data races are among the anomalies that can still be observed. A transaction is only shielded against another transaction, in the same way that a lock-protected critical section is only protected from another critical section which holds a common lock. Therefore, in the TM setting, non-transactional and transactional code may also compete when accessing shared variables, leading to low-level dataraces. Likewise high-level dataraces in lock-based programs, the misspecification of the scope of two or more memory transactions may lead to high-level dataraces in the TM setting.

There are several approaches for detecting low-level dataraces in lock-based programs, both static [6, 9, 14], dynamic [8, 13, 16], and hybrid [17]. Likewise, there are also some approaches for detecting high-level dataraces in lock-based programs [3, 5, 10, 22, 24]. As locks relate to transactions, these works on low- and high-level dataraces also relate to TM, but none of them targets specifically this setting, which is in the core of our approach.

In Section 2, we discuss a process that enables the usage of a low-level datarace detector meant for locks in a TM-based program. In Section 3 we propose a definition for high-level dataraces in the TM setting and address their detection using static analysis, by conservatively extracting all possible concurrent execution traces of a program and searching for anomalies using a pattern-

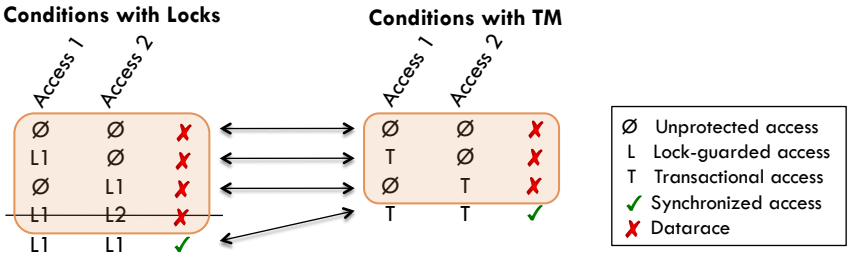

Fig. 1. Conditions for a datarace in Transactional and Locks.

based heuristic approach. We then discuss the related work in Section 4, followed by the conclusions and future work in the last section.

2. Low-Level Dataraces in Transactional Memory

Locks enforce mutual exclusion between critical sections. If two critical sections are protected by at least one common lock, then no two threads may execute them at the same time. Transactional memory, on the other hand, does not enforce mutual exclusion. Instead, two transactional code blocks may execute concurrently, provided by the TM run-time with the guarantees of *Isolation* and *Atomicity*. TM usually provides the *serializability* of transactions, ensuring that if two memory transactions take place concurrently and both succeed, then its final outcome is the same as if those two transactions were executed one after the other. Violations of serializability usually lead to dataraces.

Consider the distinct situations that may lead to a low-level datarace between two lock-based synchronized threads:

1. None of the accesses are performed while holding a lock;
2. One of the accesses is performed holding no locks; or
3. Both accesses are performed while holding disjoint sets of locks.

When using locks, the user chooses which critical sections shall be mutually exclusive by acquiring and holding the appropriate lock. In the TM setting, all transactions are guaranteed to be atomic and isolated from all other concurrent transactions, which excludes the third case above. Hence, as illustrated in Figure 1, the only two situations that may lead to low-level dataraces in TM are:

1. None of the accesses is performed in the scope of a transaction; or
2. Only one of the accesses is performed in the scope of a transaction.

Listing 1 illustrates cases of dataraces in both lock-based and transactional memory programs (left and right columns respectively). In the lock-based case, blocks A, B, and C, are assumed to execute concurrently. Likewise for the TM case, where blocks W, X, and Y, are also assumed to execute concurrently.

```
// A                              // W
synchronized (a) {                atomic {
    a.x = 0;                          a.x = 0;
}                                 }
...                               ...
// B                              // X
print(a.x);                       print(a.x);
...                               ...
// C                              // Y
synchronized (this) {             atomic {
    a.x++;                            a.x++;
}                                 }
```

Listing 1. Example of a low-level dataraces with locks (left) and with transactional memory (rigth)

The lock-based version has two different kinds of dataraces. Blocks A and B have a no-lock conflict, as block B is accessing $a.x$ without holding a lock. The same applies to blocks C and B. Blocks A and C have a wrong-lock conflict, as both are holding different locks and thus their concurrent accesses to $a.x$ are not protected. All the dataraces in the TM-based version are of a single type. Blocks W and X have a data race resulting from the execution of block X outside the scope of a transaction, and the same applies to blocks Y and X.

Our approach to identify low-level dataraces in TM programs resorts to their similarities and relations to the low-level dataraces in lock-based programs, interpreting the TM `atomic` blocks as if they were synchronized on a single global lock and then apply techniques and tools used in the detection of dataraces in lock-based programs. Hence, each of the scenarios of low-level datarace in TM maps into an analogous scenarios in a single lock setting, as denoted by the arrows in Figure 1, making this approach both sound and complete [21].

2.1. Detection Approach

Our approach to identify low-level dataraces in transactional memory programs is depicted in Figure 2.

The TM Java program is processed with AJEX [7], an extension to Polyglot [15], that recognizes the keyword `atomic` as a new Java construct to denote a transactional memory code block and generates the corresponding Abstract Syntax Tree (AST). The AST generates by AJEX is then traversed using the Polyglot framework and the transactional blocks are replaced with blocks synchronized on a single unused global lock. The definition of the new global lock is added to the main class, if one exists, otherwise to another arbitrary class. The identifier of the global lock has a fresh name inside the possessing class and is a public static object. Figure 3 illustrates this transformation process.

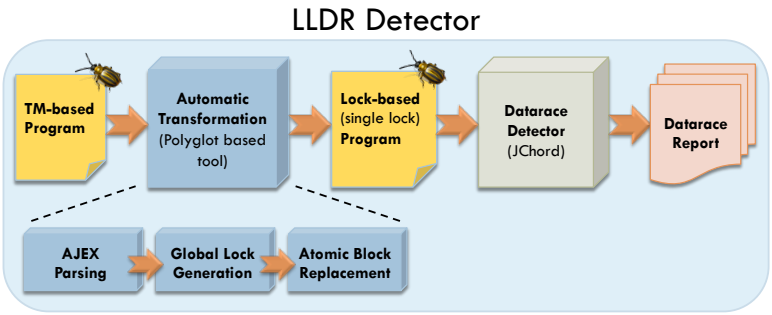

Fig. 2. The low-level datarace detection procedure.

This automatic transformation process generates a Java compliant version of the original TM-based program. This program is then fed to a lock-based datarace detector. We used JChord [14] as a datarace detector, but other similar tools could be used instead.

2.2. Experiments

In order to validate our approach for transforming TM programs to synchronized single-lock programs, a set of validation tests have been carried out [21]. Some of these tests are well-known erroneous programs intended to benchmark validation tools like our own. Others were developed specifically to test our tool, containing simple stub programs with dataraces. We also tested Lee-TM [2], a renowned transactional memory benchmark. It was necessary to have two versions of each test, one using locks and another using TM. This implies that the existing tests meant for locks had to be manually rewritten using TM. We succeeded in keeping the original semantics (and errors) in the TM versions of the test programs, except for a small number of well identified cases.

Tests were carried out by initially running JChord on the lock-based version of each test. The results were registered for future reference. Then, we applied our approach to the TM versions of those tests, by transforming them into single global-lock programs and feeding them to JChord. The results were again registered. For each test, the results for both executions of JChord — in the original and transformed TM versions — were then compared. All the results obtained fit into one of the following scenarios: for tests where the lock-based and TM-based versions were strictly equivalent, the analysis results were equivalent as well; when the TM and lock-based versions of a test would have slightly different semantics, since some lock-based bugs could not be replicated using the TM model, results were slightly different, but all those differences could be clearly mapped to the semantic variations between the two versions.

As an example, consider the Lee-TM benchmark. By running JChord in the original lock-based version, we identified 52 dataraces. A careful analysis of

```
private boolean hasSpaceLeft() {
    atomic { return (list.size() < MAX_SIZE); }
}
private void store(Object obj) {
    atomic { list.add(obj); }
}
```

```
public Object GLOCK;

private boolean hasSpaceLeft() {
    synchronized (GLOCK) { return (list.size() < MAX_SIZE); }
}
private void store(Object obj) {
    synchronized (GLOCK) { list.add(obj); }
}
```

Fig. 3. Transformation of transactions into code blocks synchronized in a global lock.

these dataraces proved them all to be either false positives or, although confirmed, harmless. Running JChord in the single-lock version resulting from our automatic transformation process, we identified 48 dataraces, all with a direct correspondence to the dataraces observed in the original lock-based version. The remaining 4 dataraces correspond to *wrong-lock* situations, which do not exist in TM.

3. High-level Dataraces

A program that is free from low-level dataraces may still suffer from concurrency errors. Unlike low-level dataraces, high-level dataraces do not result from unsynchronized accesses to shared variables, but rather from a combination of multiple synchronized accesses, which may lead to incorrect behaviors if executed in a specific order.

As an example, consider the program in Listing 2, showing a bounded data structure whose size cannot go beyond `MAX_SIZE`. All accesses to the `list` fields are safely enclosed inside transactions, therefore no low-level datarace exists. But there is nonetheless a high-level datarace.

In the function `attemptToStore()`, a thread always checks for available room before storing an item in the queue. However, between the executions of `hasSpaceLeft()` and `store()`, the list may be filled by another concurrent thread executing the same code, leaving no space left; thus, the first thread would now be adding an element to an already full list. Both method calls to `hasSpaceLeft()` and `store()` should have been executed inside the same transaction. This was not the case, thus leading to a high-level datarace.

In the following sections we will discuss the conditions that may trigger high-level anomalies, propose a possible categorization of those anomalies, and present our approach for their identification.

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

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

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

Listing 2. Example of a High-level Anomaly

3.1. Thread Atomicity

High-level anomalies are related to sets of transactions involving different threads, which leave the program in an inconsistent state when executed in a specific order. This happens because two or more of the transactions executed by one thread are somehow related, making assumptions about each other (e.g., assuming success), but there is a scheduling in which another thread issues a concurrent transaction which breaks that assumption. The simplest way to solve this problem is to merge those related transactions into a single one. Furthermore, through empirical observation, it seems that many of such anomalies involve only three transactions. Two consecutive transactions from one thread and a third transaction from another thread, that when scheduled to run between the other two, causes an anomaly.

Without further information from the developer intention on the program semantics, at compile time it is not possible to infer all the relations among transactions. It is possible, however, to identify transactions that may or will affect other transactions, and use this information to identify potential high-level anomalies.

Consider a coordinate pair object shared between multiple threads. Assume that a thread T_1 issues a transaction $t_{1.1}$ to read value x, and then issues transaction $t_{1.2}$ to read y. In between them, thread T_2 could issue transaction $t_{2.1}$ which sets both values to 0, and so thread T_1 would have read values corresponding to the old x and new y (zero), when it is likely that both read operations were meant to read one single instant, i.e., either both before or after $t_{2.1}$. In this scenario, the final outcome is not equivalent to a situation in which both read operations were ran without interleaving. The property of a set of threads whose interleavings are guaranteed to be equivalent to their sequential execution is called *thread atomicity* [24], and will be further discussed in Section 4. It is common to pursue thread atomicity as being a correctness criterion.

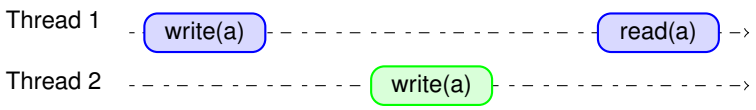

Fig. 4. An unserializable pattern which does not appear to be anomalous.

3.2. Anomaly Patterns in Transactional Memory Programs

Since full thread atomicity may be too restrictive, thus triggering too many false positive scenarios, we opted for a more relaxed semantic that allows a restricted number of atomicity violations. As an example of an atomicity violation which in principle is not an error, consider the example in Figure 4, where each rectangle corresponds to a transactional code block. The second operation in T_1 will be retrieving the results written by T_2. In order for this set of threads to be serializable, and thus thread atomic, all possible interleavings would have to be equivalent to the scenario in which the read immediately follows the write of the same thread.

However, given the specific context of TM and the set of operations presented in Figure 4, it seems unintuitive that this particular set would contain an error. The read operation is retrieving a, and it seems unlikely that an operation will be performed based on the value written before by the same thread, as it would possibly be already outdated. The only error scenario involving this particular setup would be the case in which after the read, the first thread would do a set of operations that depend on both, the value just read and the value previously written and assuming them to be equal.

We propose a framework for detecting a configurable set of patterns, and we opted to include only those most likely will result in concurrency anomalies. Out of all the patterns that incur in atomicity violations, we have isolated three highly suspicious patterns which describe possible high-level anomalies. These patterns are summarized in Figure 5 (with $x \neq y$).

Read–write–Read or ***RwR* —** Non-atomic global read. A thread reads a global state in two or more separate transactions, and the global state was changed by another thread meanwhile. If the first thread makes assumptions based on that state, it will most probably be a high-level anomaly.

Write–read–Write or ***WrW* —** Non-atomic global write. This is the opposite scenario from above. A thread is changing the global shared state in multiple separate transactions. Other thread reading the global state will observe this state as inconsistent.

Read–write–Write or ***RwW* —** Non-atomic *compare-and-swap*. In this pattern a thread checks a variable value, and based on that value it changes the state of the variable. If the variable was changed meanwhile, the update will probably may not make sense anymore.

Anomalies between two consecutive transactions can be defined a triple of transactions (T_1, T_2, T_3), such that the execution of T_2 by one thread interferes

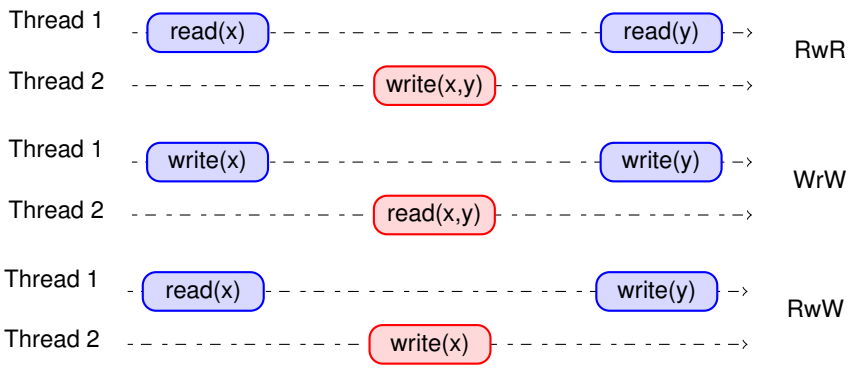

Fig. 5. Common anomalous access patterns.

with the normal execution of T_1—T_3 by another thread. It is common for a program to have multiple anomalies $A = (T_1, T_2, T_3)$ and $A' = (T_1, T_2', T_3)$. Even though these anomalies are of different nature, they reflect the lack of atomicity between the same pair of transactions. We define a *main anomaly* as a triple (T_1, S, T_2) were S is the set of transactions that interfere with the expected execution of T_1—T_3.

In the following, we will present our approach for statically matching suspicious patterns against the program source code, and will report on the experiments that assess the applicability and effectiveness of these patterns.

3.3. Symbolic Execution of Transactional Memory Programs

To detect high-level anomalies in TM programs, we perform a symbolic execution of the program and generate a set of possible execution traces of the transactional code. From these traces, we generate the set of possible interleavings of transactional code blocks and check if there are matches with any of the patterns identified in Section 3.2. Our approach for the detection of high-level anomalies in TM programs was also implemented resorting to the Polyglot [15] framework and AJEX [7].

The thread traces are obtained by performing a symbolic execution of the given program. When the program to be analyzed is loaded, all class declarations that contain main thread methods are retrieved. This includes classes that have a execution entry point such as a `main()` method, and classes that inherit from `java.lang.Thread` or `java.lang.Runnable` containing a `run()` method declaration. Hence, we obtain a list of all thread bootstrap methods. Statements in these thread methods are then analyzed. Whenever a transactional code block is found, it is added to the current trace, together with the full list of read and write operations of that transaction.

```
<?xml version="1.0" encoding="UTF-8"?>
<classes>
  <class id="a.package.AClass">
    <method id="doThings(Object,␣Object)">
      <changes>this</changes> <!-- Target object may change -->
      <changes>1</changes> <!-- First argument may change -->
    </method>
  </class>
</classes>
```

Listing 3. Example of the XML file that specified access to unavailable methods.

To find out the read and write operations of a method call, we in-line the called code, i.e., we replace the method call with the body of the target method, so that the transactions performed by that method are still seen as being performed by the current thread. Care must be taken not to perform infinite inlinings when in the presence of recursive methods.

In real world programs much code is already compiled, and the original source code is unavailable (this includes Java standard libraries). It may also happen that a program calls native methods, that may not be analyzed by our tool. When these cases arises our tools issues a warning, with the full qualified signature of the method it cannot analyze. With this information the user can build a database of the accesses performed by methods whose source code is not available. This is performed with an XML file as illustrated in Listing 3.

Additional challenges derive from disjunctions in the program control flow. When there are multiple possible execution flows, such as with *if-else* or *switch* statements, the current trace must still represent all possible executions. Instead of having numerous alternative traces for the same thread, a special disjunction node is added to the trace, symbolizing a disjunction point, where the execution can follow one of the multiple alternative paths. Thus, the trace actually takes the form of a tree representing all the transactional blocks in all the possible execution paths for a thread.

Finally, we also have to deal with loop structures in the input program. This is solved by considering the representative scenarios of the execution of loops. The trace tree only considers the cases in which the loop is not executed or is executed twice. We need to consider zero executions of the loop body, for the case in which the transaction that precedes and the one that follows the loop are both involved in an anomaly. When considering two executions of the loop we cover three different cases: when a transaction that precedes the loop is involved in an anomaly with a transaction in the loop; when a transaction in the loop is involved in an anomaly with the transaction that follows the loop; and when a transaction in the loop is involved in an anomaly with itself in the next iteration of the loop. It is not necessary to check for a single execution of the loop as two loop unrolls generate a super-set of the cases generated by a single loop unroll. It is also not necessary to consider more than two consecutive

Table 1. Experimental results summary.

Test Name	Total Anomalies	Total Warnings	Correct Warnings	False Warnings	Missed Anomalies
Connection [5]	2	3	2	1	0
Coordinates'03 [3]	2	7	2	5	0
Local Variable [3]	1	1	1	0	0
NASA [3]	1	1	1	0	0
Coordinates'04 [4]	1	2	1	1	0
Buffer [4]	0	1	0	1	0
Double-Check [4]	0	1	0	1	0
StringBuffer [10]	1	1	1	0	0
Account [23]	1	1	1	0	0
Jigsaw [23]	1	2	1	1	0
Over-reporting [23]	0	1	0	1	0
Under-reporting [23]	1	1	1	0	0
Allocate Vector [1]	1	2	1	1	0
Knight Moves [21]	1	1	1	0	0
Arithmetic Database [21]	1	2	2	0	1*
Total	**14**	**27**	**15**	**12**	**1**

* This anomaly was partially detected.

executions, since all the anomalies detected with three or more expansions of the loop body are duplicates of those detected with just two expansions.

3.4. Validation of the Approach

We ran a total of 15 tests for detecting high-level anomalies in TM programs. Many of those tests consist of small programs taken from the literature [3, 4, 5, 10, 23] with well studied high-level anomalies. The *Allocate Vector* test was taken from the IBM concurrency benchmark repository [1]. We also developed two of the tests [21]. All the 15 test programs were analyzed with success by our tool.

We measure the effectiveness of our high-level datarace detector by the number of *main anomalies* reported, as defined in Section 3.2. This metric is better than the regular, pattern specific, anomaly count, since it reflects the number of spots that lack inter-transaction atomicity. It is also more meaningful for the user since it point our the transactions that should be merged.

The results are summarized in Table 1. In a total of 14 anomalies present in these programs, 13 were correctly pointed out.

In the *Arithmetic Database* test our tool indicates two anomalies; these anomalies are part of a larger anomaly which was not detected as a whole. This program performs four transactions that should be merged in a single one. Since our detection approach is based on the most common case of anomalies between a pair of adjacent transactions, we fail to see the lack of atomicity of these four transactions. It is also worth noticing that two specific cases of the anomaly were reported.

In addition to the correctly detected anomalies, there were also 12 false positives (45% of total warnings). We group the causes for these imprecisions in 4 different categories.

Out of these 12 false warnings, 2 were due to redundant read operations: when reading `object.field`, we consider that two readings are actually being performed, one to `object` and another to `field`. It makes no sense for two instances of this statement to be involved in an anomaly. It is possible to eliminate these false positives by tailoring the analysis and consider only one read operation in the access to `field`.

Another 4 false positives are related to cases for which additional semantic information would have to be provided by the software developer or somehow inferred. These false warnings could be eliminated with the aid of other available techniques, such as *points-to* and *may-happen-in-parallel* analyses.

Two other false positives could be eliminated by refining the definition of the *anomaly patterns* described in Section 3.2. For example, an *RwR* anomaly could be ignored if the last transaction reads both values involved.

Finally, 4 false warnings which are matched by our anomaly patterns are definite false positives. Further study would be necessary to adapt the anomaly patterns in order to leave out these correct accesses, without compromising the precision of the detector. If we fix all the previous false positives but these last four, we will be able to reduce the percentage of false positives from 45% to 15%, which is much better than what can be observed in related works.

4. Related Work

Low-level datarace detection, either by observing a program's execution (dynamic approach) or its specification (static approach) has been an area of intense research [6, 8, 9, 13, 14, 16, 17]. We are unaware of any work that specifically targets the detection of low-level dataraces in the TM setting. However, we have shown that current algorithms and tools, which are intended for use with lock-based mechanisms, may as well be applied to transformed TM programs.

There are some relevant works on high-level anomaly detection that, although not targeting the TM setting, share some principles and features with our own work. One of the earliest works on the subject is the one by Wang and Stoller [24]. They introduce the concept of thread atomicity, with *atomicity* having a different meaning than the one stated in the ACID properties provided by TM systems. In this case, thread atomicity is more related to *serializability*, and it means that any concurrent execution of a set of threads must be equivalent

to some sequential execution of the same set of threads. Wang and Stoller provide two algorithms for dynamically (i.e., at runtime) finding atomicity violations. Other authors have based on this work to develop other approaches [5,10]. Our approach, being less strict than the one from Wang and Stoller [24], tends to be more precise and generates much less false positives.

An attempt to provide a more accurate definition of anomalies is the work on *High-Level Dataraces* (HLDRs) by Artho et. al [3]. Informally, an HLDR in this context refers to variables that are related and should be accessed together, but there is some thread that does not access that variable set atomically. This is different from thread atomicity, which considers the interaction between transactions, without regard for relations between variables.

Because HLDR is concerned with sets of related variables, some atomicity violations are not regarded as anomalies, such as those concerning only to one variable. On the other hand, it is possible that an HLDR does not incur in an atomicity violation. This work is in some way related to ours, in that it attempts to increase the precision of thread atomicity by reducing the false positives cases. However, while our approach is to simply disregard some atomicity violations as safe, the work by Artho founds a new definition, which still exhibits some false positives, and also introduces some false negatives. This work is also related to our in that they both automatically infer data relationships and do not require processing user annotations which state those relationships.

A different approach has been taken by Vaziri et. al [22]. Their work focuses on a static pattern matching approach. The patterns reflect each of all the possible situations that may lead to an atomicity violation. The anomalies are detected based on sets of variables that should be handled as a whole. To this end, the user must explicitly declare the sets of values that are related. This work is similar to ours in that both approaches are static, and both follow a pattern-matching scheme. However, our approach is intended to be applied to existing programs, and so it assumes that any set of variables may be related. Contrarily, the work by Vaziri demands that the user explicitly declares which sets of variables are meant to be treated atomically, and so it can trigger anomalies on all atomicity violations, without too many false positives.

5. Concluding Remarks

In this paper we proposed to detect low-level dataraces in transactional memory programs by explore the correspondence between low-level dataraces in these programs with equivalent low-level dataraces in lock-based programs. We proposed to convert all memory transactions in a program into synchronized blocks, all synchronizing in a single global lock. This was achieved with static analysis of the source code and a source-to-source transformation.

Application of this technique to well known test programs proved to be effective in the detection of low-level anomalies in transactional memory programs.

We have analyzed common criteria for reporting high-level anomalies, and attempted to provide a more useful criteria by defining three anomaly pat-

terns. A new approach to static detection of high-level concurrency anomalies in Transactional Memory programs was defined and implemented. This new approach works by conservatively tracing transactions and matching the interference between each consecutive pair of transactions against a set of well defined anomaly patterns. Our approach raises false positives, although at an acceptable level; and well known techniques can be applied to prune the false warnings to and even lower level. When compared with the existing reports from literature, these results are, in general, considerably better. We may therefore conclude that our conservative tracing of transactions is a reasonable indicator of the behavior of a program, since our results rival with those of dynamic approaches.

The developed framework can be improved by further refining the error patterns. The addition of *points-to* and *may-happen-in-parallel* analyses would help to improve the tool by reducing the number of states to be analyzed. Other improvements could be achieved by enabling the analysis of standard or unavailable methods, and by solving the issue of redundant read accesses.

Our approach is novel because it is based in static analysis; it extracts conservative trace trees aiming at reducing the number of states to be analyzed; and it detects anomalies using a heuristic based in a set of suspicious patterns believed to be anomalous.

References

1. IBM's Concurrency Testing Repository, `https://qp.research.ibm.com/concurrency_testing`
2. Ansari, M., et al.: Lee-TM: A non-trivial benchmark suite for transactional memory. In: Proceedings of ICA3PP '08. pp. 196–207. Springer-Verlag, Berlin (2008)
3. Artho, C., Havelund, K., Biere, A.: High-level data races. Softw. Test., Verif. Reliab. 13(4), 207–227 (2003)
4. Artho, C., Havelund, K., Biere, A.: Using block-local atomicity to detect stale-value concurrency errors. In: Wang, F. (ed.) ATVA. Lecture Notes in Computer Science, vol. 3299, pp. 150–164. Springer (2004)
5. Beckman, N.E., Bierhoff, K., Aldrich, J.: Verifying correct usage of atomic blocks and typestate. SIGPLAN Not. 43(10), 227–244 (2008)
6. deok Choi, J., Loginov, A., Sarkar, V., Logthor, A.: Static datarace analysis for multithreaded object-oriented programs. Tech. rep., IBM Research Division, Thomas J. Watson Research Centre (2001)
7. Dias, R., Teixeira, B.: Ajex: A source-to-source java stm framework compiler. Tech. rep., DI-FCT/UNL (Apr 2009)
8. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. SIGPLAN Not. 26(12), 85–96 (1991)
9. Flanagan, C., Freund, S.N.: Type-based race detection for Java. SIGPLAN Not. 35(5), 219–232 (2000)
10. Flanagan, C., Freund, S.N.: Atomizer: a dynamic atomicity checker for multithreaded programs. In: Proceedings of POPL'04. pp. 256–267. ACM, New York, NY, USA (2004)

11. Herlihy, M., Luchangco, V., Moir, M., Scherer, I.W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of PODC'03. pp. 92–101. ACM, New York, NY, USA (2003)
12. Herlihy, M., Luchangco, V., Moir, M., William N. Scherer, I.: Software transactional memory for dynamic-sized data structures. In: PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing. pp. 92–101. ACM, New York, NY, USA (2003)
13. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: Proceedings of PLDI'09. pp. 134–143. ACM, New York, NY, USA (2009)
14. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: PLDI. pp. 308–319. ACM Press (2006)
15. Nystrom, N., Clarkson, M.R., Myers, A.C.: Polyglot: An extensible compiler framework for java. In: CC. pp. 138–152 (2003)
16. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. SIGPLAN Not. 38(10), 167–178 (2003)
17. Qi, Y., Das, R., Luo, Z.D., Trotter, M.: Multicoresdk: a practical and efficient data race detector for real-world applications. In: Proceedings of the 7th Workshop on Parallel and Distributed Systems. pp. 1–11. ACM, New York, NY, USA (2009)
18. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of PODC'95. pp. 204–213. ACM, New York, NY, USA (1995)
19. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of PODC'95. pp. 204–213. ACM, New York, NY, USA (1995)
20. Teixeira, B., Sousa, D., Lourenço, J., Dias, R., Farchi, E.: Detection of transactional memory anomalies using static analysis. In: Proceedings of PADTAD'10. pp. 26–36. ACM, New York, NY, USA (2010)
21. Teixeira, B.: Static Detection of Anomalies in Transactional Memory Programs. Master's thesis, Universidade Nova de Lisboa (Apr 2010)
22. Vaziri, M., Tip, F., Dolby, J.: Associating synchronization constraints with data in an object-oriented language. In: Proceedings of POPL'06. pp. 334–345. ACM, New York, NY, USA (2006)
23. von Praun, C., Gross, T.R.: Static detection of atomicity violations in object-oriented programs. In: Journal of Object Technology. p. 2004 (2003)
24. Wang, L., Stoller, S.D.: 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)

João Lourenço is an Assistant Professor at the Computer Science Department of New University of Lisbon, Portugal. He received his MSc and PhD in 1995 and 2004 respectively, both from the New University of Lisbon. His current research interests focus on Software Transactional Memory, more specifically in Transactional Memory run-time and programming language support, and also software development environments and tools for TM, including testing and debugging tools. He is currently a member of the Transactional Systems Research Team (TrxSys) at the Center for Informatics and Information Technologies (CITI).

Diogo Sousa is a BSc student at the Computer Science Department of New University of Lisbon, Portugal. He will conclude his BSc in July 2011 and plans to enroll in the MSc immediately after. He has been collaborating in the research activities of the Transactional Systems Research Team (TrxSys) at the Center for Informatics and Information Technologies (CITI) since his first year of the BSc, always under the supervision of Dr. João Lourenço. He also frequently participates in programming contests and is a supporter of Free Software.

Bruno Teixeira is currently working as an IT Consultant at a private corporation in Lisbon, Portugal. He received his MSc in 2010 from the New University of Lisbon with a dissertation entitled “Static Detection of Anomalies in Transactional Memory Programs” and was advised by Dr. João Lourenço.

Ricardo Dias is a PhD student at the Computer Science Department of New University of Lisbon, Portugal. He received his MSc in 2008 from the New University of Lisbon. His current research interests focus on Software Transactional Memory, more specifically in static verification of isolation anomalies in transactional programs.