A Static Approach for Detecting Concurrency Anomalies in Transactional Memory

Bruno Teixeira, João Lourenço, and Diogo Sousa*

CITI — Departamento de Informática,
Universidade Nova de Lisboa, Portugal
bct18897@fct.unl.pt Joao.Lourenco@di.fct.unl.pt
dm.sousa@fct.unl.pt

Abstract. Programs containing concurrency anomalies will most probably exhibit harmful erroneous and unpredictable behaviors. To ensure program correctness, the sources of those anomalies must be located and corrected. Concurrency anomalies in Transactional Memory (TM) programs should also be diagnosed and fixed. In this paper we propose a framework to deal with two different categories of concurrency anomalies in TM. First, we will address *low-level TM anomalies*, also called dataraces, which arise from executing programs in weak isolation. Secondly, we will address *high-level TM anomalies*, also called high-level dataraces, bringing the programmer's attention to pairs of transactions that the programmer has misspecified, and should have been combined into a single transaction. Our framework was validated against a set of programs with well known anomalies and demonstrated high accuracy and effectiveness, thus contributing for improving the correctness of TM programs.

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

1 Introduction

Concurrent programming is inherently hard. The fact that more than one ordering of events may take place at runtime makes it difficult to consider all possible execution scenarios of a program, and may expose unpredicted and harmful behaviors. The most notorious of these concurrency errors is the *datarace*, or *low-level datarace*, which happens 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 using locks, thus establishing a mutual exclusion between certain code blocks of the program.

* 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.

Even in the absence of low-level dataraces, a program may still exhibit concurrency errors which result from incorrect ordering of correctly synchronized critical sections. Such is the case when a thread executes two separate critical sections which are related and should be merged into a single one in order to ensure correctness. We shall call these errors *high-level dataraces* or *high-level anomalies*. In contrast, dataraces, or low-level dataraces, will also be referred in this paper as *low-level anomalies*.

This paper addresses the detection of both low-level and high-level anomalies in the Transactional Memory (TM) [11,17] setting. TM is a promising approach that offers multiple advantages for concurrent programming. In contrast to the usage of locks, which enforces mutual exclusion, TM is neutral concerning the execution model. Memory transactions usually execute optimistically in real concurrency, assuming that no transaction will interfere with another. An underlying TM framework monitors the system and aborts conflicting transactions.

TM is inherently immune to some concurrency errors that storm lock-based programs, such as deadlocks. However, low-level dataraces 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.

There are several approaches to detect dataraces in lock-based programs, both static [6, 9, 13], dynamic [8, 12, 15], and hybrid [16]. Likewise, there are many approaches for high-level anomaly detection [3, 5, 10, 20, 22]. Even though some results could also be applied to TM programs, none of these works targets specifically the TM setting. In this paper, we consider the different nature of TM programs, providing detection approaches for both low-level and high-level anomalies in TM. For this, we discuss how to apply a low-level anomaly detector meant for locks to a TM-based program, and we also propose a new definition for high-level anomalies. We also describe a new static approach for detecting high-level anomalies in TM, which conservatively extracts all possible execution traces of a program and searches for anomalies using a pattern-based approach.

We will discuss the detection of low-level TM anomalies in Sect. 2, and of high-level anomalies in Sect. 3; followed by a discussion of the related work in Sect. 4; and by the conclusions and future work in the last section.

2 Low-Level Dataraces

The TM approach provides several advantages over currently existing concurrency control mechanisms. In particular, the usage of TM alone guarantees the absence of some concurrency errors, such as deadlocks and priority inversion. However, depending on the particular underlying TM system, *dataraces* may still be observed. In this section we show how to detect dataraces in TM by converting transactions into lock-protected critical sections and applying an existing lock-oriented datarace detector.

2.1 Dataraces in Transactional Memory vs. Locks

Locks enforce mutual exclusion between critical sections. If two distinct critical regions are protected by at least one common lock, then no two threads may execute them at the same time. One the other hand, in most cases TM does not enforce mutual exclusion. Instead, two transactional code blocks may execute concurrently, provided with the guarantees of *Isolation* and *Atomicity*. TM provides *serializability* of transactions, ensuring that if two 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.

Consider the distinct situations that may lead to a low-level datarace between two threads, when using locks to control access to shared data:

1. None of the accesses is performed while holding a lock;
2. Only one of the accesses is performed while holding a lock; and
3. Both accesses are performed while holding locks; but there is no common lock shared between them.

With locks, the user chooses which critical section will be mutually exclusive with each other. Because in TM all transactions have the guarantees of Isolation and Atomicity against *all* other concurrent transactions, the third case above does not apply. Hence, as illustrated in Fig. 1, the situations that may lead to 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.

The general idea of our approach is to use an existing datarace detection framework, interpreting atomic blocks as though they were simple lock-based synchronized blocks. We propose an approach to automatically convert transactional blocks into lock-based blocks, all synchronized on a single global lock. Fig. 1 illustrates how each datarace case in TM is to converted to a related situation with locks that result in the same outcome.

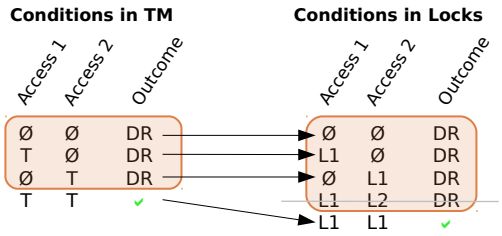

Fig. 1. Conditions for a datarace in TM and Locks

2.2 Detection Approach

Our approach was implemented through the use of AJEX [7], an extension to the Polyglot compiler framework [14] for Java, that already parses atomic blocks. Although our approach is independent from the lock-based datarace detector, in the current implementation we opted to use JChord [13].

The automatic transformation process of TM programs to synchronized single-lock programs consists of three distinct phases:

1. *Parse the source code with the AJEX Polyglot extension.* AJEX already handles atomic blocks, inserting them in the generated program's AST.
2. *Global lock generation.* The global lock must be globally accessible and have an unique, unused name. Our automatic approach automatically decides on how it should be named (a randomly generated name guaranteed to not collide with other already existing names); and where it should be declared (usually in the main class).
3. *Generation of the transformed program.* This new program contains synchronized blocks instead of the original atomic blocks. The datarace detector, JChord, is then invoked on the generated program and the results are presented.

2.3 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 [19]. 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 concurrency benchmark. For each test it was necessary to have both lock-based and TM-based versions, and some of the existing tests meant for locks had to be manually rewritten using TM.

Tests were carried out by initially running JChord on the lock-based version of each test. 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. For each test, the results for both executions of JChord 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 situations could not be modeled using the TM programming model), results were slightly different, but all those differences could be clearly mapped to the semantic variations between the two versions.

2.4 Discussion

We have presented a static approach to detect low-level dataraces in TM programs. This approach is carried out by automatically converting transactions

into proper lock-protected sections, and then invoking a lock-based datarace detector. We have elaborated on the experimental results that show the validity of this solution. The results have demonstrated that it is possible to detect real anomalies in TM programs with our approach. More details on the detection of low-level dataraces in TM programs can be found in [19].

3 High-Level Anomalies

A program that is free of low-level dataraces may still exhibit concurrency errors. Unlike low-level anomalies, high-level anomalies 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 ran in a specific order.

As an example, consider the program in Fig. 2, showing a bounded data structure whose size should not go beyond `MAX_SIZE`. Before a thread asks for an item to be stored, it checks for available room. All accesses to the `list` fields are safely enclosed inside transactions, and therefore no low-level datarace may exist. However, due to interleaving with another thread running the same code, between the executions of `hasSpaceLeft()` and `store()` the size of the list could have changed to the maximum allowed; thus, the first thread would now be adding an element to an already full list. Therefore, both method calls should have been done inside the same transaction.

```
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);
    }
}
```

Fig. 2. Example of a High-level Anomaly

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 detection.

3.1 Thread Atomicity

High-level anomalies are related with sets of transactions involving different threads, which leave the program in an inconsistent state if ran in a specific

order. This happens because two or more transactions executed by a thread are somehow related and make assumptions about each other (e.g., assumption of 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 atomic sections into a single transaction. Furthermore, through empirical observation, it seems that most or all 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 on the intended program semantics, it is not possible to infer at compile time all the relations among transactions. However, it is possible to identify transactions that may or will affect other transactions, and use this information to identify possible high-level anomalies.

Consider a coordinate pair object shared between multiple threads. Let us 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* [22], and will be addressed again in Section 4. It is common to pursue thread atomicity as being a correctness criterion.

3.2 Anomaly Patterns in Transactional Memory Programs

Feeling that full thread atomicity is too restrictive, thus triggering 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 Fig. 3, where each rectangle corresponds to a transactional code block. The second (read) 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.

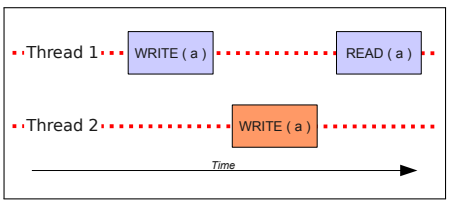

Fig. 3. An unserializable pattern which does not appear to be anomalous.

However, given the specific context of TM and the set of operations presented in the figure, 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 (e.g., assuming them to be equal).

Therefore, 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 suspicious patterns which describe possible high-level anomalies.

Read–write–Read or *RwR* — Non-atomic global read. A thread reads a global state in two or more separate transactions. If it make assumptions based on that state, this 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, but 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*. A thread checks a variable value and, based on that value, alters the state of variable. If the variable is changed meanwhile, the update will probably not make sense anymore.

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 Sect. 3.2. Our approach for the detection of high-level anomalies in TM programs was also implemented using Polyglot framework and AJEX. As described before, Polyglot is a framework for performing transformations and analysis on Java programs, and AJEX is an extension to Polyglot that parses atomic blocks in TM programs.

The thread traces are obtained by symbolic execution. When the program to be analyzed is loaded, all class declarations that contain main thread methods are retrieved. This includes the classes that have a `public static void main (String args[])`, the classes that inherit `java.lang.Thread` or `java.lang.Runnable` and that contain 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.

Challenges arise when the program code is not strictly linear. When a method call is encountered, the solution is to in-line the called code, i.e., to 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 in-linings when in the presence of recursive methods.

Additional challenges derive from disjunctions in the program control flow. When there are multiple alternative paths, such as when using *if* or *case* statements, the current trace must still represent all possible alternative paths. Rather than having numerous alternative traces for the same thread, our approach adds a special disjunction node 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. Therefore, the trace tree considers the cases in which the loop 1) is not executed, 2) is executed once, or 3) is executed twice. It is not necessary to consider more than two consecutive executions, as all the anomalies detected with three or more expansions of the loop body are duplicates of those detected with just two expansions. On the other hand, two expansions of the loop body may yield anomalies that would not be detected with a single expansion. This is the case when the loop body includes two or more transactions that are involved in anomalies among themselves. Also, it is necessary to consider zero executions of the loop body, for the case in which the statements that precede and the ones that follow the loop are both involved in an anomaly.

3.4 Validation of the Approach

We ran a total of 14 tests, consisting of small programs taken from the literature [3, 4, 5, 10, 21], with studied high-level anomalies and that were analyzed by our tool. Additionally, one test consisted on the *Allocate Vector* from the IBM concurrency benchmark repository [1], and another test was developed by ourselves [19]. The results are summarized in Table 1.

In a total of 12 anomalies present in these programs, 10 were correctly pointed out: 2 *RwR* anomalies, 3 *WrW* and 5 *RwW*.

The false negatives were not due to imprecision of the anomaly patterns, but rather to data accesses in JRE classes, whose source code is not available. Those JRE methods may possibly read or update internal data, but since their code is not available, these methods are ignored, thus missing potential anomalies. As a possible approach to solve this problem, these unavailable methods could be assumed to read and modify the involved objects.

Table 1. Test results summary.

Test Name	Total Anomalies	Total Warnings	Correct Warnings	False Warnings	Missed Anomalies
Connection [5]	2	2	1	1	1
Coordinates'03 [3]	1	4	1	3	0
Local Variable [3]	1	2	1	1	0
NASA [3]	1	1	1	0	0
Coordinates'04 [4]	1	4	1	3	0
Buffer [4]	0	7	0	7	0
Double-Check [4]	0	2	0	2	0
StringBuffer [10]	1	0	0	0	1
Account [21]	1	1	1	0	0
Jigsaw [21]	1	2	1	1	0
Over-reporting [21]	0	2	0	2	0
Under-reporting [21]	1	1	1	0	0
Allocate Vector [1]	1	2	1	1	0
Knight Moves [19]	1	3	1	2	0
Total	**12**	**33**	**10**	**23**	**2**

In addition to the correctly detected anomalies, there were also 23 false positives (70% of total warnings). We group the causes for these imprecisions in 4 different categories.

First, out of these 23 false warnings, 5 were due to redundant reading operations. In a read operation `object.field`, two readings are actually being performed: `object` and `field`. It makes no sense that two instances of this statement be involved in a *RwR*. It would be possible to eliminate these false positives if the accesses were considered the same.

Another 6 false positives are related to cases for which additional semantic information would have to be provided or inferred. These false warnings could be eliminated with the aid of other available techniques, such as pointer analysis.

Of the remaining false positives, 10 could be eliminated by refining the definition of the *anomaly patterns* described in Section 3.2, with alterations that are indeed intuitive. For example, an *RwR* anomaly could be ignored if the second transaction would write both values involved. However, these alterations should be made carefully, as they could harm the overall behavior in other tests.

Finally, the remaining 2 false reports are also related to correct accesses which are matched by our anomaly patterns. Further study would be necessary to adapt the anomaly patterns in order to leave out these correct accesses, without seriously compromising the precision of the approach.

3.5 Discussion

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

have defined and implemented a new approach to static detection of high-level concurrency anomalies in Transactional Memory programs. This new approach works by conservatively tracing transactions and matching the interference between each consecutive pair of transactions against a set of defined anomaly patterns. Our approach raises false positives, although at an acceptable level. When compared with the existing reports from the literature, these results are somewhat better. The two false negatives were related to the unavailability of the source code and not to the inadequacy of the anomaly patters. 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. More details on the detection of high-level dataraces in TM programs can be found in [18, 19].

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, 12, 13, 15, 16]. We are unaware of any work that specifically targets TM; however, we have shown that current algorithms, which are mainly intended for use with lock-based mechanisms, may as well be applied to transformed TM programs.

Although high-level anomaly detection is not such a hot topic, there are some relevant works which share some principles and features with our own. One of the earliest works on the subject is the one by Wang and Stoller [22]. They introduce the concept of thread atomicity, *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].

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 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 which concern only 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 lowering its false positives. 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 [20]. 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

We have proposed a framework for the detection of both low-level and high-level anomalies in Transactional Memory programs. The framework resorts to statical analysis of the program's source code to detect and report those anomalies.

The methodology used to detect low-level dataraces, based in a source-to-source transformation of a TM program to a lock-based one, was proven to provide adequate results, thus being a possible strategy to detect this kind of anomalies.

The methodology used to detect high-level dataraces, based in static analysis and symbolic code execution, and matching transactions' interleavings with suspicious patterns, has also provided good quality results, comparable to or even better than those reported in the literature for analogous problems in lock-based programs.

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.

The developed tool could be improved by further refining the error patterns. The addition of *points-to* and *happens-in-parallel* analyses would also 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, as discussed in Sect. 3.4.

References

1. IBM's Concurrency Testing Repository. `https://qp.research.ibm.com/concurrency_testing`.
2. Mohammad Ansari et al. Lee-TM: A non-trivial benchmark suite for transactional memory. In *Proceedings of ICA3PP '08*, pages 196–207, Berlin, 2008. Springer-Verlag.

3. Cyrille Artho, Klaus Havelund, and Armin Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4):207–227, 2003.
4. 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.
5. Nels E. Beckman, Kevin Bierhoff, and Jonathan Aldrich. Verifying correct usage of atomic blocks and typestate. *SIGPLAN Not.*, 43(10):227–244, 2008.
6. Jong deok Choi, Alexey Loginov, Vivek Sarkar, and Alexey Logthor. Static datarace analysis for multithreaded object-oriented programs. Technical report, IBM Research Division, Thomas J. Watson Research Centre, 2001.
7. Ricardo Dias and Bruno Teixeira. Ajex: A source-to-source java stm framework compiler. Technical report, DI-FCT/UNL, April 2009.
8. Anne Dinning and Edith Schonberg. Detecting access anomalies in programs with critical sections. *SIGPLAN Not.*, 26(12):85–96, 1991.
9. Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. *SIGPLAN Not.*, 35(5):219–232, 2000.
10. Cormac Flanagan and Stephen N Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proceedings of POPL'04*, pages 256–267, New York, NY, USA, 2004. ACM.
11. 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.
12. Daniel Marino, Madanlal Musuvathi, and Satish Narayanasamy. Literace: effective sampling for lightweight data-race detection. In *Proceedings of PLDI'09*, pages 134–143, New York, NY, USA, 2009. ACM.
13. Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for java. In *PLDI*, pages 308–319. ACM Press, 2006.
14. Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *CC*, pages 138–152, 2003.
15. Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. *SIGPLAN Not.*, 38(10):167–178, 2003.
16. Yao Qi, Raja Das, Zhi Da Luo, and Martin Trotter. Multicoresdk: a practical and efficient data race detector for real-world applications. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems*, pages 1–11, New York, NY, USA, 2009. ACM.
17. Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of PODC'95*, pages 204–213, New York, NY, USA, 1995. ACM.
18. B. Teixeira, D. Sousa, J. Lourenço, R. Dias, and E. Farchi. Detection of transactional memory anomalies using static analysis. In *Proceedings of PADTAD'10*, pages 26–36, New York, NY, USA, 2010. ACM.
19. Bruno Teixeira. Static detection of anomalies in transactional memory programs. Master's thesis, Universidade Nova de Lisboa, April 2010.
20. Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of POPL'06*, pages 334–345, New York, NY, USA, 2006. ACM.
21. Christoph von Praun and Thomas R. Gross. Static detection of atomicity violations in object-oriented programs. In *Journal of Object Technology*, page 2004, 2003.
22. 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).