DATA RACE DETECTION

Literature Review

Rahul Thapar
Computer Science Department
University of Texas, Arlington
1001768619
rahul.thapar@mavs.uta.edu

Abstract

Bugs due to data races in multithreaded programs often show non-deterministic properties and are very difficult to find but their detection is highly essential for debugging multithreaded programs and assuring their correctness. According to the papers used for this review, there is no single universal technique that is capable of detecting all data race conditions efficiently, since the data race detection problem is computationally hard in the general case. All currently available tools, when applied to simple case programs, usually produce false alarms or a large number of undetected races. Another major drawback of currently available tools is that they are restricted, for performance reasons, to detection units of fixed size. Thus, they all suffer from the same problem—choosing a small unit might result in missing some of the data races, while choosing a large one might lead to false detection.

This paper aims to discuss about the differences between static and dynamic data race detectors, introduce dynamic data race detectors available in the market, compare them on the basis of different benchmarks and apply them to the Eclipse environment to conclude which data race detector performs well under which circumstance. The paper also discusses about the different approaches and algorithms that these dynamic tools use, why they are better than the other and which tool can be used in certain situation.

1. Introduction

Multithreading is of the common programming paradigms that works well for multiprocessor environments. Parallelism and improved performance makes multithreading advantageous over single threaded programming. However, multithreading also introduces the problem of data races. A data race occurs when two or more threads concurrently access a shared location, and at least one of the accesses is for writing. Such a situation is usually considered to be an error (a.k.a. a bug). This is an undesirable case, as it might lead to false and unpredictable results and thus lead to incorrect program execution. Data races usually occur due to errors made by programmers who fail to implement synchronization correctly in the program. The problem of deciding if a given program contains potential data races is computationally very hard. Feasible data races are defined as races that are based on the possible behaviour of the program (i.e., the semantics of the program computation). These are the real races that might happen in any specific execution of a program. The problem of exactly locating feasible data races is NP-hard in the general case. Since data races are usually a result of improper synchronization that does not prevent concurrency of accesses. These are approximations of feasible data races, based on the behaviour of the explicit synchronization only; they are defined in the context of a specific program execution. Apparent data races are simpler to locate than feasible data races, but they are also less accurate than the latter. It was proved that apparent races exist if and only if at least one feasible race exists somewhere in the execution. Yet, according to some papers, the problem of exhaustively locating all apparent data races is still NP-hard.

The rest of the paper is organised as follows. Section 2 discuss about the static and dynamic data race detection techniques. Section 3 presents various dynamic detection tools, their underlying approach, algorithm and performance. Section 4 compares the tools with each other and give performance results. Section 5 implements FastTrack on Eclipse environment to evaluate the performance. The paper concludes with the final results about FastTrack.

2. Static vs Dynamic

Data race detection tools generally rely on one of static and dynamic techniques. The key difference between static and dynamic techniques boils down to their information point. Static techniques use program information to report data races from source code without any execution. On the contrary, Dynamic techniques use execution information from the program to identify data races. The major drawback of static race detection techniques is that a large number of fraudulent data race warnings can often be generated which do not correspond to true bugs. Static detectors are sound, but imprecise since they report too many false positives. Dynamic detectors are precise or imprecise, but unsound since they cannot guarantee to locate the existence of at least one data race in a given execution of the program if there exists any.

Dynamic detectors employ trace based post-mortem methods or on-the-fly methods, which report data races occurred in an execution of a programs. Post-mortem methods analyse the traced information or re-execute the program after an execution. On-the-fly methods are based on three different analysis methods: lockset analysis, happens-before analysis, and hybrid analysis.

The lockset analysis is simple and can be implemented with low overhead. However, lockset analysis may lead to many false positives, because it ignores synchronization primitives which are non-common lock such as signal/wait, fork/join, and barriers. The happens-before analysis is precise, since it does not report false positives and can be applied to all synchronization primitives. However, it is quite difficult to be efficiently implemented due to the performance overheads. The hybrid method tries to reduce the main drawback of pure lockset analysis and to get more improved performance than pure happens-before analysis.

Many dynamic data race detection techniques are used in automation tools because of the advantages they offer. These techniques used in detectors also have some limitations, because they analyse only the dynamic execution of a program with a single input. Most dynamic detectors try to cover the limitations by considering the ordering of synchronization operations, such as fork-join, locks, signal-waits, and barriers, obtained in an actual execution of the program, but they still provide limited advantages (e.g. supporting particular synchronization primitives, improving the efficiency or the preciseness of execution overhead, etc.). The next section introduces five dynamic race detectors, their technical approach, evaluation followed by comparison with each other.

3. Dynamic Race Detection Tools, Evaluation and Comparison

FastTrack

FastTrack [Flanagan and Freund 2010a] is state-of-the art happens-before based race predictor. It improves the performance of more heavy-weight dynamic analysis tools by
identifying millions of irrelevant, race-free memory accesses that can be ignored. FastTrack is an order of magnitude faster than a traditional vector-clock race detector, and roughly twice as fast as the high-performance DJIT+ algorithm. It is even comparable in speed to ERASER.
The key approach behind FAST-TRACK is that the full generality of vector clocks is not necessary in over 99% of these read and write operations: a more lightweight representation of the happens-before information can be used in-stead. Only a small fraction of operations performed by the target program necessitate expensive vector clock operations.
FastTrack has time and space complexities that depend on the maximum parallelism of the program to partially maintain expensive data structures, such as vector clocks.

GoldiLocks

Goldilocks is based solely on the concept of locksets and is able to capture all mutual-exclusion synchronization idioms uniformly with one mechanism. The algorithm can be used, both in the static or the dynamic context, to develop analyses for concurrent programs, particularly those for detecting data-races, atomicity violations, and failures of safety speculations. Goldilocks combines the precision of vector clocks with the computational efficiency of locksets.

They key approach of this detector is to detect races at runtime, Goldilocks approach requires a transaction manager to provide or make possible for the runtime to collect for each transaction commit (R, W) the sets R and W and the place of commit point of the transaction in the global synchronization order.

In order to verify the performance of GoldiLocks in the Kaffe, eleven benchmarks programs were used. Six of the benchmarks were from the Java Grande Forum Benchmark Suite. The paper reduced the size of inputs to the Grande benchmarks because of the long running time of the applications.

Multi-Race

Multi-Race is a novel testing tool which combines improved versions of Djit and Lockset—two very powerful on-the-fly algorithms for dynamic detection of apparent data races. Both extended algorithms detect races in multithreaded programs that may execute on weak consistency systems, and may use two-way as well as global synchronization primitives.

By exploiting a unique configuration of memory mappings, called views, and the technique of pointer sizzling [4, 8], Multi-Race detects data races in granularity of objects in the program, rather than in fixed-size units. To the best of our knowledge, it is the first entirely transparent on-the-fly framework for multithreaded environments that is capable of doing so. Multi-Race carries out this task with the help of automatic and transparent source code instrumentation. In this approach, the code of the tested program, written in C++, is pre-processed, modified, and recompiled, such that calls to detection mechanisms are injected in places where accesses to shared locations are performed.

The Multi-Race overheads measured for six classical benchmark applications: Integer Sort (IS), Water-nsquad

(WATER), LU-contiguous (LU), Fast Fourier Transform (FFT), Successive Over-Relaxation (SOR) and the Traveling Salesman Problem (TSP). For evaluation of overheads the paper used data-race-free versions of the applications. Therefore, the paper was able to place each of the allocated arrays on single mini pages (the default configuration of Multi-Race).

Djit+

Djit+, is a revised version of the earlier Djit algorithm. Djit+ can correctly operate on weakly ordered systems, such as the one presented by Adve and Hill in,and still detect a greater number of races as they occur in the program's execution. Djit+ relies on a formal framework called vector time frames, which is based on Mattern's virtual time vector time-stamps. The algorithm also assumes the existence of some logging mechanism (to be described later), capable of dynamically recording all accesses to each of the shared memory locations. The general idea of the algorithm is to log every shared access and to check whether it "happens-before" prior accesses to same location.
The approach Djit+ algorithm follows can be described as:

- Each thread has its own clock that is incremented at lock synchronization operations with release semantics.
- Each thread also keeps a vector clock Ct.
- For a thread u, Ct (u) gives the clock for the last operation of u that happened before the current operation of t.
- Each lock has a vector clock.
- Each shared variable x has two vector clocks Rx and Wx.

Upon initialization:
1. Each initializing thread t fills its vector of time frames with ones—∀ i : stt[i] ← 1.

2. The access history of each shared location v is filled with zeros (since no thread has accessed it yet)— ∀ i : arv[i] ← 0, awv[i] ← 0.

3. The vector of each synchronization object S is filled with zeros—∀ i : stS[i] ← 0.

Upon an acquire of synchronization object S :
1. The issuing thread t updates each entry in its vector to hold the maximum between its current value and that of S's vector—∀ i : stt[i] ← max(stt[i],stS[i]).

Upon a release of synchronization object S:
1. The issuing thread t starts a new time frame. Therefore, it increments the entry corresponding to t in t's vector—stt[t] ← stt[t]+1.
2. Each entry in S's vector is updated to hold the maximum between the current value and that of t's vector—∀ i : stS[i] ← max(stt[i],stS[i]).

Upon a first access to a shared location v in a time frame or a first write to v in a time frame :
1. The issuing thread t updates the relevant entry in the history of v. If the access is a read, it performs arv[t] ← stt[t]. Otherwise, it performs awv[t] ← stt[t].
2. If the access is a read, thread t checks whether there exists another thread u which also wrote to v, such that awv[u] ≥ stt[u]. In other words, t checks whether it knows only about a release that preceded the write in u, and if so reports a data race. If the access is a write, thread t checks whether there exists another thread u, such that awv[u] ≥ stt[u] or arv[u] ≥ stt[u]. In other words, t checks all reads as well as all writes by other threads to v.

Figure 1. The full Djit+ protocol

Eraser

Eraser is a tool which is used for dynamically detecting data races in lock-based multithreaded programs. Eraser uses binary rewriting techniques to monitor every shared-memory reference and verify that consistent locking behaviour is observed.
Approach, Eraser simply checks that all shared-memory accesses follow a consistent locking discipline. A locking discipline is a programming policy that ensures the absence of data races. For example, a simple locking discipline is to require that every variable shared between threads is protected by a mutual exclusion lock. The paper argues that for many programs Eraser's approach of enforcing a locking discipline is simpler, more efficient, and more thorough at catching races than the approach based on happens-before. According to the paper, Eraser is the first dynamic race detection tool to be applied to multithreaded production servers.

The key idea behind it is to track lock sets that govern access to each shared location. A data race is an access to a shared variable that is not governed by lock(s). It finds more races then happens-before based tools, but still causes 10-30x slow-down.

Precision and Performance

To ensure reliable comparisons, all tools were implemented on top of ROAD-RUNNER as similarly as possible. For example, BASICVC, DJIT+, MULTIRACE, and FASTTRACK use the same vector clock implementation, and all checkers were profiled and tuned to eliminate unnecessary bottlenecks. The paper describes that several additional techniques could be used to improve the performance of all these checkers. For example, a separate static analysis could be included to reduce the need for run-time checks; (2) include a separate dynamic escape analysis; (3) add un-sound optimizations; (4) tune ROADRUNNER to better support one particular kind of analysis; (5) implement these checkers directly on a JVM rather than on top of ROADRUNNER; (6) implement the checkers inside the JVM itself (sacrificing portability, maintainability, etc.); or (7) at the extreme, implement the checkers directly in hardware. The paper suggests that these techniques would improve performance in ways that are orthogonal to the central contributions of the paper, and at the cost of additional complexity. In order to present the results, the paper does not take these complementary optimizations into consideration.

Benchmark

Experiments were performed on 16 benchmarks: elevator, a discrete event simulator for elevators; hedc, a tool to access astrophysics data from Web sources ;tsp, a Traveling Salesman Problem solver ; mtrt, a multithreaded ray-tracing program from the SPEC JVM98benchmark suite ;jbb, the SPEC JBB2000 business object simulator ; crypt, lufact, sparse, series, sor, moldyn, Monte Carlo, and raytracer from the Java Grande benchmark suite ;the colt scientific computing library; the raja ray tracer ; and philo, a dining philosophers simulation.The Java Grande benchmarks were configured to use four worker threads and the largest data set provided (except crypt, for which we used the smallest data set because BASICVC, DJIT+, and MULTIRACE ran out of memory on the larger sizes).All experiments were performed on an Apple Mac Pro with dual3GHz quad-core Pentium Xeon processors and 12GB of memory, running OS X 10.5.6 and Sun's Java Hotspot 64-bit Server VMversion 1.6.0. All classes loaded by the benchmark programs were instrumented, except those from the standard Java libraries. The timing measurements include the time to load, instrument, and execute the target program, but it excludes JVM start-up time to reduce noise. The tools report at most one race for each field of each class, and at most one race for each array access in the program source code.

4. Performance Comparison

FastTrack vs Eraser

The performance results show that implementation of ERASER incurs an overhead of 8.7x, which is competitive with similar Eraser implementations built on top of unmodified JVMs. Surprisingly, FASTTRACK is slightly faster than ERASER on some programs, even though it performs a precise analysis that traditionally has been considered more expensive. More significantly, ERASER reported many spurious warnings that do not correspond to actual races. Augmenting the ERASER implementation to reason about additional synchronization constructs would eliminate some of these spurious warnings, but not all. Onhedc, ERASER reported a spurious warning and also missed two of the real race conditions reported by FASTTRACK, due to an (intentional) unsoundness in how the Eraser algorithm reasons about thread-local and read-shared data.

FastTrack vs DJIT+ and BASIC-VC

DJIT+ and BASICVC reported exactly the same race conditions as FASTTRACK. That is, the three checkers all yield identical precision. In terms of performance, however, the results show that FASTTRACK significantly out performs the other checkers. In particular, it is roughly 10xfaster than BASICVC and 2.3x faster than DJIT+. These performance improvements are due primarily to the reduction in the allocation and use of vector clocks. Over all the benchmarks, DJIT+ allocated more over 790 million vector clocks, whereas FASTTRACK allocated only 5.1 million. DJIT+ performed over 5.1 billion O(n)-time vector clock operations, while FAST-TRACK performed only 17 million. The memory overhead for storing the extra vector clocks leads to significant cache performance degradation in some programs, particularly those that perform random accesses to large arrays.

FastTrack vs MultiRace

MULTIRACE maintains DJIT+'s instrumentation state, as well as a lock set for each memory location . The checker updates the lock set for a location on the first access in an epoch, and full vector clock comparisons are performed after this lock set becomes empty. This synthesis substantially reduces the number of vector clock operations, but introduces the overhead of storing and updating lock sets. In addition, the use of ERASER's unsound state machine for thread-local and read-shared data leads to imprecision. In combination with a coarse-grain analysis, this approach produced substantial performance improvement. The re-implementation of the MULTIRACE algorithm in ROAD-RUNNER used fine-grain analysis and exhibited performance com-parable to DJIT+. Interestingly, over all benchmarks MULTIRACE performed less than half the number of VC operations as FAST-TRACK. However, this did not lead to speed-ups over DJIT+, because the memory footprint was even larger than DJIT+, leading to substantial cache performance degradation. Additionally, on average roughly 10% of all operations required an ERASER operation that also imposed some additional overhead.

FastTrack vs GoldiLocks

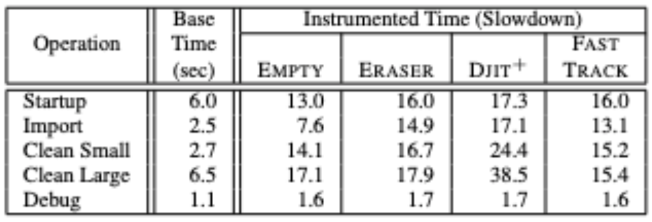
GOLDILOCKS is a precise race detector that does not use vector clocks to capture the happens-before relation. Instead, it maintains, for each memory location, a set of "synchronization devices" and threads.

A thread in that set can safely access the memory location, and a thread can add itself to the set (and possibly remove others) by performing any of the operations described by the synchronization devices in the set. GOLDILOCKS is a complicated algorithm that required 1,900 lines of code to implement in ROADRUNNER, as compared to fewer than 1,000 lines for each other tool. GOLDILOCKS also ideally requires tight integration with the underlying virtual machine and, in particular, with the garbage collector, which is not possible un-der ROADRUNNER. Both of these factors cause GOLDILOCKS to incur a high slowdown. GOLDILOCKS implemented in ROADRUNNER incurred a slowdown of 31.6x across the benchmarks (but ran out of memory on lufact), even when utilizing an unsound extension to handle thread-local data efficiently. (This extension caused it to miss the three races in hedc found by other tools.) According to the paper some performance improvements are possible, for both GOLDILOCKS and the other tools, by integration into the virtual machine. As another data point, the original GOLDILOCKS study report edits slowdown for the compute-intensive benchmarks in common to be roughly 4.5x. In nutshell, GOLDILOCKS is an interesting algorithm but its complexity and JVM-integration issues make it difficult to implement efficiently, and so GOLDILOCKS may not provide significant performance benefits over FASTTRACK.

5. FastTrack on Eclipse Environment to check Race Conditions

To validate FASTTRACK in a more realistic setting, the paper applied it to the Eclipse development environment, version 3.4.0 . Two lines of source code were modified to report the time to perform each user-initiated operation. No other modifications were made. Sun's Java 1.5.0 Hotspot Client VM was used because the test platform must run Eclipse as a 32-bit application with a maximum heap size of 2GB.
The experiments were performed with the following five Eclipse operations:

Startup: Launch Eclipse and load a workspace containing four projects with 65,000 lines of code.
Import: Import and perform an initial build on a project containing 23,000 lines of code.
Clean Small: Rebuild a workspace with four projects containing a total of 65,000 lines of code.
Clean Large: Rebuild a workspace with one project containing 290,000 lines of code.
Debug: Launch the debugger and run a small program that immediately throws an exception.



Operation	Base Time (sec)	Instrumented Time (Slowdown)			
		EMPTY	ERASER	DJIT$^+$	FAST TRACK
Startup	6.0	13.0	16.0	17.3	16.0
Import	2.5	7.6	14.9	17.1	13.1
Clean Small	2.7	14.1	16.7	24.4	15.2
Clean Large	6.5	17.1	17.9	38.5	15.4
Debug	1.1	1.6	1.7	1.7	1.6

Benchmark for the evaluation: For these tests, FASTTRACK loaded and instrumented roughly 6,000 classes corresponding to over 140,000 lines of source code. Several classes using reflection were not instrumented to avoid current limitations of ROADRUNNER, but these classes did not involve interesting concurrency. Native methods were also not instrumented. Eclipse used up to 24 concurrent threads during these tests.

6. Result:

Eclipse startup time was heavily impacted by the byte code instrumentation process, which accounted for about 75% of the observed slowdowns. While ROADRUNNER does not currently support off-line byte code instrumentation, that feature would remove a substantial part of startup time.

1. ERASER reported potential races on 960 distinct field and array accesses for these five tests, largely because Eclipse uses many synchronization idioms, such as wait/notify, semaphores, readers-writer locks, etc. that ERASER cannot handle. Additional extensions could handle some of these cases.
2. FASTTRACK reported 30 distinct warnings. While the paper has not been able to fully verify the correctness of all code involved in these races, none caused major failures or data corruption during the tests.
3. DJIT+ reported 28 warnings. These overlapped heavily with those reported by FASTTRACK, but scheduling differences led to several being missed and several new (benign) races being identified. The items listed above were reported by both tools.

7. Conclusion

FASTTRACK performed quite well on the three most compute-intensive tests (Import, Clean Small, and Clean Large), exhibiting performance better than DJIT+ and comparable to ERASER. Monitoring the Debug test incurred less overhead for all tools because that test spent much of its time starting up the target VM and switching views on the screen. FASTTRACK is able to scale to precisely check large applications with lower run-time and memory overheads than existing tools.

8. Acknowledgement

I would like to thank the professor, Dr. Lei and the TA, Ana Jovanovic for their valuable guidance and feedback throughout the course. This work was also influenced by the feedback given by other students during the class presentation.

References

Y. Lei is with the Department of Computer Science and Engineering, University of Texas at Arlington, Arlington, TX 76019. E-mail: ylei@cse.uta.edu Reachability Testing of Concurrent Programs (2006)

Al-Zain, A.D., Trinder, P.W. and Hammond, K. (2012) 'Orchestrating computational algebra components into a high-performance parallel system', International of High Performance Computing and Networking, Vol. 7, No. 2, pp.76–86.
(PDF) Evaluation and comparison of ten data race detection techniques. Available from: https://www.researchgate.net/publication/319672758_Evaluation_and_comparison_of_ten_data_race_detection_techniques [accessed Jul 08 2020].

Ma, Peijun (). Evaluation and comparison of ten data race detection techniques. , 10, 279-. https://www.researchgate.net/publication/319672758_Evaluation_and_comparison_of_ten_data_race_detection_techniques

Tong Zhang, Changhee Jung, Dongyoon Lee ProRace: Practical Data Race Detection for Production Use https://people.cs.vt.edu/~dongyoon/papers/ASPLOS-17-ProRace.pdf

Tayfun Elmas, Shaz Qadeer, Serdar Tasiran Goldilocks: A Race and Transaction-Aware Java Runtime https://www.researchgate.net/publication/220752265_Goldilocks_A_race_and_transaction-aware_Java_runtime

Gautam Altekar and Ion Stoica. Odr: output-deterministic replay for multicore debugging. In Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, pages 193–206. ACM, 2009.

Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Efficient, software-only data race exceptions. In Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '15, 2015.

Michael D. Bond, Katherine E. Coons, and Kathryn S. McKinley. Pacer: Proportional detection of data races. In Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '10, pages 255–268, 2010

Yuan Yu, Tom Rodeheffer, Wei Chen RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking http://web.eecs.umich.edu/~mosharaf/Readings/RaceTrack.pdf

C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA), Oct. 2001

J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise data race detection for object oriented programs. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pages 285–297, 2002

H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. PLDI '08, pages 68–78, New York, NY, USA, 2008. ACM

T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware Java runtime. PLDI '07, pages 245–255

J. Zhao, S. Nagarakatte, M. M. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. POPL '12, pages 427–440, New York, NY, USA, 2012. ACM

The Coq Development Team. The Coq Proof Assistant Reference Manual (Version 8.5), 2016. URL https://coq. inria.fr/refman/

E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. PPoPP '03, pages 179–190, New York, NY, USA, 2003. ACM

G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, Faster, Stronger SFI for the x86. PLDI '12, pages 395–404, New York, NY, USA, 2012. ACM

J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. POPL '05, pages 378–391, New York, NY, USA, 2005. ACM

W. Mansky, D. Garbuzov, and S. Zdancewic. An Axiomatic Specification for Sequential Memory Models. CAV '15, pages 413–428, 2015

X. Leroy and S. Blazy. Formal Verification of a C-like Memory Model and Its Uses for Verifying Program Transformations. J. Autom. Reason., 41:1–31, July 2008.

L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. Commun. ACM, 21(7):558–565, July 1978

A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward Integration of Data Race Detection in DSM Systems. J. Parallel Distrib. Comput., 59(2):180–203, Nov. 1999

R. O'Callahan and J.D. Choi, "Hybrid Dynamic Data Race Detection," Symp. Principles Practice Parallel Programming, San Diego, CA, USA, June 11–13, 2003, pp. 167–178.

M.S. Yu and D.H. Bae, "SimpleLock+: Fast and Accurate Hybrid Data Race Detection," Comput. J., vol. 59, no. 6, 2016, pp. 793– 809.

J. Huang, P.O. Meredith, and G. Rosu, "Maximal Sound Predictive Race Detection with Control Flow Abstraction," Conf. Programming Language Des. Implementation, Edinburgh, Ireland, June 9–11, 2014, pp. 337–348

M. Naik, A. Aiken, and J. Whaley, "Effective Static Race Detection for Java," Conf. Appli, Tampa Bay, FL, USA, Oct. 14–18, 2001, pp. 56–69. Programming Language Des. Implementation, Ottawa, Canada, June 10–16, 2006, pp. 308–319

M. Naik and A. Aiken, "Conditional Must Not Aliasing for Static Race Detection," Symp. Principles Programming Languages, Nice, France, Jan. 17–19, 2007, pp. 327–338

B.P. Wood, L. Ceze, and D. Grossman, "Low-Level Detection of Language-Level Data Races with LARD," Conf. Archit. Support Programming Languages Operating Syst., Salt Lake, Canada, Mar. 1–5, 2014, pp. 671–686

C. Boyapati and M. Rinard, "A Parameterized Type System for Race-Free Java Programs," Conf. Object-Oriented Programming, Syst. Languages,

(PDF) Evaluation and comparison of ten data race detection techniques. Available from: https://www.researchgate.net/publication/319672758_Evaluation_and_comparison_of_ten_data_race_detection_techniques [accessed Jul 08 2020].