FRESA: A Frequency-Sensitive Sampling-Based Approach for Data Race Detection

Neng Huang, Zhiyuan Shao, Hai Jin

FRESA: A Frequency-sensitive Sampling-based Approach for Data Race Detection

Neng Huang, Zhiyuan Shao, and Hai Jin

Services Computing Technology and System Lab
Cluster and Grid Computing Lab
School of Computer Science and Technology
Huazhong University of Science and Technology, Wuhan, 430074, China
{nenghuang,zyshao,hjin}@hust.edu.cn

Abstract. Concurrent programs are difficult to debug due to the inherent concurrence and indeterminism. One of the problems is race conditions. Previous work on dynamic race detection includes fast but imprecise methods that report false alarms, and slow but precise ones that never report false alarms. Some researchers have combined these two methods. However, the overhead is still massive. This paper exploits the insight that full record on detector is unnecessary in most cases. Even prior sampling method has something to do to reduce overhead with precision guaranteed. That is, we can use a frequency-sensitive sampling approach. With our model on sampling dispatch, we can drop most unnecessary detection overhead. Experiment results on DaCapo benchmarks show that our heuristic sampling race detector is performance-faster and overhead-lower than traditional race detectors with no loss in precision, while never reporting false alarms.

Keywords: Concurrency, Data Race, Sampling, Bug Detection

1 Introduction

Multithreading is getting more and more popular in today's software. In other words, software must become more parallel to exploit hardware trends, which are increasing the number of processors on each chip. Unfortunately, correct and scalable multithread programming is quite difficult. The instructions in different threads can be interleaved randomly. A data race occurs when two different threads access the same memory location without an ordering constraint enforced between the accesses, and at least one of the accesses is a write [18]. Data races are not necessary errors in and of themselves, but they indicate a variety of serious concurrency errors that are difficult to reproduce and debug such as atomicity violations [13], order violations [12], and sequential consistency violations [14]. As some races occur only under certain inputs, environments, or thread schedules, deploying low-overhead and precise-coverage race detection tool is necessary to achieve highly robust deployed software.

There has been much effort to develop automatic tools for detecting data races. Ultimately, the detection techniques are broadly categorized according to

the time they are applied to the program: static and dynamic. Static techniques [11,17,21,22,24,29] provide maximum coverage by reasoning about data races on all execution paths. However, they tend to make conservative assumptions that lead to a large number of false data races alarm. On the other hand, dynamic techniques [6, 7, 10, 20, 26, 30, 31] are more precise than static tools, but their coverage is limited to the paths and thread interleaving explored at runtime. In practice, the coverage of dynamic tools can be increased by running more tests.

Most dynamic data race detectors are not widely used due to their runtime overhead. Data race detectors like RaceTrack [30] incurs about 2x to 3x slowdown and Intel's Thread Checker [23] takes performance overhead on the order of 200x. Such large performance overhead leads to the lack of data race detectors used in practice. The main reason for this very large performance overhead is each memory operation executed by the program needs to be recorded and analyzed. LiteRace [15] uses sampling to reduce the runtime overhead of data race detector. It presents a sampling algorithm which based on the cold-region hypothesis that data races are likely to occur when a thread is executing a "cold" (infrequently accessed) region in the program. PACER [3] shows a "get what you pay for" approach that provides scalable performance and scalable odds of finding any race. It provides a qualitative improvement over prior approaches.

This paper presents a frequency-sensitive sampling-based approach called FRESA. FRESA makes a precise and coverage guarantee: no matter what sampling methods you used, it learns and updates its sampling strategy intelligently. In other words, "get what you want but pay for less".

FRESA collects and organizes historical sampling results for the next sampling. In order to make the sampler more effective, it owns a finding density table in each schedule path. Compared with previous table, FRESA computes and creates a sampling probability interval, which helps making a more appropriate proportional sampling rate.

The rest of the paper is organized as follows: section 2 introduces our motivation and section 3 describes our algorithm. In section 4, we show our experimental results and performance. We list some related work in section 5. Then we conclude our current work and future work in section 6.

2 Motivation

We motivate our work by a common program shown below.

```
public class Test{
    int test1,test2,test3;
    void func1(int x){
        test1 += x;
    }
    void func2(int x){
        test2 += x;
    }
```

```
    void func3(int x){
        test3 += x;
    }
}

public class TestTest{
    Test test = new Test();
    Thread t1 = new Thread(){     //thread t1
        public void run(){
            for(int i = 0; i < 10000; i++)
                test.func1(1);
            test.func2(1);
            test.func3(1);
            }
        }
    Thread t2 = new Thread(){     //thread t2
        public void run(){
            test.func1(1);
            for(int i = 0; i < 100; i++)
                test.func2(1);
            test.func3(1);
            }
        }
    Thread t3 = new Thread(){     //thread t3
        public void run(){
            test.func1(1);
            test.func2(1);
            test.func3(1);
            }
        }

    public static void main(){
        t1.start();
        t2.start();
        t3.start();
        t1.join();
        t2.join();
        t3.join();
    }
}
```

In the program the shared memory *test1*, *test2*, *test3* are accessed 10002, 102, 3 times respectively, which shows an asymmetrically accessed distribution by *thread1*, *thread2* and *thread3*. The asymmetry like the above program is quite common in real applications.

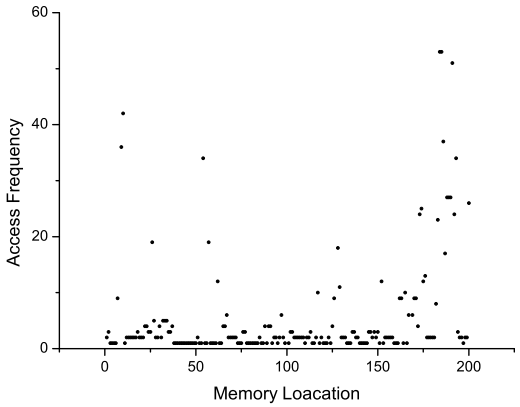


Fig. 1. A random selected 200 sequential accesses in DaCapo eclipse.

We have an instrumentation in DaCapo eclipse. Figure 1 lists a random selected 200 sequential accesses frequency. Access frequency imbalance is quite obviously as shown. Previous race detection methods based on sampling never take it into account. Obviously, quite a lot overhead could be reduced through asymmetrical sampling rate.

3 FRESA Algorithm

In order to leverage the asymmetry. This section presents our FRESA algorithm. FRESA is a dynamic data race detection sampling method based on happen-before relationship, which uses a frequency statistics for the next time's sampling based on asymmetrical access information.

FRESA starts at a user-given sampling rate r_0 for every shared memory in sampling areas. It collects shared memory access frequency at run time. With the program execution, different shared memory's access frequency appears different. FRESA groups all the memories into different groups by considering their access frequencies. As more and more different memories grouped, each group can have a unique sampling rate based on user-given sampling rate. FRESA believes that the more frequently the shared memory accessed, the lower proportion the shared memory could have a data race. Since data races that occur in frequently accessed memory of well-tested programs either have already been found or fixed.

3.1 Frequency Statistics

FRESA collects the frequency of a memory location accessed, with the aim of grouping the memory of similar access frequency into the same group. By

this means, different memories can be partitioned into different groups. FRESA assigns different sampling rates to different groups.

FRESA uses a hashtable to maintain variables access frequency information. We define the memory location x in hashtable as a tuple (siteId, frequencyNum). $siteId$ refers to the call site at which x is allocated and $frequencyNum$ stands for the access frequency of x. We update the hashtable after every access using the algorithm below.

Algorithm 1 Hashtable Update Algorithm

```
1: if hashtable.get(siteId)!=NULL then
2:     frequencyNum()++
3: else
4:     hashtable.put(siteId,1)
5: end if
```

With the program execution, the hashtable updates at run time. Assume we get a hash map contains n different accesses x_1, x_2, x_3 ... x_n and with the relevant access frequencies f_1, f_2, f_3 ... f_n. In a general way, we first simply divide our n tuples into 5 groups.For an empirical practice, we set four group frequency threshold as $g_1 = 20$, $g_2 = 50$, g_3=100, $g_4 = 200$. For each (x_i,f_i), we dispatch it into a group in the following formula:

$$(x_i, f_i) \in \begin{cases} G_1 & f_i \leq g_1 \\ G_2 & g_1 < f_i \leq g_2 \\ G_3 & g_2 < f_i \leq g_3 \\ G_4 & g_3 < f_i \leq g_4 \\ G_5 & f_i > g_4 \end{cases} \tag{1}$$

We only list a simple piecewise function above. The group threshold can be different in practice due to the different frequency distribution.

3.2 Adaptive Sampling

FRESA does the same thing as PACER either in sampling periods or non-sampling periods. During sampling periods, FRESA fully tracks the happen-before relationship on all synchronization operations, and variable reads and writes, using FASTTRACK algorithm. In non-sampling periods, FRESA also reduces the space and time overheads of race detection by simplifying analysis on synchronization operations and variable reads and writes. However, FRESA adaptively changes its sampling rate using variable access frequency information. As for one access x, sampling rate r is decided by variable group classification as the following formula:

$$r_{x_i} = F\left(g_j | r_0\right) \;\; x_i \in g_j \tag{2}$$

where r_{x_i} indicates the proper sampling rate for access x_i. $F(g_j|r_0)$ defines the sampling rate for group g_j.

In our experiment, we use an exponential decline equation $r_{x_i} = r_0/2^j \; x_i \in g_j$ (j = 1, 2, 3, 4) in our example.

3.3 Theoretical Accuracy and Slowdown

Table 1 summarizes the effect of FASTTRACK, PACER and FRESA have on (1) the detection rate for any race and (2) program performance for sampling rate r and data race occurrence rate o.

Table 1. Theoretical accuracy and slowdown

	Det. race	**Slowdown**
FASTTRACK	o	$c_{\text{rw}}+c_{\text{sync}}n$
PACER	$o \times r$	$c_{\text{sampling}}(c_{\text{rw}} + c_{\text{sync}}n)r + c_{\text{nonsampling}}$
FRESA	$\sum_{i=1}^{m} o_i \times r_i \sum_{i=1}^{g} c_{\text{sampling}}(c_{\text{rw}}^{'} + c_{\text{sync}}n)r_i + c_{\text{nonsampling}}$	

Constant c_{rw} is the slowdown due to analysis at reads and writes, and $c_{\text{sync}}n$ is the linear slowdown in the number of threads n due to analysis at synchronization operations. PACER essentially scales FASTTRACK's overhead by r, as well as a small constant factor c_{sampling} due to PACER's additional complexity (e.g., indirect metadata lookups). PACER adds a slowdown $c_{\text{non-sampling}}$ during non-sampling periods, which is small and near-constant in practice. In FRESA, constant $c_{\text{rw}}^{'}$ is a little bigger than that in PACER as shared memory's access frequency information update.

4 Performance Evaluation

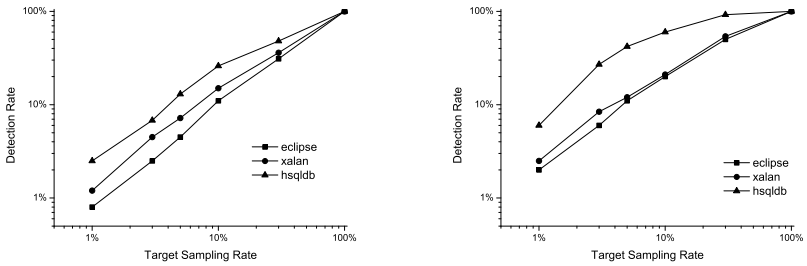
4.1 Implementation

FRESA is implemented in Jikes RVM 3.1.0[1], a high-performance Java-in-Java virtual machine [1]. FRESA is built on PACER's source code[2]. We execute all experiments on a Pentium Dual-Core CPU E5300 @2.6 GHz system with 2 GB main memory running openSUSE Linux 3.4.6-2.10. We used the multithreaded DaCapo benchmarks [2] (eclipse, hsqldb, and xalan; versions 2006-10-MR1). The range of sampling rates we use in our experiments is [0.000625, 1], and we set the minimum user given sampling rate $r_{min} = 0.01$. As data races occur infrequently and sampling decreases the probability of observing a race, we do many trials to evaluate accuracy.

[1] http://dacapo.anu.edu.au/regression/perf/2006-10-MR2.html
[2] http://www.jikesrvm.org/Research+Archive

4.2 Effectiveness of Data Race Detection

Figure 2(a) and 2(b) show FRESA's detection rate versus sampling rate for each benchmark. Figure 2(a) counts the average number of dynamic evaluation races per run that FRESA detects. A race's detection rate is the ratio of (1) average dynamic races per run at sampling rate r to (2) average dynamic races per run with $r = 100\%$. Each point is the average of all evaluation races' detection rates. The plot shows that FRESA reports races at a somewhat better rate than the sampling rate. However, this may have some different observations in different executions.



(a) FRESA's accuracy on dynamic races. (b) FRESA's accuracy on distinct races.

Fig. 2. FRESA's accuracy on dynamic and distinct races.

Figure 2(b) shows the detection rate for distinct races. If a static race occurs multiple times in one trial, this plot counts it only once. The detection rate is much higher because FRESA's main concept of memory access frequency-sensitive, which means infrequent accesses have more sampling cost.

4.3 Time and Space Overheads

Time overhead. Figures 3 shows the slowdown incurred by FRESA on each benchmark for r ranging over [0, 100%]. The graphs show that FRESA has overheads that scale roughly linearly with the target sampling rate, although the slowdown factors on different benchmarks vary a lot. The slowdown factor in this experiment includes the execution time of the program, the overhead incurred by the JikesRVM platform, the overhead incurred by dynamic memory access and synchronization instrumentation and the overhead incurred by the sampling algorithms. FRESA incurs slowdowns by a factor less than 3x on three benchmarks (*eclipse*, *hsqldb*, *xalan*) at a target sampling rate of 1%, and can detect races with a relatively higher probability (80%) than PACER. When working at a target sampling rate of 100%, FRESA has no sampling effort and is functionally equivalent to FASTTRACK[21]. In this scenario, FRESA slows

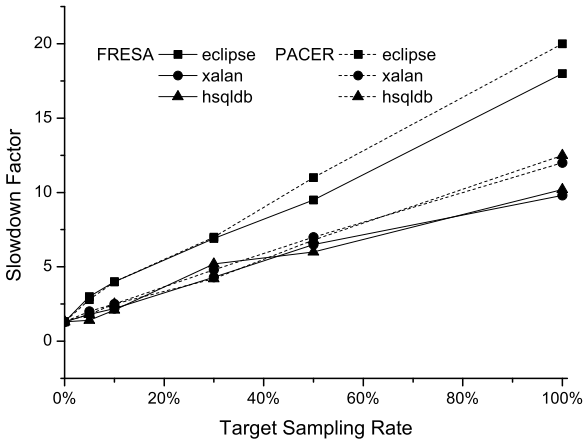


Fig. 3. Slowdown vs. sampling rates

down the three programs by a factor of 10x on average, compared with 8x by FASTTRACK and 12x by PACER. Though FRESA uses hashtable leading to little more access time, it still incurs less time overhead than PACER due to the saving of sampling cost.

Space overhead. Figure 4 shows the maximum live memory space overhead incurred by FRESA with various FRESA configurations. The measurement includes application, VM, PACER, and FRESA memory. For each target sampling rate shown, we take the mean overhead over all executions. Base shows the max memory used by eclipse running on unmodified Jikes RVM. OM only adds two words per object and a few percent all-the-time overhead. The other configurations (except LITERACE) are PACER and FRESA at various sampling rates. At a sampling rate of 100%, FRESA takes no differences as PACER. At other sampling rates, FRESA uses significantly less memory than PACER. The result shows that FRESA can scale well with the sampling rate in terms of memory space used, and with a low sampling rate of $r = 1\%$, its space overhead appears to be very low.

5 Related Work

A large part of researches [4–11, 16, 17, 19–22, 24–31] focus on dynamic or static race detection. Dynamic race detection techniques are either based on lockset or on happen-before or hybrid of them. Lockset based dynamic techniques could predict data races that does not happen in a concurrent execution which leads to report many false warnings. Happen-before based techniques detect races that

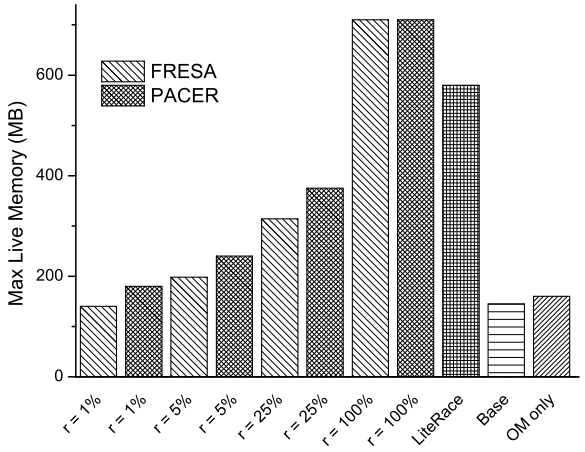


Fig. 4. Max live memory for eclipse

actually happen in an execution. Therefore, these techniques lack of good coverage though precise. On the other hand, happens-before based race detectors cannot predict races that could happen on a different schedule or they cannot create a schedule that could reveal a real race. In practice, the coverage can be increased by running more tests. Recently happens-before race detection has been successfully extended to classify harmful races with sampling. Hybrid techniques combine lockset with happens-before to make dynamic race detection both precise and predictive. But these techniques also report many false warnings. Static race detection techniques provide maximum coverage by reasoning about data races on all execution paths. However, they tend to make conservative assumptions that lead to a large number of false data races.

In general, the traditional methods using vector clocks takes $O(n)$ time and space overhead, n is the number of threads. The FASTTRACK algorithm replaces most $O(n)$ analysis with $O(1)$ analysis with precise guaranteed. However, these methods still incurs significant overhead.

A novel approach that explores the above tradeoff is sampling. LITERACE uses a sampling dispatch to decide whether to sample synchronization events only or together with all memory access of a function. On the other hand, PACER uses a more effective sampling strategy. It divides an execution into a scenario of sampling or non-sampling periods. PACER can detect race in a magic shortest data race way and with a probability of r. This significantly reduces the slowdown overheads.

A common limitation of the above techniques is that although existing precise sampling-based data race detectors such as PACER and LITERACE can effectively reduce overheads so that lightweight precise race detection can be

performed efficiently in testing or post-deployment environments, they are ineffective in detecting races with much unnecessary repeated sampling cost. Our insight is that along an execution trace, a program may sample some race-infrequently variable in high sampling rate. These unnecessary sampling cost potentially indicate a saving cost degree in a sampling region. Intuitively, they may perform redundant memory access sampling, which lowers the chance of detecting rare data races and costs more unnecessary sampling overhead.

Recently, a couple of random testing techniques for concurrent programs have been proposed. These techniques randomly seed a program under test at shared memory accesses and synchronization events. Although these techniques have successfully detected bugs in many programs, they have two limitations. These techniques are not systematic or reproducible. Many researchers look for effective sampling method to solve this problem.

6 Conclusions and Future Work

Data race is a common problem in multithreaded program. It often indicates serious concurrency errors which are easy to introduce but difficult to reproduce, discover, and fix. Prior approaches reduce overhead by sampling. But they waste too many cost on repeated checking which can be omitted or reduced. In other words, frequency statistical sampling strategy could be more effective with detection precise guaranteed. This paper presents a data race detection method that provides a detection rate for each memory location that has inverse relationship with access frequency, and adds less time and a little more space overheads than PACER. FRESA achieves a qualitative improvement over prior work: its access frequency based sampling strategy suits more comfortable for performance and accuracy sensitive program, which makes it more suitable for all-the-time use in a variety of deployed environments. Our future work should simplify happen-before relationship and optimize the instrumentation with a lower cost level. In addition, we could generalize the approach to deal with different types of bugs caused by frequently access and develop new methods to further refine the approach.

7 Acknowledgements

Thanks to the anonymous reviewers for feedback on this work. This work is supported in part by the National High-tech R&D Program of China (863 Program) under grant No.2012AA010905.

References

1. Alpern, B., Attanasio, C.R., Cocchi, A., Lieber, D., Smith, S., Ngo, T., Barton, J.J., Hummel, S.F., Sheperd, J.C., Mergen, M.: Implementing jalapeño in java. In: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. pp. 314–324. ACM (1999)

2. Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A., Jump, M., Lee, H., Moss, J.E.B., Phansalkar, A., Stefanović, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: The dacapo benchmarks: Java benchmarking development and analysis. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. pp. 169–190. ACM (2006)
3. Bond, M.D., Coons, K.E., McKinley, K.S.: Pacer: proportional detection of data races. In: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation. pp. 255–268. ACM (2010)
4. Boyapati, C., Lee, R., Rinard, M.: Ownership types for safe programming: Preventing data races and deadlocks. SIGPLAN Not. 37(11), 211–230 (2002)
5. Christiaens, M., De Bosschere, K.: Trade, a topological approach to on-the-fly race detection in java programs. In: Proceedings of the 2001 Symposium on Java TM Virtual Machine Research and Technology Symposium. vol. 1, pp. 15–15. USENIX Association (2001)
6. Dinning, A., Schonberg, E.: Detecting access anomalies in programs with critical sections. In: Proceedings of the 1991 ACM/ONR workshop on Parallel and distributed debugging. pp. 85–96. ACM (1991)
7. Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware java runtime. In: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation. pp. 245–255. ACM (2007)
8. Engler, D., Ashcraft, K.: Racerx: effective, static detection of race conditions and deadlocks. SIGOPS Oper. Syst. Rev. 37(5), 237–252 (2003)
9. Flanagan, C., Freund, S.N.: Type-based race detection for java. ACM SIGPLAN Notices 35(5), 219–232 (2000)
10. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 121–133. ACM (2009)
11. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. pp. 1–13. ACM (2004)
12. Lu, S., Park, S., Seo, E., Zhou, Y.: Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems. pp. 329–339. ACM (2008)
13. Lu, S., Tucek, J., Qin, F., Zhou, Y.: Avio: detecting atomicity violations via access interleaving invariants. In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. pp. 37–48. ACM (2006)
14. Manson, J., Pugh, W., Adve, S.V.: The java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 378–391. ACM (2005)
15. Marino, D., Musuvathi, M., Narayanasamy, S.: Literace: effective sampling for lightweight data-race detection. In: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation. pp. 134–143. ACM (2009)
16. Min, S.L., Choi, J.D.: An efficient cache-based access anomaly detection scheme. In: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems. pp. 235–244. ACM (1991)

17. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proceedings of the twentieth ACM symposium on Operating systems principles. pp. 308–319. ACM (2006)
18. Netzer, R.H., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Lett. Program. Lang. Syst. 1, 74–88 (1992)
19. O'Callahan, R., Choi, J.D.: Hybrid dynamic data race detection. ACM Lett. Program. Lang. Syst. 38(10), 167–178 (2003)
20. Pozniansky, E., Schuster, A.: Efficient on-the-fly data race detection in multithreaded c++ programs. In: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 179–190. ACM (2003)
21. Pratikakis, P., Foster, J.S., Hicks, M.: Locksmith: context-sensitive correlation analysis for race detection. In: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation. pp. 320–331. ACM (2006)
22. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation. pp. 14–24. ACM (2004)
23. Sack, P., Bliss, B.E., Ma, Z., Petersen, P., Torrellas, J.: Accurate and efficient filtering for the intel thread checker race detector. In: Proceedings of the 1st workshop on Architectural and system support for improving software dependability. pp. 34–41. ACM (2006)
24. Sasturkar, A., Agarwal, R., Wang, L., Stoller, S.D.: Automated type-based analysis of data races and atomicity. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 83–94. ACM (2005)
25. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multithreaded programs. ACM Trans. Comput. Syst. 15(4), 391–411 (1997)
26. Schonberg, E.: On-the-fly detection of access anomalies. SIGPLAN Not. 39(4) (1989)
27. Sterling, N.: Warlock: A static data race analysis tool. In: USENIX Winter. pp. 97–106 (1993)
28. Von Praun, C., Gross, T.R.: Object race detection. In: Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. vol. 36, pp. 70–82. ACM (2001)
29. Voung, J.W., Jhala, R., Lerner, S.: Relay: static race detection on millions of lines of code. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 205–214. ACM (2007)
30. Yu, Y., Rodeheffer, T., Chen, W.: Racetrack: efficient detection of data race conditions via adaptive tracking. In: Proceedings of the twentieth ACM symposium on Operating systems principles. pp. 221–234. ACM (2005)
31. Zhai, K., Xu, B., Chan, W., Tse, T.: Carisma: a context-sensitive approach to race-condition sample-instance selection for multithreaded applications. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. pp. 221–231. ACM (2012)