BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Excel@FIT 2023
http://excel.fit.vutbr.cz

Static Data Race Detection in Low-Level C Code

Lucie Svobodová*

Abstract
In this paper, we present DarC, a novel static data race analyser designed for low-level C code. Our tool is implemented as an analyser plugin of the Facebook/Meta INFER framework.

*xsvobo1x@stud.fit.vutbr.cz, *Faculty of Information Technology, Brno University of Technology*

1. Introduction

Nowadays, *multithreading* has become a popular approach to utilize the many processors of modern computers better. However, concurrency leads not only to faster execution, but concurrent programs are also more complex and harder to understand, test and debug due to the many ways in which concurrently running threads can interleave, with errors hidden in just a few of them. Even after running tests on a concurrent program multiple times, it may still be challenging to uncover all the possible *interleavings* of the threads that could lead to errors.

To improve the coverage of rare behaviors in concurrent programs, different approaches such as *systematic testing* [1] and *noise-based testing* [2] can be used. Another effective approach is to use *extrapolating dynamic checkers* like [3], which can identify potential errors based on the detection of their symptoms, even if they are not encountered during testing runs. Unfortunately, even though these checkers have proven quite useful in practice, they can still miss errors. Furthermore, monitoring a run of a large program through such checkers may be quite expensive and time-consuming.

On the other hand, approaches based on *model checking* can ensure the detection of all potential errors. However, the scalability of these techniques is still limited and so far cannot handle truly large industrial code. Even with the help of methods such as *sequentialization* [4], which is one of the most scalable approaches in the field, it is still challenging to analyse such codebases.

Static analysis is an alternative to the above approaches. It can scale better than model checking and find bugs not found by dynamic analysis, though for the price of potentially missing some errors and/or producing false alarms. The area of static analysis is very extensive and includes many different approaches, such as *data-flow analysis* and *abstract interpretation*. The latter approach is supported, e.g., in *Facebook/Meta Infer* [5] — an open-source framework for creating highly scalable, compositional, incremental, and interprocedural static analysers based on abstract interpretation. Infer provides several analysers that check for various types of bugs, such as buffer overflows, null-dereferencing, or memory leaks. As for *concurrency-related bugs*, Infer provides a support for finding some forms of *data races* and *deadlocks*, but it is limited to Java programs only and fails for C programs, which use a lower-level lock manipulation [6, 7]. In this work, we propose DarC, a *data race checker designed for low-level C code* that fits the common principles of analyses used in Infer.

The design of the DarC checker was inspired by *RacerD* [6], a data race detector for Java programs, that is already implemented in Infer. Both checkers use a bottom-up approach based on computing function summaries, but RacerD is limited to programs that use high-level locking and class constructs. Existing solutions for data race detection in C code include the *RacerX* [8] and *Coderrect/O2* [9] static analysers. The former analyser uses a top-down approach, is flow-sensitive and context-sensitive, and uses various heuristics, such as a ranking algorithm, to reduce the number of false alarms emitted. Unfortunately, annotations must be added to the code before using this tool. Coderrect/O2 is an analyser for both C/C++ and Java/Android applications, and it is powered by origins, an abstraction of threads and events.

2. Static Data Race Detection

For scalability reasons, DarC runs *along the call tree* of a given program in a *bottom-up manner*, starting from its leaves. Each function is analysed *only once*, without knowing its possible call context. As the function is analysed, a *summary* is derived and used when analysing functions higher up in the call tree.

The content of the *summaries* that we have proposed in our work consists of a *set of accesses* that occur in the analysed function, a *set of locks* that remain locked at the end of the function, a *set of threads* that may be running, and *aliases*, a set of pairs of variables that may alias. Out of these elements, the *set of accesses* is probably the most important for the data races computation.

Each *access* contains information about the *accessed variable*, *type* of the access (i.e. read or write), a set of *locks that must be locked* at the given program point, a set of *threads that may be currently running*, and the *thread* on which the variable is accessed. However, due to the bottom-up approach, the information about threads and locks is incomplete and is updated later, when the function is called from higher-level functions. Listing 2 shows the set of accesses computed for the main function in Listing 1.

```
1  int i;
2  void *foo() {
3    pthread_mutex_lock(lock);
4    i = 0;
5    pthread_mutex_unlock(lock);
6  }
7  int main() {
8    pthread_create(thread1, foo);
9    i = 42;
10 }
```

Listing 1. A sample code illustrating a data race between two accesses using C/Pthreads.

```
accesses: {
  (i, line 4, Write, {main, thread1}, {lock}, thread1),
  (i, line 9, Write, {main, thread1}, {}, main)}
```

Listing 2. A set of accesses in the summary of the main function from Listing 1.

During analysing a function, the set of *variables that may alias* is stored. When a variable is accessed and any alias of the variable is found, accesses to all variables that may be aliased are added to the set of accesses. A set of variables that are local to the function is also stored during analysis, and only those accesses that are not to local variables are stored in the function's summary, reducing the size of the set of accesses and increasing the scalability.

The *computation of data races* takes place after the analysis of the main function is completed. Using a system of conditions, it is checked if there exists a *pair of accesses* in the set, where *multiple threads may access the same variable without proper synchronization*. If a pair of accesses that may cause a data race is found, it is reported. Only one data race for a variable is reported to avoid overwhelming developers during report examination.

We have evaluated our checker on both simple and more complex real-world programs. In our first set of experiments, we have applied *DarC* on two benchmarks and compared the analysis results with results produced by *Coderrect/O2* [9], as well as two dynamic analysers, *ThreadSanitizer* [10] and *Helgrind* [11]. The results of the analysis on the *DataRaceBenchmark* [12], a benchmark consisting of 67 C/Pthreads programs, which was designed to evaluate concurrency analysers, are shown in Table 1. DarC reported nearly the same number of data races as the dynamic analysers, while Coderrect missed at least 11 of them. We also analysed a benchmark of 82 programs that we developed ourselves as a test suite for DarC [13]. For this benchmark, DarC reported 55 programs with data races, where 45 of them were confirmed by ThreadSanitizer. Coderrect detected only 24 of them and the analysis time for Coderrect was more than three times longer than for DarC.

Table 1. Results of DarC, Coderrect, ThreadSanitizer and Helgrind analysers on DataRaceBenchmark [12].

analyser	races	no races	errors	time
DarC	41	26	0	26.3s
Coderrect	29	33	1	1m20s
ThreadSanitizer	40	23	0	17.7s
Helgrind	40	22	1	28.5s

Regarding experiments with more complex programs, we have analysed the *sort*, *grep*, *tgrep*, *memcached*, and *Fast-DDS* applications. DarC did not report any data races in any of these programs. However, during our investigation of *tgrep*, we discovered that `pthread_create()` wrappers are used in the program, which our checker does not currently support. This may be a reason why no data races were reported. As an experiment, we replaced the wrappers with calls to the `pthread_create()` function, which led to the detection of two data races. However, we do not have the information if these are false alarms or not. As a sanity check, we removed one `pthread_mutex_lock()` function call in *tgrep* and DarC successfully detected the introduced data race.

We find the results of our experimental evaluation quite promising, and the improvement of our new data race analyser based on the identified problems will be the subject of our future work.

Acknowledgements

I would like to thank my supervisor Tomáš Vojnar and Tomáš Dacík for their help. The work was supported by the Czech Science Foundation project AIDE (23-06506S) and the Horizon Europe project CHESS (101087529).

References

[1] J. Wu, Y. Tang, H. Hu, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proc. of PLDI'12*. ACM, 2012.

[2] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-based Testing. *Software Testing, Verification and Reliability*, 24(7):1–38, 2014.

[3] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of PLDI'09*. ACM, 2009.

[4] T. L. Nguyen, B. Fischer, S. L. Torre, and G. Parlato. Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs. In *Proc. of ATVA'16*, volume 9938 of *LNCS*. Springer, 2016.

[5] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving Fast with Software Verification. In *Proc. of NFM'15*, volume 9058 of *LNCS*. Springer, 2015.

[6] S. Blackshear, N. Gorogiannis, P. O'Hearn, and I. Sergey. RacerD: Compositional Static Race Detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):144:1–144:28, 2018.

[7] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling Static Analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.

[8] D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc. of SOSP'03*. ACM, 2003.

[9] Bozhen Liu, Peiming Liu, Yanze Li, Chia-Che Tsai, Dilma Da Silva, and Jeff Huang. When threads meet events: Efficient and precise static race detection with origins. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2021, page 725–739, New York, NY, USA, 2021. Association for Computing Machinery.

[10] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, page 62–71, New York, NY, USA, 2009. Association for Computing Machinery.

[11] Home of memcheck, helgrind and drd.

[12] DataRaceBenchmark. https://github.com/marchartung/DataRaceBenchmark.

[13] ConcurrencyBenchmark. https://github.com/svobodovaLucie/ConcurrencyBenchmark.