Static Deadlock Detection in Low-Level C Code*

Dominik Harmim, Vladimír Marcin, Lucie Svobodová, and Tomáš Vojnar

Faculty of Information Technology, Brno University of Technology
Brno, Czech Republic
{iharmim,vojnar}@fit.vut.cz

Abstract. We present a novel *scalable deadlock analyser* (L2D2) that works *compositionally* and applies to *C code* with *low-level unstructured lock manipulation*. We proposed and implemented L2D2 as a plugin of the Facebook/Meta INFER framework. L2D2 runs *along the call tree* of a given program, *starting from its leaves*. Each function is thus analysed just once without any knowledge of its possible call contexts. For each analysed function, we derive a summary that contains *lock dependencies*. In the end, we compute the transitive closure of these dependencies, and we consider found cycles as deadlocks. Our experimental evaluation on a set of C programs with POSIX threads derived from a Debian GNU/Linux distribution with in total 11.4 MLoC shows that our approach is effective in practice, finding all known deadlocks and giving false alarms in less than 4 % of the considered programs only.

1 Introduction

Nowadays, programs often use *multithreading* to utilise the many processors of current computers better. However, concurrency does bring not only speed-ups but also a much larger space for nasty errors easy to cause but difficult to find. The reason why finding errors in concurrent programs is particularly hard is that concurrently running threads may *interleave* in many different ways, with bugs hiding in just a few of them. Such interleavings need not be discovered even by many-times repeated testing.

Coverage of such rare behaviours can be improved using approaches such as *systematic testing* [22] and *noise-based testing* [8, 10, 11]. Another way is to use *extrapolating dynamic checkers*, such as [12, 13], which can report warnings about possible errors even if those are not seen in the testing runs, based on spotting some of their symptoms. Unfortunately, even though such checkers have proven quite useful in practice, they can, of course, still miss errors. Moreover, monitoring a run of large software through such checkers may also be quite expensive.

On the other hand, approaches based on *model checking*, i.e., exhaustive state-space exploration, can guarantee the discovery of all potentially present errors — either in general or at least up to some bound, which is usually given in the number of context switches. However, so far, the scalability of these techniques is not sufficient to handle truly large industrial code, even when combined with methods such as *sequentialisation* [18, 20], which represents one of the most scalable approaches in the area.

* The work was supported by the Czech Science Foundation (the project No. 20-07487S) and the Brno Ph.D. Talent Scholarship Programme.

An alternative to the above approaches, which can scale better than model checking and can find bugs not found dynamically (though for the price of potentially missing some errors and/or producing false alarms), is offered by approaches based on *static analysis*, e.g., in the form of *abstract interpretation* [6] or *data-flow analysis* [16]. The former approach is supported, e.g., in Facebook/Meta INFER — an open-source framework for creating highly scalable, compositional, incremental, and interprocedural static analysers based on abstract interpretation [4]. INFER has initially been a rather specialised tool focused on *sound verification* of the absence of memory safety violations, which was first published in the well-known paper [5]. Once Facebook/Meta has purchased it, its scope significantly widened and abandoned the focus on sound analysis only. INFER has grown considerably, but it is still under active development. It is employed every day not only in Facebook/Meta itself but also in other companies, such as Amazon, Microsoft, Mozilla, Spotify, or Uber. Currently, INFER provides several analysers that check for various types of bugs, such as buffer overflows, null-dereferencing, or memory leaks. However, most importantly, INFER is a *framework* for building new analysers quickly and easily. As for *concurrency-related bugs*, INFER provides support for finding some forms of *data races* and *deadlocks*, but it is limited to *high-level* Java and C++ programs only and fails for C programs, which use a *lower-level lock manipulation* [1, 7].

In this paper, we propose a *deadlock checker* that fits the common principles of analyses used in INFER and is applicable to *C code* with *lower-level lock manipulation*. Our checker is called L2D2 for "low-level deadlock detector".

Structure of the Paper The rest of this paper is organised as follows. Section 1.1 discusses related work. A design of the L2D2 analyser is introduced in Section 2. The paper then goes on in Section 3 to outline several considered heuristics that increase accuracy of the analysis. This is followed by Section 4, which covers the experimental evaluation of the implemented approach. Finally, the paper is concluded and further research directions are provided in Section 5.

1.1 Related Work

Nowadays, there is a number of tools for deadlock detection in multi-threaded programs. A common deficiency of a large number of them is that they are unsound and/or incomplete, or they are precise, but their requirements on time and resources are unacceptable when applied to large codebases.

To our knowledge, L2D2 is the only available *compositional static deadlock analyser* for *low-level code*. Nevertheless, in this section, we briefly compare our contribution to related work, first, on dynamic tools, and second, on static tools.

Dynamic Analyses Dynamic analysers work with *program traces*. They require the whole program as well as appropriate *test input data*, and their scalability is often very limited. These techniques tend to focus on completeness (however, this does not need to hold for *extrapolation based approaches* such as the below-mentioned GOODLOCK) but cannot be sound.

GOODLOCK [14] is a well-known dynamic analysis for Java programs implemented in Java PathFinder (JPF) [15]. This approach uses *deadlock prediction* to detect a deadlock. It makes predictions about an exponential number of permutations of single execution history. Essentially, it monitors the lock acquisition history by creating a *dynamic lock-order graph*, followed by checking the graph for the existence of deadlock candidates by searching for cycles in it. A drawback of this approach is that it may produce a high rate of false positives.

AIRLOCK [3] is one of the state-of-the-art dynamic deadlock analysers. It adopts and improves the basic approach from GOODLOCK by applying various optimisations to the extracted lock-order graph. Moreover, AIRLOCK operating on-the-fly, running a polynomial-time algorithm on the lock graph to eliminate parts without cycles before running the higher-cost algorithm to detect actual lock cycles.

Static Analyses Static deadlock detectors can often handle much larger systems and may produce sound results. However, static techniques are generally incomplete, so soundness is sometimes dropped to minimise the number of false alarms. Most of the existing deadlock analysers are interprocedural, top-down, context-sensitive, and non-compositional. Although, this is not the case of L2D2 and the below-mentioned STARVATION checker, which are both *bottom-up*, *context-insensitive*, and *compositional*. Below are listed three analyses in some aspects close to L2D2; indeed, some of the approaches used in L2D2 are inspired by them.

RACERX [9] is a flow-sensitive (based on *data-flow analysis*) and context-sensitive analysis for C programs building a *static lock-order graph* by computing so-called *lock-sets*, i.e., sets of currently held locks, and reporting possible deadlocks in case of cycles in it. It does not use a pointer analysis, and many heuristics are employed to reduce false-positive reports. This approach, however, still produces many false alarms due to the used approximations, and its context sensitivity reduces the scalability.

The deadlock analyser implemented within the CPROVER framework [17] targeting C code with POSIX threads uses a combination of multiple analyses to create a sound analysis. The analysis is context-sensitive and is based on *abstract interpretation*. It also builds a lock-order graph and searches for cycles to detect deadlocks. The most significant limitation of this approach is the pointer analysis used, which takes most of the analysis time. An experimental comparison with this tool is given in Section 4.

STARVATION [2] is one of the most successful state-of-the-art deadlock analysers. It is implemented in the INFER framework; therefore, it is bottom-up, context-insensitive, compositional, and abstract interpretation based. As a matter of fact, to our best knowledge, it is the only existing deadlock detector that works compositionally. It detects deadlocks by deriving lock dependencies for each function and checking whether some other function uses the locks in inverse order. It is thus similar to L2D2, but STARVATION is limited to *high-level* Java and C++ programs with *balanced locks* only. Moreover, it implements many heuristics explicitly tailored for Android Java applications.

2 Static Deadlock Detection in Low-Level Concurrent C Code

This section presents the design of the L2D2 analyser. First, we sketch out the basic ideas of the analysis. Further, we introduce the algorithm of the analyser in more detail. In the end, we show an algorithm for reporting deadlocks.

As already mentioned, L2D2 is designed to work *compositionally*[1] and be applied to *C code* with *low-level unstructured lock manipulation*. For scalability reasons, L2D2 does not run the analysis along the *Control-Flow Graph (CFG)* as it is done in classical analyses based on the concepts, e.g., from [21]. Instead, L2D2 performs the analysis of a program *function-by-function along the call tree*, *starting from its leaves* (as typical for the INFER framework). Therefore, each function is analysed just once without any knowledge of its possible call contexts. For each analysed function, we derive a *summary* that consists of a *pre-condition* and *post-condition*. The summaries are then used when analysing functions higher up in the call hierarchy. More details on how the summaries look and how they are computed will be given in Section 2.1.

We use *syntactic access paths* [19] to represent lock objects. This mechanism is already built in the INFER framework. The access paths represent heap locations via the paths used to access them. Then, L2D2 does not have to perform a classical *alias analysis*, i.e., a precise analysis for saying whether arbitrary pairs of accesses to lock objects may alias (such an analysis is considered too expensive — no such sufficiently precise analysis works compositionally and at scale). According to the authors of [1], the access paths' syntactic equality is a reasonably efficient way to say (in an under-approximate fashion) that heap accesses touch the same address. They used this mechanism in RACERD [1] to detect many data races in real-world programs.

2.1 Computing Function Summaries

[TODO: . . .]

2.2 Reporting Deadlocks

[TODO: . . .]

3 Increasing Analysis Accuracy

[TODO: . . .]

4 Experimental Evaluation

[TODO: . . .] [17]

[1] *Compositionality* allows us to concentrate only on *modified files* and their dependants. A summary of a function call depends only on the current state of the caller, which can be computed in advance. This means that — when analysing a code revision — we do not need to re-analyse the unchanged functions in the program (i.e. the vast majority).

5 Conclusions and Future Work

[TODO: . . .]

References

1. S. Blackshear, N. Gorogiannis, P. O'Hearn, and I. Sergey. RacerD: Compositional Static Race Detection. *Proc. of ACMPL*, 2(OOPSLA):144:1–144:28, 2018.
2. J. Brotherston, P. Brunet, N. Gorogiannis, and M. Kanovich. A Compositional Deadlock Detector for Android Java. In *Proc. of ASE'21*. IEEE, 2021.
3. Y. Cai, R. Meng, and J. Palsberg. Low-Overhead Deadlock Prediction. In *Proc. of ICSE'20*. ACM, 2020.
4. 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.
5. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. In *Proc. of POPL'09*. ACM, 2009.
6. P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approx. of Fixpoints. In *Proc. of POPL'77*. ACM, 1977.
7. D. Distefano, M. Fähndrich, F. Logozzo, and P. O'Hearn. Scaling Static Analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.
8. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-Threaded Java Programs. *Concur. Computat.: Pract. Exper.*, 15(3–5):485–499, 2003.
9. D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc. of SOSP'03*. ACM, 2003.
10. J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-Based Testing of Concurrent Software. *Softw. Test. Verif. Reliab.*, 25(3):272–309, 2015.
11. J. Fiedor, M. Mužikovská, A. Smrčka, O. Vašíček, and T. Vojnar. Advances in the ANaConDA Framework for Dynamic Analysis and Testing of Concurrent C/C++ Programs. In *Proc. of ISSTA'18*. ACM, 2018.
12. C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of PLDI'09*. ACM, 2009.
13. C. Flanagan, S. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. of PLDI'08*. ACM, 2008.
14. K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *SPIN Model Checking and Software Verification*, volume 1885 of *LNCS*. Springer, 2000.
15. K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *Inter. Jour. on STTT*, 2(4):366–381, 2000.
16. G. Kildall. A Unified Approach to Global Program Optimization. In *Proc. of POPL'73*. ACM, 1973.
17. D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound Static Deadlock Analysis for C/Pthreads. In *Proc. of ASE'16*. ACM, 2016.
18. A. Lal and T. Reps. Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis. In *Proc. of CAV'08*. Springer, 2008.
19. J. Lerch, J. Späth, E. Bodden, and M. Mezini. Access-Path Abstraction: Scaling Field-Sensitive Data-Flow Analysis with Unbound. Access Paths. In *Proc. of ASE'15*. IEEE, 2015.
20. T. Nguyen, B. Fischer, S. 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.

21. T. Reps, S. Horwitz, and M. Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. of POPL'95*. ACM, 1995.
22. 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.