Static Deadlock Detection in Low-Level C Code

Dominik Harmim, Vladimír Marcin, Lucie Svobodová, and Tomáš Vojnar
FIT, Brno University of Technology, Czech Republic

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 the 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 by approaches such as *systematic testing* [9] and *noise-based testing* [5]. Another way is using *extrapolating dynamic checkers*, such as [6], 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 be quite expensive too.

On the other hand, approaches based on *model checking*, i.e., exhaustive state-space exploration, can guarantee the discovery of all potentially present errors. However, the scalability of these techniques is so far not sufficient to handle truly large industrial code, even when combining it with methods such as *sequentialisation* [8], which represents one of the most scalable approaches in the area.

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* or *data-flow analysis*. The former approach is supported, e.g., in *Facebook Infer* — an open-source framework for creating highly scalable, compositional, incremental, and interprocedural static analysers based on abstract interpretation [2]. Currently, 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 support for finding some forms of *data races* and *deadlocks*, but it is limited to high-level Java and C++ programs [1, 3]. In this work, 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".

Out of existing static deadlock detectors, L2D2 is probably the closest to RacerX [4], and some of the approaches used in L2D2 are inspired by RacerX. However, unlike L2D2, RacerX uses a context-sensitive analysis, meaning that each function needs to be re-analysed in a new context, which reduces the scalability.

2 Static Deadlock Detection in Low-Level Concurrent C Code

For scalability reasons, L2D2 runs *along the call tree* of a given program, *starting from its leaves*. 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. Intuitively, the pre-condition of a function states which locks are *expected to be locked or unlocked* upon a call of the function to avoid possible double-locking or unlocking, resp. The post-condition contains information about which locks *may be locked or unlocked* at the exit of the function. Next, the summary contains so-called *lock dependencies* in the form of pairs of locks (l_1, l_2) where locking of l_2 was seen while l_1 was locked. At the end of the analysis, detecting cycles in the lock dependencies is used to detect possible deadlocks. Finally, the summary also contains information on which *locks may be locked and then again unlocked* within the given function, which is needed for detecting lock dependencies with such locks in functions higher up in the call hierarchy.

L2D2 further implements several heuristics intended to decrease the number of possible false alarms. For instance, since double-locking/unlocking errors are quite rare, their detection may instead be used as an indicator that the analysis is over-approximating too much, and it may reset (some of) the working sets. An example of another heuristic used in L2D2 is the detection of so-called *gate locks*, i.e., locks guarding other locks (upon which deadlocks on the nested locks are not reported).

Within our experimental evaluation of L2D2, we have applied it on a set of 1,002 C programs with POSIX threads derived from a Debian GNU/Linux distribution, originally prepared for evaluating the static deadlock analyser based on the CProver framework proposed in [7]. The benchmark consists of 11.4 MLOC. Eight of the programs contain a known deadlock. Like CProver, L2D2 was able to detect all the deadlocks. On the remaining 994 programs, it produced 39 false alarms (78 of the programs failed to compile since the Infer's front-end does not support some of the constructions used). We find this very encouraging, considering that the CProver's deadlock detector produced 114 false alarms, 453 timeouts (w.r.t. our 30-minute time limit), and ran out of the available 24 GB of memory in 135 cases. Altogether, L2D2 consumed less than 1 % of the time taken by CProver.

Acknowledgement. The work was supported by the project No. 20-07487S of the Czech Science Foundation.

References

1. S. Blackshear, N. Gorogiannis, P. O'Hearn, and I. Sergey. RacerD: Compositional Static Race Detection. *Proc. of the ACM on Programming Languages*, 2(OOPSLA):144:1–144:28, 2018.
2. 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*, LNCS 9058. Springer, 2015.
3. D. Distefano, M. Fähndrich, F. Logozzo, and P. O'Hearn. Scaling Static Analyses at Facebook. *Commun. ACM*, 62(8):62–70, 2019.
4. D. Engler and K. Ashcraft. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proc. of SOSP'03*. ACM, 2003.
5. 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.
6. C. Flanagan and S. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of PLDI'09*. ACM, 2009.
7. D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound Static Deadlock Analysis for C/Pthreads. In *Proc. of ASE'16*. ACM, 2016.
8. T. Nguyen, B. Fischer, S. L. Torre, G. Parlato. Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs. *Proc. of ATVA'16*, LNCS 9938. Springer, 2016.
9. 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.