GOAT

Saeed Taheri and Ganesh Gopalakrishnan

University of Utah, Salt Lake City UT 84112, USA,
{staheri,ganesh}@cs.utah.edu,

Abstract. **Keywords:** golang,concurrency,testing,debugging

1 Introduction

2 Background

3 GOAT Design

ss

ppt/slideLayouts/slideLayout4
asdasdasd

4 Benchmark

GoBench

Results

Discussion

5 Related Work

Go comes with a few dynamic tools for analysis and debugging its programs. For example, the *race detector* [16] which is basically a wrapper around Thread-Sanitizer [11], tracks memory accesses and detect races that happened during execution. A few other facilities for code coverage measurement, profiling, and tracing are provided to deliver insight into the testing quality and performance behavior. However, there is still a considerable gap for debugging concurrency. Several research groups have recently proposed and developed a range of *static* (source-level) or *dynamic* (execution-level) theories and tools towards filling this gap. Ng and Yoshida [10] first proposed a static tool to detect global deadlock in Go programs using choreography synthesis. Later, Stadtmuller et al. [12] proposed a static trace-based global detection approach based on forkable regular expressions. Lange et al. proposed more static verification frameworks for checking channel safety, and liveness [7], and behavioral model checking [8]. Both methods approximate Go programs with session types and behavioral contracts extracted from their SSA intermediate representation. The mentioned work has limitations for handling dynamic (e.g., in-loop) goroutine or channel creation. They also do not scale and are impractical in real-world programs due to the state explosion problem. Besides, similar to other static analysis methods, they often suffer from false positives due to conservative constraints.

Zhao et al. [18] introduced a runtime monitoring approach for deadlock detection for Occam based on wait-for graphs and some heuristics. Occam is a concurrent language based on CSP semantics, and similar to Go, it uses channels to establish communication between processes. Sulzmann and Stadtmuller proposed a dynamic verification approach for synchronous (unbuffered) channels [13], and a vector-clock-based approach for asynchronous channels [14]. Both approaches require heavy code instrumentation and replay of collected traces. Although they may support a larger subset of the Go language, they only focus on channels as the root cause of deadlocks and evaluated only on relatively small examples. Generally, dynamic analyzers are not *sound* meaning that they are only able to catch occurred bugs and might miss potential unhappened bugs.

Systematic testing combines ideas from static and dynamic approaches to reduce the state space and reflect realistic behavior. Assuming the scheduler causes concurrency bugs (and not the program input), they may not manifest during conventional testing and difficult to reproduce, both due to non-deterministic decisions that the scheduler makes. Stress testing the scheduler to examine possible interleaving is useful to expose hidden bugs, but they are exponentially expensive relative to the number of concurrent units. Researchers have applied different methods [15] to reduce the interleaving space to explore effectively and efficiently. Delay-bounded [1,4] and preemption-bounded [9] techniques systematically "fuzz" the scheduler to equally and fairly cover feasible interleaving. Other tools like Maple [17], CalFuzzer [6], and ConTest [2,3] *actively* control the scheduler to maximise a pre-defined concurrency coverage criterion [5] or the probability of bug exposure [1].

Adopting ideas from existing concurrency testing tools, we systematically navigate the scheduler towards executing likely-erroneous interleaving. We first identify the usage of concurrency primitives (*critical points*) in the program using a tracing mechanism. We automatically inject random delays around the critical points to increase the probability of bug exposure (more in section 3).

References

1. Burckhardt, S., Kothari, P., Musuvathi, M., Nagarakatte, S.: A randomized scheduler with probabilistic guarantees of finding bugs. In: Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems. p. 167–178. ASPLOS XV, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/1736020.1736040, https://doi.org/10.1145/1736020.1736040

2. Edelstein, O., Farchi, E., Goldin, E., Nir-Buchbinder, Y., Ratsaby, G., Ur, S.: Framework for testing multithreaded java programs. Concurrency and Computation: Practice and Experience **15** (2003)
3. Edelstein, O., Farchi, E., Nir, Y., Ratsaby, G., Ur, S.: Multithreaded java program test generation. In: Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande. p. 181. JGI '01, Association for Computing Machinery, New York, NY, USA (2001). https://doi.org/10.1145/376656.376848, https://doi.org/10.1145/376656.376848
4. Emmi, M., Qadeer, S., Rakamarić, Z.: Delay-bounded scheduling. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 411–422. POPL '11, Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926432, https://doi.org/10.1145/1926385.1926432
5. Hong, S., Ahn, J., Park, S., Kim, M., Harrold, M.J.: Testing concurrent programs to achieve high synchronization coverage. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. p. 210–220. ISSTA 2012, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2338965.2336779, https://doi.org/10.1145/2338965.2336779
6. Joshi, P., Naik, M., Park, C.S., Sen, K.: Calfuzzer: An extensible active testing framework for concurrent programs. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. pp. 675–681. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
7. Lange, J., Ng, N., Toninho, B., Yoshida, N.: Fencing off go: Liveness and safety for channel-based programming. In: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages. p. 748–761. POPL 2017, Association for Computing Machinery, New York, NY, USA (2017). https://doi.org/10.1145/3009837.3009847, https://doi.org/10.1145/3009837.3009847
8. Lange, J., Ng, N., Toninho, B., Yoshida, N.: A static verification framework for message passing in go using behavioural types. In: Proceedings of the 40th International Conference on Software Engineering. p. 1137–1148. ICSE '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3180155.3180157, https://doi.org/10.1145/3180155.3180157
9. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation. p. 446–455. PLDI '07, Association for Computing Machinery, New York, NY, USA (2007). https://doi.org/10.1145/1250734.1250785, https://doi.org/10.1145/1250734.1250785
10. Ng, N., Yoshida, N.: Static deadlock detection for concurrent go by global session graph synthesis. In: Proceedings of the 25th International Conference on Compiler Construction. p. 174–184. CC 2016, Association for Computing Machinery, New York, NY, USA (2016). https://doi.org/10.1145/2892208.2892232, https://doi.org/10.1145/2892208.2892232
11. Serebryany, K., Iskhodzhanov, T.: Threadsanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. p. 62–71. WBIA '09, Association for Computing Machinery, New York, NY,

USA (2009). https://doi.org/10.1145/1791194.1791203, https://doi.org/10.1145/1791194.1791203

12. Stadtmüller, K., Sulzmann, M., Thiemann, P.: Static trace-based deadlock analysis for synchronous mini-go. In: Igarashi, A. (ed.) Programming Languages and Systems. pp. 116–136. Springer International Publishing, Cham (2016)
13. Sulzmann, M., Stadtmüller, K.: Trace-based run-time analysis of message-passing go programs. CoRR **abs/1709.01588** (2017), http://arxiv.org/abs/1709.01588
14. Sulzmann, M., Stadtmüller, K.: Two-phase dynamic analysis of message-passing go programs based on vector clocks. PPDP '18, Association for Computing Machinery, New York, NY, USA (2018). https://doi.org/10.1145/3236950.3236959, https://doi.org/10.1145/3236950.3236959
15. Thomson, P., Donaldson, A.F., Betts, A.: Concurrency testing using schedule bounding: An empirical study. In: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. p. 15–28. PPoPP '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2555243.2555260, https://doi.org/10.1145/2555243.2555260
16. Vyukov, D., Gerrand, A.: Introducing the go race detector (2013), https://blog.golang.org/race-detector
17. Yu, J., Narayanasamy, S., Pereira, C., Pokam, G.: Maple: A coverage-driven testing tool for multithreaded programs. p. 485–502. OOPSLA '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2384616.2384651, https://doi.org/10.1145/2384616.2384651
18. Zhao, J., Abe, H., Nomura, Y., Cheng, J., Ushijima, K.: Runtime detection of communication deadlocks in occam 2 programs pp. 97–107 (1997)