GOBLINT: Autotuning Thread-Modular Abstract Interpretation
(Competition Contribution)

Simmo Saan[1(✉)⋆], Michael Schwarz[2], Julian Erhard[2], Manuel Pietsch[2], Helmut Seidl[2], Sarah Tilscher[2], and Vesal Vojdani[1]

[1] University of Tartu, Tartu, Estonia
{simmo.saan,vesal.vojdani}@ut.ee
[2] Technische Universität München, Garching, Germany
{m.schwarz,julian.erhard,m.pietsch,helmut.seidl,sarah.tilscher}@tum.de

Abstract. The static analyzer GOBLINT is dedicated to the analysis of multi-threaded C programs by abstract interpretation. It provides multiple techniques for increasing analysis precision, e.g., configurable context-sensitivity and a wide range of numerical analyses. As a rule of thumb, more precise analyses decrease scalability, while not always necessary for solving the task at hand. Therefore, GOBLINT has been enhanced with *autotuning* which, based on syntactical criteria, adapts analysis configuration to the given program such that *relevant* precision is obtained with acceptable effort.

1 Verification Approach

GOBLINT is a static analysis framework for C programs based on abstract interpretation [6]. It features scalable thread-modular analysis of concurrent programs on top of flow- and context-sensitive interprocedural analysis. The analysis is specified as a side-effecting constraint system [2], which can conveniently express flow-insensitive invariants as well as flow-sensitive information per program point [16] and is solved using a local generic solver [15]. Here, we detail some recent SV-COMP–related advances in GOBLINT. The previous competition tool paper [11] provides further details on the general approach.

New abstract domains have been added to enhance precision. In addition to interval analysis of integer variables, GOBLINT now performs interval analysis of floating-point variables following Miné [9], and maintains congruence information [7]. Furthermore, the APRON library [8] has been integrated for relational analysis. GOBLINT includes novel approaches to relational analysis of concurrent programs [14], inferring relations between jointly-protected global variables.

In the previous tool paper, we suggested dynamically tailoring GOBLINT to the program under analysis. This can increase precision, by activating analyses that are more expensive yet offer crucial precision, and also decrease resource

⋆ Jury member

S. Sankaranarayanan and N. Sharygina (Eds.): TACAS 2023, LNCS 13994, pp. 547–552, 2023.
https://doi.org/10.1007/978-3-031-30820-8_34

usage, by deactivating redundant analyses. To this end, we have implemented analysis configuration *autotuning* based on cheap *syntactic* heuristics on the program, before the analysis begins. The particular features have been chosen according to how expert users might configure Goblint for a given program. Measurements of program size (e.g. number of functions, loops, variables) are taken into account to limit slowdown on larger programs.

Goblint provides a multitude of concurrency-related analyses (e.g. races, symbolic locking patterns, thread joins [14, 16]) that have no use in single-threaded programs which abound in SV-COMP. Hence, all such analyses are now automatically deactivated for programs that never create any threads.

Goblint implements a wide variety of numerical abstract domains, but most are not necessary for every program, thus, offering many possibilities for autotuning. Interval information is omitted in calling contexts of recursive functions to avoid an explosion of contexts in which they are to be analyzed. While the congruence domain is generally active on small programs, for medium-sized programs it is only enabled for functions involving the modulo operator, either directly or indirectly (up to fixed depth in the call stack). If the program uses enums, then an integer domain for sets of enumeration values is activated. Octagon analysis is enabled for those local variables which occur most often in linear expressions and conditions. Interval and octagon widening thresholds are extracted from conditional expressions containing constants. Such thresholds are especially useful for flow-insensitive analysis of global variables in multi-threaded programs, since no narrowing is performed on flow-insensitive invariants.

Loop unrolling is a well-known technique to increase the precision of static analysis. Goblint now unrolls loops up to their static bounds or feasible unrolled code size. Loops which contain memory allocation, thread creation, or error function calls, are prioritized since unique heap locations and threads are key to maintaining analysis precision.

Schwarz et al. [13] enhanced Goblint with a suite of concurrent value analyses and evaluated their precision. Following their observations, we use the cheap yet sufficiently precise *Protection-Based Reading*. Data-race detection was made more precise using *may-happen-in-parallel* analysis [14], to filter out spurious races with threads that have already been joined or have not yet been created.

2 Software Architecture

Goblint is implemented in OCaml and uses an updated fork of CIL [10] as its parser frontend for the C language. It depends on Apron [8] for relational analyses. No other major libraries or external tools are required.

Goblint employs a modular architecture [1] where a combination of analyses can be selected at runtime. Analyses are defined through their abstract domains and transfer functions, which can communicate with other analyses using predefined queries and events. The combined analyses together with the control-flow graphs of the functions yield a side-effecting constraint system [2],

which is solved using a local generic solver [15]. The solution is post-processed to determine the verdict and construct a witness.

3 Strengths and Weaknesses

GOBLINT focuses on *sound* static analysis which is confirmed by the competition: our tool does not produce any incorrect results. A major limitation of our approach is that, due to over-approximation, the tool can only prove the absence of bugs, but not their presence. Thus, when GOBLINT flags a potential violation, it answers "unknown" in the competition.

In SV-COMP 2023, *NoDataRace* became an official category and existing *ConcurrencySafety* reachability tasks were newly included into it. This is where GOBLINT really shines: it proves 652 out of 783 programs race-free, thereby winning the category. Overall, the strengths and weaknesses of GOBLINT w.r.t. categories remain the same as described in our previous tool paper. Therefore, we describe here the impact of autotuning, based on our own preliminary comparative evaluation. Unlike official SV-COMP evaluation, we used a 1 GB memory limit, which is sufficient for most tasks GOBLINT can solve, and no witness validators.

As noted above, the majority of SV-COMP programs across all categories are single-threaded, thus, the greatest improvement comes from disabling all concurrency analyses in those cases. This yields a notable reduction in runtime and memory usage as shown in table 1, improving overall efficiency without compromising precision.

The second greatest improvement is due to the use of relational analysis with octagons. Although this incurs a runtime penalty, it increases the number of correct verdicts notably. The improvement is especially visible in *NoOverflows*, where it yields 104 additional correct results. We also confirmed that the automatic selection of octagon variables is better than tracking all variables: our selection yields more correct verdicts (due to fewer timeouts) while successfully avoiding an unnecessarily large performance penalty.

Autotuning along the other axes is not as impactful. Nevertheless, each leads to GOBLINT being able to solve tasks it could not otherwise. Hence, a small increase in score is achieved, justifying their use. Although disabling unnecessary

Table 1. Reduction in resource usage due to disabling all concurrency analyses for single-threaded programs, as reported by BENCHEXEC using ordinary least squares (OLS) regression.

	unreach-call		*no-overflows*	
Tasks	CPU time	Memory	CPU time	Memory
Correct only	16%	4%	5%	0%
All	5%	8%	16%	6%

concurrency analyses reduces resource usage, overall this performance improvement is canceled out by the simultaneous use of expensive analyses enabled by autotuning, such as octagons. Thus, Goblint can solve more tasks while retaining the same level of overall efficiency observed in previous editions of the competition [3].

Many future opportunities for autotuning exist: Goblint implements a number of concurrent value analyses offering different tradeoffs between time and precision [13, 14], but only used the fastest and least precise of these in SV-COMP. If appropriate heuristics for using the more involved analyses are identified, autotuning could enable these when they are likely to yield a benefit. Autotuning could be extended to supply a sequence of configurations, increasing in precision, for a portfolio of analyses, instead of relying on the autotuning to immediately pick the most appropriate configuration. While the current autotuning in Goblint is hand-crafted, machine learning may provide additional improvements.

4 Tool Setup and Configuration

Goblint version `svcomp23-0-g4f5dcf38f` participated in SV-COMP 2023 [4, 12]. It is available in both binary (Ubuntu 22.04) and source code form at our GitHub repository under the `svcomp23` tag.[3] The only runtime dependency is Apron [8]. Instructions for building from source can be found in the `README`.

Both the tool-info module and the benchmark definition for SV-COMP are named `goblint`. They correspond to running the tool as follows:

```
./goblint --conf conf/svcomp23.json \
          --set ana.specification property.prp input.c
```

Goblint participated in the following categories: *ReachSafety*, *ConcurrencySafety*, *NoOverflows*, *SoftwareSystems* and *Overall*, while opting-out from *MemSafety*, *Termination* and *SoftwareSystems-*-MemSafety*.

5 Software Project and Contributors

Goblint development takes place on GitHub,[4] while related publications are listed on its website.[5] It is an MIT-licensed joint project of the Technische Universität München (Chair of Formal Languages, Compiler Construction, Software Construction) and University of Tartu (Laboratory for Software Science).

Acknowledgements. This work was supported by Deutsche Forschungsgemeinschaft (DFG) – 378803395/2428 ConVeY and the Estonian Centre of Excellence in IT (EXCITE), funded by the European Regional Development Fund. We would like to thank everyone who has contributed to Goblint over the years, especially the students who contributed various autotunable analyses.

[3] https://github.com/goblint/analyzer/releases/tag/svcomp23
[4] https://github.com/goblint/analyzer
[5] https://goblint.in.tum.de

Data Availability. All data of SV-COMP 2023 are archived as described in the competition report [4] and available on the competition web site. This includes the verification tasks, results, witnesses, scripts, and instructions for reproduction. The version of GOBLINT as used in the competition is archived together with other participating tools [5] and individually [12] on Zenodo.

Bibliography

[1] Apinis, K.: Frameworks for analyzing multi-threaded C. Ph.D. thesis, Technische Universität München (2014)

[2] Apinis, K., Seidl, H., Vojdani, V.: Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In: APLAS '12, pp. 157–172, Springer (2012), DOI: 10.1007/978-3-642-35182-2_12

[3] Beyer, D.: Progress on software verification: SV-COMP 2022. In: TACAS '22, pp. 375–402, Springer (2022), DOI: 10.1007/978-3-030-99527-0_20

[4] Beyer, D.: Competition on software verification and witness validation: SV-COMP 2023. In: Proc. TACAS (2), LNCS , Springer (2023)

[5] Beyer, D.: Verifiers and validators of the 12th Intl. Competition on Software Verification (SV-COMP 2023). Zenodo (2023), DOI: 10.5281/zenodo.7627829

[6] Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77, pp. 238–252 (1977), DOI: 10.1145/512950.512973

[7] Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics **30**(3-4), 165–190 (1989), DOI: 10.1080/00207168908803778

[8] Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: CAV '09, pp. 661–667 (2009), DOI: 10.1007/978-3-642-02658-4_52

[9] Miné, A.: Relational abstract domains for the detection of floating-point run-time errors. In: ESOP '04, pp. 3–17, Springer (2004), DOI: 10.1007/978-3-540-24725-8_2

[10] Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: CC '02, pp. 213–228, Springer (2002), DOI: 10.1007/3-540-45937-5_16

[11] Saan, S., Schwarz, M., Apinis, K., Erhard, J., Seidl, H., Vogler, R., Vojdani, V.: Goblint: Thread-modular abstract interpretation using side-effecting constraints. In: TACAS '21, pp. 438–442 (2021), DOI: 10.1007/978-3-030-72013-1_28

[12] Saan, S., Schwarz, M., Erhard, J., Pietsch, M., Seidl, H., Tilscher, S., Vojdani, V.: Goblint at SV-COMP 2023 (Nov 2022), DOI: 10.5281/zenodo.7467093, tool artifact

[13] Schwarz, M., Saan, S., Seidl, H., Apinis, K., Erhard, J., Vojdani, V.: Improving thread-modular abstract interpretation. In: SAS '21, pp. 359–383, Springer (2021), DOI: 10.1007/978-3-030-88806-0_18

[14] Schwarz, M., Saan, S., Seidl, H., Erhard, J., Vojdani, V.: Clustered relational thread-modular abstract interpretation with local traces. In: ESOP '23, Springer (2023)
[15] Seidl, H., Vogler, R.: Three improvements to the top-down solver. Mathematical Structures in Computer Science p. 1–45 (2022), DOI: 10.1017/S0960129521000499
[16] Vojdani, V., Apinis, K., Rõtov, V., Seidl, H., Vene, V., Vogler, R.: Static Race Detection for Device Drivers: The Goblint Approach. In: ASE '16, pp. 391–402, ACM (2016), DOI: 10.1145/2970276.2970337