Towards Compiler-Guided Static Analysis

Benjamin Mikek
Georgia Institute of Technology
Atlanta, USA
bmikek@gatech.edu

Abstract

Static analysis consists of a large body of tools that analyze programs without running them. Despite substantial progress in recent years, both theoretical and practical, static analyses still face bottlenecks in SMT solving, equivalence checking, and compiler verification. This work proposes alleviating these bottlenecks by using harnessing existing compilers, and asks whether their internal state and the transformations they perform can inform static analysis and expand the set of tractable problems. We plan to investigate the problem by recording the analyses and optimizations a compiler performs on a particular input program. In existing work, we find that the strategy is promising by showing that compilers can speed up SMT solving. Our proposal is to extend the work to a general framework for recording and filtering the actions performed by a compiler and to evaluate it on translation validation and other use cases.

CCS Concepts: • **Software and its engineering → Compilers**; **Formal software verification**.

Keywords: compiler instrumentation, SMT solving, translation validation

ACM Reference Format:
Benjamin Mikek. 2025. Towards Compiler-Guided Static Analysis. In *Companion Proceedings of the 2025 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '25), October 12–18, 2025, Singapore, Singapore.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3758316.3762823

1 Motivation

Static analysis tools analyze programs without running them and can provide formal guarantees that a particular program is secure, correct, or performant. Static analysis techniques have seen impressive progress over the last 20 years, and static analysis is increasingly used in practical applications, both for safety-critical and commercially important programs. The recent influx of LLM-generated code into software projects has further increased the need for formal analysis to catch bugs or behavioral differences in code not written by human programmers.

Recent advances in static analysis have included the development of new and more efficient SMT solvers [5], better translation validation tools [8], and new methods for reachability analysis. Despite these advances, though, many analysis tasks of great importance remain out of reach. These include, *inter alia*, the bottleneck of SMT solving, translation validation, and the related problem of compiler verification. In each case, providing information beyond that available in the syntax of the program can enable an analysis to perform better. Take as an example the problem of translation validation: proving that a program optimized by a compiler is semantically equivalent to the original. Translation validation of individual statements and straight-line programs can be done by solving constraints representing the semantics before and after optimization. But for programs with complex control flow, including loops, it is challenging to "line up" program statements to verify equivalence; doing so requires extra information not present in the program syntax [3]. We hypothesize that this extra-syntactic information can be derived from a compiler, thus enabling translation validation in the presence of complex control flow.

This work proposes a deeper investigation of the potential of compilers for enabling the better performance of program analyses without the downsides of dynamic analysis. In existing work on SMT solving, we have found that the reasoning performed by compilers can be used to speed up SMT solving. We aim to discover whether compiler-derived information is also useful in alleviating the bottlenecks in translation validation and other static analyses by developing a general framework for harnessing compilers' transformation and analysis capabilities to improve static analysis. Our proposed approach involves extracting and filtering compiler-internal information—the result is to a compiler what eBPF is to the Linux kernel. Reducing the bottlenecks of static analysis would allow tools to verify more optimizations, solve larger sets of constraints, and analyze larger LLM-generated codebases—in short, to increase users' trust in more code.

2 Problem

This proposal aims to tackle many program analysis problems with a single class of solution. In particular, the key bottlenecks we address are SMT solving, which is central to applications like symbolic execution [2], translation validation [8], and the related problem of compiler verification [7].

SPLASH Companion '25, Singapore, Singapore

ACM ISBN 979-8-4007-2141-0/25/10
https://doi.org/10.1145/3758316.3762823

These applications are diverse—they originate in different theoretical domains and have historically been approached from different angles of attack in practice—but they share one key common feature: the need for program-external information to inform the analysis. Our approach is to obtain this extra-syntactic information from a compiler. Formally, we address the following problem:

> Construct a general framework for harnessing a compiler's transformations to provide information that improves multiple static analysis applications while allowing use-specific filtering of results.

In the SMT solving context, compiler information may be useful in optimizing constraints directly, or in informing how they should be optimized. In the context of translation validation, records of compiler transformations allow programs subject to complex control flow transformations to be stitched back together, enabling verification. In both cases, compiler information is the key to improving analysis.

The key challenge for the approach lies in identifying the right balance between completeness and relevance for the collected compiler information. On the one hand, access to more information makes analysis richer, and can, for example, enable additional types of simplification for constraints. On the other hand, instrumenting every low-level action of a compiler, like the peephole optimization which transforms two instructions into a single one, can provide a flood of actions and data which is difficult for an analysis tool to sort through. The challenge lies in balancing these two phenomena: the compiler-derived information must be sufficient for a particular use case but must also rest at a high enough level of abstraction to allow meaningful use by an analysis tool.

3 Approach

Completed Work. We have already found that compiler-derived information can improve SMT solving. SMT constraints are logical formulas that extend the classic SAT problem. SMT solving forms the core of static analyses as diverse as symbolic execution, termination proving, and program verification, yet even state-of-the-art SMT solvers still time out on thousands of constraints. In our work on SMT–LLVM Optimizing Translation (SLOT) [9], we found that using compiler optimizations speeds up SMT solving by simplifying constraints. SLOT works by translating a constraint into the intermediate representation of the popular compiler framework LLVM, applying LLVM's optimization passes, and then translating back. The approach achieves speedups as high as $3x$ on large constraints, and renders tractable thousands of constraints that existing solvers cannot handle. In followup work, we introduced SMT theory arbitrage [10], which uses a novel abstract interpretation approach to convert unbounded constraints into bounded ones, achieving *further* speedups up to 40% and unlocking compiler optimizations for the unbounded theories of integers and real numbers. Theory arbitrage demonstrates the power of compiler-derived information in improving constraint solving even for unbounded constraints which are not directly related to compilers but have important roles in static analysis, including for termination proving [6] and modeling automata [4].

Next Steps. To show the generality and power of our approach, we turn to the question of translation validation. Given an input program P and some transformation process $\mathcal{T}$ (usually a compiler), which produces a transformed program P', the goal is to determine whether P' preserves the semantics of P. For simple straight-line programs, this can be achieved by directly encoding both programs as constraints and then taking advantage of SMT solvers to compare their semantics. However, approaches along these lines like Alive2 [8], can only handle complex control flow up to a fixed bound, and are not suitable for verifying the equivalence of programs where loop structure has been changed. One technique for handling loops is *program alignment* [3], where the statements of two programs are matched up and tagged with comparison predicates, thereby allowing reasoning about equivalence piece-by-piece. Aligning two programs, though, requires some external information about which parts of P to match up with which parts of P'. Existing approaches [3] derive this information from dynamic analysis of the underlying program by collecting traces. We instead propose to obtain the information by instrumenting the compiler.

Our first planned milestone is to handle the translation validation use case. To do so, we instrument a subset of LLVM to produce *transformation records* of the optimizations performed on some input program. Consider for example loop vectorization, where a loop with body B is vectorized by LLVM. The vectorized program will include a vector version of B, call it B_v, and a "cleanup loop" C_v which performs any excess iterations of B not handled by B_v. Suppose the *vectorization factor* is k. To verify equivalence, we need to align p iterations of B, with p/k iterations of B_v and $p \mod k$ iterations of C_v, as shown in Equation 1. Existing literature attempts to derive the realtionship by collecting execution traces [3]; we instead replace the execution information with information obtained by instrumenting LLVM's `LoopVectorize` pass, extracting Equation 1 by instrumenting the calls that create B_v and C_v from B. The instrumentation approach enables translation validation for loops without collecting traces.

$$B^p \sim B_v^{\left(\frac{p}{k}\right)} C_v^{(p \mod k)} \tag{1}$$

The next milestone, and the heart of this research work, is to generalize—to develop a framework for extracting information from an existing compiler and applying it to client static analyses. Generalization will require capturing both

compiler-internal information through *instrumentation*, and observing the transformations the compiler performs on input programs. Given a record of the actions of the compiler, we must next *filter* these actions to only those which are relevant for a given application. For example, instrumentation might hook a low-level LLVM API such as the `Instruction` class, but most instruction-level changes would be irrelevant to the program alignment problem, where the appropriate level of abstraction is the basic block. Thus, the results of instrumentation should be filtered to focus on operations over LLVM's `BasicBlock` class. Choosing the right degree of filtering will require a lightweight framework that allows users to specify which types of compiler actions should be kept during filtering by dividing them according to LLVM class, level of abstraction, and type of action.

Risks and Mitigations. While our existing work shows that compilers have great promise in providing information to improving static analysis, generalization faces two key obstacles. First, it is possible that collecting a complete low-level transformation record and then filtering is too costly for performance-critical static analyses. In this case, we may reduce the extent of filtering and instead raise the level of abstraction where instrumentation is performed. This could be achieved, for instance, by allowing users to specify a tradeoff between runtime and detail. Second, the SMT solving and translation validation applications may not be general enough to fully exploit the potential of compiler information. In this case, we may turn to additional applications such as symbolic execution. This application already uses some compiler optimizations to reduce workload, but could further benefit from the guidance contained in debug information or other records of the transformations a compiler chooses to perform on a subject program.

4 Evaluation Methodology

The question of whether compiler-derived information is useful for informing program analysis comes in two parts:

- **RQ1:** Do compilers provide enough information to guide static analyses around existing bottlenecks?
- **RQ2:** Can compiler information useful for a practical application be effectively extracted from the compiler?

Existing work provides a direction to follow for both questions. The effectiveness of SLOT on a large set of standard SMT benchmarks [1] suggests that the answer to RQ1 is in the affirmative, at least for the SMT application. To evaluate whether it is true in general, we plan to investigate the extent to which instrumentation of LLVM provides enough information to align programs for translation validation. This will involve experiments on benchmark programs with complex loop structures optimized by LLVM, where the relevant evaluation metric is the number of programs for which compiler instrumentation leads to a verification result, but existing approaches do not.

For RQ2, the existing results for SLOT and theory arbitrage show one use case in which compilers' reasoning is directly harvested by applying optimizations to constraints. To fully answer the question, though, we intend to evaluate also on the translation validation context to determine whether each structure-transforming pass can be instrumented to produce information that can align transformed programs at the correct level of abstraction. Here, evaluation also includes experiments on validating translations of benchmark programs, but the evaluation metric focuses on the compiler instead: how many passes can be usefully instrumented and which passes, and for what reasons, cannot be handled?

Both research questions also relate to the larger issue of generality—for how many use cases does compiler information provide benefits. Preliminary results on transformation records for LLVM's vectorization show that the approach has promise for translation validation. To fully investigate this question, though, we plan to generalize the framework further, allowing users to specify levels of abstraction for instrumentation and filtering. In the end, a general instrumentation framework would be to LLVM what eBPF is to Linux. It would provide a way to extract relevant compiler information without including low-level details that make interpretation a challenge.

References

[1] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2017. *The SMT-LIB Standard: Version 2.6.* Technical Report. Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI 2008*. USENIX Association, USA, 209–224.

[3] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *PLDI 2019*. ACM, New York, NY, USA, 1027–1040. doi:10.1145/3314221.3314596

[4] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. 2012. A quantifier-free SMT encoding of non-linear hybrid automata. In *Formal Methods in Computer-Aided Design, FMCAD 2012*. IEEE, 187–195.

[5] et al. Haniel Barbosa. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *TACAS 2022 (Lecture Notes in Computer Science, Vol. 13243)*. Springer, 415–442. doi:10.1007/978-3-030-99524-9_24

[6] Jera Hensel, Constantin Mensendiek, and Jürgen Giesl. [n. d.]. AProVE: Non-Termination Witnesses for C Programs - (Competition Contribution). In *TACAS 2022*. 403–407. doi:10.1007/978-3-030-99527-0_21

[7] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. doi:10.1145/1538788.1538814

[8] Nuno P. Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. 2021. Alive2: bounded translation validation for LLVM. In *PLDI 2021*. ACM, New York, NY, USA, 65–79. doi:10.1145/3453483.3454030

[9] Benjamin Mikek and Qirun Zhang. 2023. Speeding up SMT Solving via Compiler Optimization. In *ESEC/FSE 2023*. ACM, New York, NY, USA, 1177–1189. doi:10.1145/3611643.3616357

[10] Benjamin Mikek and Qirun Zhang. 2024. SMT Theory Arbitrage: Approximating Unbounded Constraints using Bounded Theories. *Proc. ACM Program. Lang.* 8, PLDI, Article 157 (June 2024), 26 pages. doi:10.1145/3656387

Received 2025-07-31; accepted 2025-08-16