Optimizing Compilers for Safety-Critical Robotic Systems

Dietmar Schreiner

Vienna University of Technology
Institute of Computer Languages, Compilers and Languages Group

schreiner@complang.tuwien.ac.at

Abstract. Mobile autonomous robots are systems that on the one hand are seriously constricted to processing power, energy consumption and execution time, but on the other hand typically have to issue strict safety guarantees. A robot must consume as little power and CPU cycles as possible in order to maximize its sustainability but also its reactivity. In addition, a robot has to operate under strict safety conditions as it might inflict serious damage to the real world or even human beings. Safety properties in robotic software are often proven at source code or even model level. A well established methodology for improving execution time and energy consumption is that of compiler optimization. When translating a given program into machine executable code, an optimizing compiler transforms the original program into an improved but semantically equivalent one. Unfortunately, safety properties specified and verified at source code level might be violated in a transformed program. Hence, aggressive compiler optimization is not suitable for state-of-the-art safety-critical robotic systems. This paper outlines ongoing work on optimizing compilers that perform translation validation for all executed program transformations. Consequently, the outcome of optimizing transformations is formally verified and therefore can be applied to safety-critical robotic systems.

1 Motivation

Autonomous mobile robotic systems are steadily growing in terms of number but also of importance. Rescue operations in hazardous areas, surveillance or reconnaissance missions in hostile environments, but also less spectacular but nevertheless highly important tasks like auto-piloting of vehicles in public traffic or service robots have already found their way into reality. In all these use-cases robots closely interact with their environment, the real world. Although a robot's application has a high benefit, it also introduces high potential for serious damage or even casualties in case of faulty or erroneous behavior. In consequence, software for safety-critical systems has to be certified, which implies verification of the over-all system and especially its software.

As the level of autonomy and mobility increases, energy consumption becomes an important concern for robotic systems. By bringing their own battery (and

hence leaving stationary power supplies behind) robotic systems have to be resource saving and energy aware. To maximize the sustainability (the time of operation) of a mobile robot, energy saving hardware as much as resource aware software becomes mandatory. In consequence, software has to be executed on rather slow but low-power hardware. Additionally, reactive systems like robotic devices demand calculations to terminate within reasonable time. As one can easily see, software for robotic devices has to close the gap between cheap (in terms of energy consumption) hardware and expensive (in terms of complexity and runtime) algorithms.

One technique to improve software performance (less energy consumption, less CPU cycles) on a given hardware platform is that of compiler optimization. An optimizing compiler not only transforms a given source program into machine executable form but also modifies or even rewrites the program in order to be more efficient.

Unfortunately, software certification is often related to source code. Static properties, like for example bounds of array indices or loops, are checked at source code level. Any following transformation from source code to machine executable code is assumed to be correct as either a verified compiler is used, or a widely used compiler (typically open source) is applied which is considered to be correct. Both cases are not satisfactory. Existing verified compilers typically cover only a reduced subset of languages like C or C++ and do exist for selected target platforms only. Widely used compilers tend do be modified frequently and every modification has potential to introduce new flaws.

The solution we aim at is a compiler framework that performs optimizations but provides a proof of correctness for each transformation in terms of the original source code. Hence, all properties certified at source code level are proven to be present within the optimized machine executable binary.

2 Foundations for Correct Compilation

A compiler is a tool that translates a program from one representation into another, typically from human readable source code into machine executable binary code. In general two main directions for the verification of compilers can be found in literature. One aims at a verified compiler, while the other aims at verified translations.

Following the idea of a verified compiler [1–3], all algorithms used by the compiler have to be formally proven. Then the compilers implementation has to be proven with respect to the algorithms. As a result, a verified compiler is created and can be trusted in terms of correct transformations. The down-side of this approach is the rather high complexity in formalizing algorithms and sound theorems, leading to high costs, little variability when it comes to distinct target platforms, and little flexibility for future changes.

The second approach does not aim at a verified compiler itself but at verified results of a compiler run. This technique is called translation validation [4–6] and verifies that the output of a compiler (the machine executable code) is a correct translation of the source code fed into the compiler. The main difference between these approaches is that in translation validation a valid proof can be achieved even if the compiler is incorrect but performs the one correct translation that is validated. As the compiler is not subject to verification (typical for translation validation) it might be altered without impact on costs and flexibility. However, there is a downside on translation validation: As only the result of a specific translation/transformation is verified, the verification procedure has to be applied to every single program transformed by the compiler.

Another approach to assure correctness of compiled programs is that of proof-carrying code (PCC) [7, 8]. Here, a compiler that has not to be verified transforms the source program into machine executable code, and in addition generates facts, which are used to reason about a program's behavior before execution. In that way, the execution environment is enabled to check if all constraints implied by the source program's semantic hold within the compiled program.

3 The C3Pro Approach

Within research project C3Pro[1] (Correct Compilers for Correct Application Specific Processors) we merge the translation validation approach with the proof-carrying code idea. However, in our approach the concept of proof-carrying code is applied to intermediate code rather than to the final machine executable code. Hence, in our case PCC constraints are not checked by the execution environment but by subsequent compiler phases.

Results of the Verifix project [9, 10] show that a complex transformation like the translation of source code to machine executable code can be partitioned into a sequence of smaller sub-transformations in such way that correctness can be shown for every single sub-transformation. The overall transformation's correctness is guaranteed if every sub-transformation is correct [11]. Consequently we isolated a series of relevant and promising compiler passes and optimizations within an full fledge industry C compiler[2] in order to reassemble them to a translation validation and PCC aware compiler that does correct compilation. The original industry compiler exhibits 84 different passes with a total of 25 mandatory ones.

Figure 1 depicts the workflow within a compiler as proposed by our approach. Source code (Src Code) is translated into an intermediate representation (IR Code) by a specific compiler pass (CPass n). In addition the compiler pass emits PCC facts (Witness) for the generated code. In C3Pro we have developed the

[1] `http://www.complang.tuwien.ac.at/c3pro`
[2] `http://www.catena.nl/`

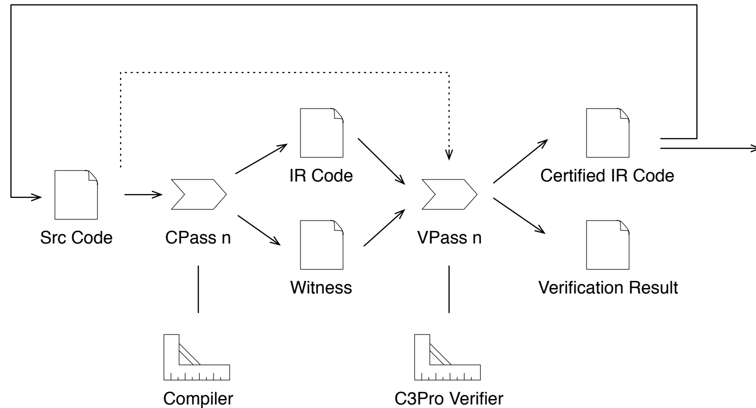

Fig. 1. Workflow of Compilation

meta-language *Hydra*, which is used to hold both IR code and PCC facts. Every compiler pass is followed by a verifier pass. The C3Pro verifiers check the correctness of the associated compiler pass taking the original program, the transformed one, and the witness facts into account. This procedure is iteratively applied to all passes. If all passes are correct, the over-all translation is correct.

By analyzing all passes of an industry C compiler, it turned out that only 4 types of verifiers, each representing one specific methodology, are needed for our approach:

Graph Transformation: This type of verifier checks properties that are completely decoupled from a programs semantic. Specific nodes or even sub-trees in a program's abstract syntax tree are exchanged in accordance to predefined static rules. Arithmetic optimization is an example for a compiler pass that can be validated by this type of verifier.

Fixed Point Iteration: Control flow or data flow properties can be calculated and verified by calculating fixed points on a programs control flow graph and abstract syntax tree. Loop-invariant code motion is an example for an optimization that can be validated by this type of verifier.

Theorem Proving: Logic formulas typically derived from simulation traces in state-transition systems are evaluated by a theorem prover. The SSA (static single assignment) transformation is an example of a compiler pass that can be covered by this type of verifier.

Abstract Execution of State-Transition Systems: By modeling the instructions' semantic as abstract state machines, execution traces can be verified with respect to side effect freeness or semantic equivalency. Instruction scheduling is one example of a compiler pass that can be validated by this methodology.

Most of these analyzers operate with almost linear complexity. However, those that are not that nice in terms of run-time do not harm the usability of our approach. Every single sub-transformation can be proven in isolation. A complete correctness proof has to be found only once after development is finished. Therefore, time consuming verifiers can be run at the finalization phase of a project, while all others can be calculated continuously.

A drawback of the proposed methodology is that we rely on the correctness of verifiers and a theorem prover. Iff one of these tools is faulty, the result of the verifiers becomes meaningless. Hence, those tools have to be considered "trusted code". However, there is good reason to pursue our approach: Our trusted code base contains rather simple and small tools that can be verified (even manually) at rather low costs. In addition, these tools do not have to be verified if the compiler is changed, rewritten, or even reworked from scratch. The trusted code required for our approach is decoupled from the work pice, the compiler.

4 Conclusion

Within this paper we discussed the relevance of correct optimizing compilers for safety-critical robotic systems and issued a position statement on correct compilation. We outlined a methodology for translation validation in combination with proof carrying code for intermediate representations in optimizing compilers. As this work is preliminary and in progress future work will include the implementation of all verifiers for the selected mandatory passes in order to prove the validity of our approach.

References

1. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43** (2009) 363–446 10.1007/s10817-009-9155-4.
2. Strecker, M.: Formal verification of a java compiler in isabelle. In Voronkov, A., ed.: Automated Deduction-CADE-18. Volume 2392 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2002) 135–163 10.1007/3-540-45620-1_5.
3. Blazy, S., Dargaye, Z., Leroy, X.: Formal verification of a c compiler front-end. In Misra, J., Nipkow, T., Sekerinski, E., eds.: FM 2006: Formal Methods. Volume 4085 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 460–475 10.1007/11813040_31.
4. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In Steffen, B., ed.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1384 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (1998) 151–166 10.1007/BFb0054170.
5. Necula, G.C.: Translation validation for an optimizing compiler. In: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. PLDI '00, New York, NY, USA, ACM (2000) 83–94

6. Barrett, C., Fang, Y., Goldberg, B., Hu, Y., Pnueli, A., Zuck, L.: Tvoc: A translation validator for optimizing compilers. In Etessami, K., Rajamani, S., eds.: Computer Aided Verification. Volume 3576 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2005) 273–286 10.1007/11513988_29.
7. Appel, A.: Foundational proof-carrying code. In: Logic in Computer Science, 2001. Proceedings. 16th Annual IEEE Symposium on. (2001) 247 –256
8. Necula, G.C.: Proof-carrying code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '97, New York, NY, USA, ACM (1997) 106–119
9. Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., Henke, F.W.V., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H., Zimmermann, W.: Compiler correctness and implementation verification: The verifix approach (1996)
10. Zimmermann, W.: On the correctness of transformations in compiler back-ends. In Margaria, T., Steffen, B., eds.: Leveraging Applications of Formal Methods. Volume 4313 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2006) 74–95 10.1007/11925040_6.
11. Goos, G., Zimmermann, W.: Verifying compilers and asms or asms for uniform description of multistep transformations. In Gurevich, Y., Kutter, P., Odersky, M., Thiele, L., eds.: Abstract State Machines - Theory and Applications. Volume 1912 of Lecture Notes in Computer Science. Springer Berlin / Heidelberg (2000) 281–297 10.1007/3-540-44518-8_11.