講演番号	031

Formally Verified Compilation in the Context of Functional Safety

Daniel Kästner [1)] Bernhard Schommer [1)] Alexander Rogovskyy [1)] Michael Schmidt [1)]
Adrian Dapprich [2)]

1) AbsInt GmbH
Science Park 1, 66123 Saarbrücken, Germany (E-mail: info@absint.com)
2) AbsInt GmbH
16-4-16 Numabukuro, Nakano, Tokyo, 165-0025, Japan (E-mail: dapprich@absint.com)

KEY WORDS: **software and its underlying technologies, ISO-26262, formal methods,** compiler, qualification (E3)

Modern compilers are highly complex software systems which contain many highly tuned and sophisticated algorithms. However, these can contain bugs. Multiple studies[(1-3)] have found numerous bugs in all investigated open source and commercial compilers, including compiler crashes and miscompilation issues. Although such wrong-code errors can be detected in the normal software testing stage it does not typically include systematic checks for them. When they occur in the field, they can be hard to isolate and to fix.

Whereas in non-critical software functional software bugs tend to have bigger impact than miscompilation errors, the importance of the latter dramatically increases in safety-critical systems. Contemporary safety standards such as ISO-26262, DO-178B/C, or IEC-61508 require identification of potential hazards and to demonstrate that the software does not violate the relevant safety goals. Many verification activities are performed at the architecture, model, or source code level, but all properties demonstrated there may not be satisfied at the executable code level when miscompilation happens. This is true, not only for source code review but also for formal, tool-assisted verification methods such as static analysers, deductive verifiers, and model checkers. Moreover, properties asserted by the operating system may be violated when its binary code contains wrong-code errors induced when compiling the OS. In consequence, miscompilation is a non-negligible risk that must be addressed by additional, difficult and costly verification activities such as more testing and more code reviews at the generated assembly code level.

CompCert compiler, the first formally-verified optimizing C compiler is available. What sets CompCert apart from any other production compiler, is that it is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues. In other words, the executable code it produces is proved to behave exactly as specified by the semantics of the source C program. This level of confidence in the correctness of the compilation process is unprecedented. In particular, using the CompCert C compiler is a natural complement to applying formal verification techniques (static analysis, program proof, model checking) at the source code level: the correctness proof of CompCert C guarantees that all safety properties verified on the source code automatically hold as well for the generated executable.

CompCert targets multiple 32-bit and 64-bit architectures common in embedded systems and recently added support for the 32-bit TriCore architecture for automotive applications.

The result of some MiBench benchmarks are illustrated in Fig. 1 which shows the execution time of the generated code. The experiments compare GCC (version 11.3) and CompCert (version 26.04) on three optimization levels. The results show that the optimized code generated by CompCert is comparable with GCC on optimization levels -O1 and -Os and often runs significantly faster than code generated by GCC without optimizations. The selected MiBench benchmarks cover various algorithms, like Image codecs (JPEG), cryptography (blowfish), checksums (SHA, CRC32) and some mathematical algorithms (FFT, etc.).

Regarding code size, the code generated by CompCert is usually on the same level or a bit larger than that of GCC with optimizations enabled and a little bit smaller than the code generated by GCC without optimizations. In general, due to lack of aggressive loop optimizations, in CompCert the performance is lower on high-performance computing codes involving dense matrix computations.

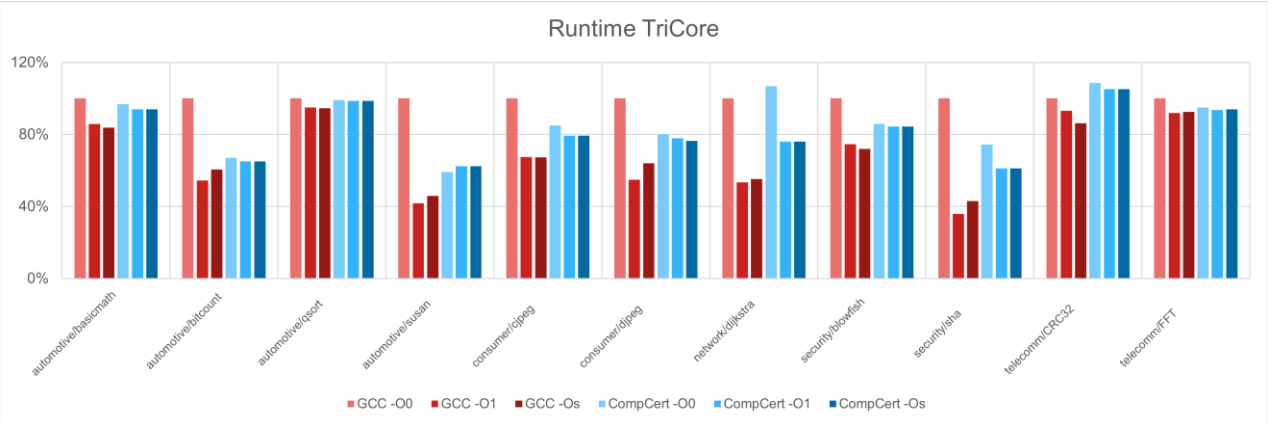

Fig. 1 Execution Time Comparison for selected MiBench Benchmarks on TriCore