# CompCert - A formally verified optimizing C Compiler for MORAL

## What is a Compiler?

A compiler translates source code written in a human-readable programming language to the machine code a target processor like Peaktop can understand and execute. During the translation a compiler may also apply sophisticated transformations and optimizations of the input program such that the generated machine code runs fast and efficient.

Those code generation and optimization processes are highly complex tasks and - as it is familiar with software - compilers too may contain bugs leading to wrong translations.

In fact, several studies [1][2][3] investigated common open-source and commercial compilers and have found numerous problems in them, including crashes of the compiler itself as well as miscompilation issues. Miscompilation here means that the compiler silently generates incorrect machine code from a correct source program.

## Trusting your Compiler

For safety-critical systems miscompilation is a serious problem since it can cause erroneous or erratic behavior including memory corruption and program crash, which may manifest sporadically and often is hard to track down.

Although such wrong-code errors can be detected during normal software testing it does not typically include systematic checks for them. Furthermore, many verification activities during software development are performed only at the architecture, model, or source code level, but all properties demonstrated there may no longer be valid at the machine code level when miscompilation happens.

In other words, activities to improve software quality and correctness like source code review, formal, or tool-assisted verification methods such as static analyzers, deductive verifiers, and model checkers focus their attention mainly onto the source code level. They neglect the compiler in assuming that the transformation of the software into machine code “works as intended”.

## Traditional Handling of the Miscompilation Problem

Miscompilation is a non-negligible risk. Because of its possible dramatic effects, many safety standards require additional, difficult and costly verification activities to show that the requirements already shown at higher levels are also satisfied at the executable object code level.

Today this is often done with a combination of “proven in use” arguments, the disabling of optimizations, and compiler test suites for qualification. However, none of these measures themselves are sufficient and neither is their combination.

The “proven in use” argument hopes that problems can be avoided by using an established and widely used compiler. I.e., it assumes that all bugs in the compiler are already found and fixed by someone else, but it cannot confidently exclude further problems.

Disabling optimizations is the attempt to reduce the complexity of a compiler's task by removing some of its transformation phases, hoping to avoid bugs occurring only in those phases. This is typically done for highly critical applications. However, it not only comes with the cost of lost performance - without optimizations, the translated program will be slower and less efficient. It also weakens the “proven in use” argument, since compilers are typically used with a standard set of optimizations and are less tested in other configurations.

Lastly, compiler test suites comprise a fixed set of test cases. The intention of these test suites is typically to show that the compiler correctly recognizes the input language — especially

all its corner cases. They also often contain tests that “stress” a compiler with selected large inputs — mostly to show that the compiler has the internal capacity to handle inputs of a certain complexity. For example such tests include deeply nested program structures or functions with many arguments.

Furthermore, compiler suites typically contain a collection of tests covering former regressions of (other) compilers. The argument here is that developers implementing a compiler for the same input language may come up with similar mistakes. What compiler test suites cannot do is to show the absence of miscompilations in general.

## Enter CompCert

CompCert is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation issues. In other words, the machine code it produces is proved to behave exactly as specified by the semantics of the source C program. CompCert therefore provides an unprecedented level of confidence in the correctness of the compilation process.

Since 2015 the CompCert compiler has been commercially available and within the MORAL project CompCert will be extended to support Peaktop as a target architecture.

Using CompCert has multiple benefits. First, the cost of finding and fixing compiler bugs and shipping patches to customers can be avoided. In the context of the MORAL project this is important since Peaktop is intended for space applications, where software problems can be very severe and impossible to fix afterwards.

Testing efforts required to ascertain software properties at the binary executable level can be reduced. The focus can instead be shifted to areas to improve software quality in general, e.g., by applying formal verification techniques (static analysis, program proof, model checking) at the source code level. CompCert's proofs guarantee that all safety properties verified on the source code automatically hold for the generated executable.

Lastly CompCert makes it feasible to utilize the full capabilities of the target hardware by allowing the use of code optimizations since now they finally can be trusted.

## CompCert Design

Like other compilers, CompCert is structured as a pipeline of compilation passes. Internally it uses up to 20 steps to translate from the input C language to the output assembly language. In between these translation steps, CompCert utilizes 11 intermediate languages gradually approaching the output language.

What sets CompCert apart from other compilers is that each of the involved languages has formal semantics, i.e., a mathematical specification that describes how a program written in this language must behave. A program behavior may e.g., be the trace of input and output operations of the program while running, whether the program itself terminates (either normally or caused by an error) or whether it runs forever.

For each translation step of CompCert a mathematical proof ensures that the transformation implemented by the step does not change the program behavior in unwanted ways. The correctness of the whole translation then follows from the composition of the small steps individually proven correct.

CompCert's proofs are a very huge, owing to the many passes and the many cases to be considered - too large to be carried out manually using pen and paper. Instead, CompCert utilizes machine assistance in the form of the Coq proof assistant. Coq allows one to write precise, unambiguous specifications, conduct proofs in interaction with the tool, and automatically re-check the proofs for soundness and completeness.

This results in very high levels of confidence in the proof. At 100,000 lines of code written in Coq and 6 person-years of effort, CompCert’s proof is among the largest ever performed with a proof assistant.

## CompCert in MORAL

CompCert will be used in the MORAL project to compile all software running on the finalized Peaktop processor. This includes the operating system as well as the demonstrator application.

## References


[1] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In EMSOFT ’08, pages 255–264. ACM, 2008.

[2] NULLSTONE Corporation. NULLSTONE for C. http://www.nullstone.com/htmls/ns-c.htm, 2007.

[3] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In PLDI ’11, pages 283–294. ACM, 2011.


## Further Reading


- D. Kästner, J. Barrho, U. Wünsche, M. Schlickling, B. Schommer, M. Schmidt, C. Ferdinand, X. Leroy, S. Blazy. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In ERTS 2018: Embedded Real Time Software and Systems, 9th European Congress, Jan 2018, Toulouse, France. Available at the HAL open archive, URL: https://hal.archives-ouvertes.fr/ERTS2018/hal-01643290v1.

- D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, C. Ferdinand. Closing the Gap - The Formally Verified Optimizing Compiler CompCert. In Proceedings of the 25th Safety-Critical System Symposium SSS 2017, Feb 2017, Bristol, UK.