CompCert – A Formally Verified Optimizing Compiler

Xavier Leroy[1] Sandrine Blazy[2] Daniel Kästner[3] Bernhard Schommer[3]
Markus Pister[3] Christian Ferdinand[3]

1: Inria Paris-Rocquencourt, domaine de Voluceau, 78153 Le Chesnay, France

2: University of Rennes 1 - IRISA, campus de Beaulieu, 35041 Rennes, France

3: AbsInt Angewandte Informatik GmbH. Science Park 1, D-66123 Saarbrücken, Germany

Abstract

CompCert is the first commercially available optimizing compiler that is formally verified, using machine-assisted mathematical proofs, to be exempt from miscompilation. The executable code it produces is proved to behave exactly as specified by the semantics of the source C program. This article gives an overview of the design of CompCert and its proof concept and then focuses on aspects relevant for industrial application. We briefly summarize practical experience and give an overview of recent CompCert development aiming at industrial usage. CompCert's intended use is the compilation of life-critical and mission-critical software meeting high levels of assurance. In this context tool qualification is of paramount importance. We summarize the confidence argument of CompCert and give an overview of relevant qualification strategies.

1 Introduction

Modern compilers are highly complex software systems that try to find a balance between various conflicting goals, like minimal size or minimal execution time of the generated code, maximum compilation speed, maximum retargetability, etc. The code generation process itself is a collection of complex transformation steps, many of which are essentially NP-complete, as e.g., the standard backend phases instruction selection, instruction scheduling, or register allocation. The quality of a compiler typically is rated on the efficiency of the generated code whereas its price should not be too high. In consequence compilers have to be efficiently developed, their structure is complex, they contain many highly tuned and sophisticated algorithms – and they can contain bugs. Studies like [12, 5] and [14] have found numerous bugs in all investigated open source and commercial compilers, including compiler crashes and miscompilation issues. Miscompilation means that the compiler silently generates incorrect machine code from a correct source program. Such wrong-code errors can be detected in the normal software testing stage which, however, does typically not 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 DO-178B/C, ISO-26262, or IEC-61508 require to identify potential functional and non-functional 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 not only true for source code review but also for formal, tool-assisted verification methods such as static analyzers, 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.

The first attempts to formally prove the correctness of a compiler date back to the 1960's [10]. Since 2015, with the CompCert compiler, the first formally verified optimizing C compiler is commercially 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 and contributes to meeting the highest levels of software assurance. 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 automati-

cally hold as well for the generated executable.

Usage of CompCert offers multiple benefits. First, the cost of finding and fixing compiler bugs and shipping the patch to customers can be avoided. The testing effort required to ascertain software properties at the binary executable level can be reduced. Whereas in the past for highly critical applications (e.g., according to DO-178B Level A) compiler optimizations were often completely switched off, using optimized code now becomes feasible.

The article is structured as follows: in Sec. 2 we give an overview of the CompCert design and illustrate its proof concept in Sec. 3. Sec. 4 summarizes experimental results and practical experience obtained with the CompCert compiler. Specific developments required for industrial application of CompCert are discussed in Sec. 5. Sec. 6 summarizes the confidence argument for CompCert and discusses tool qualification strategies, Sec. 7 concludes.

2 CompCert Structure

Like other compilers, CompCert is structured as a pipeline of compilation passes, depicted in Figure 1 along with the intermediate languages involved. The 20 passes bridge the gap between C source files and object code, going through 11 intermediate languages. The passes can be grouped in 4 successive phases, described next.

Parsing Phase 1 performs preprocessing (using an off-the-shelf preprocessor such as that of GCC), tokenization and parsing into an ambiguous abstract syntax tree (AST), and type-checking and scope resolution, obtaining a precise, unambiguous AST and producing error and warning messages as appropriate. The LR(1) parser is automatically generated from the grammar of the C language by the Menhir parser generator, along with a Coq proof of correctness of the parser [9]. Optionally, some features of C that are not handled by the verified front-end are implemented by source-to-source rewriting over the AST. For example, bit-fields in structures are transformed into regular fields plus bit shifting and masking. The subset of the C language handled here is very large, including all of MISRA-C 2004 [11] and almost all of ISO C99 [8], with the exceptions of variable-length arrays and unstructured, non-Misra `switch` statements (e.g. Duff's device).

C front-end compiler The second phase first rechecks the types inferred for expressions, then determines an evaluation order among the several permitted by the C standard. This is achieved by pulling side effects (assignments, function calls) outside of expressions, turning them into independent statements. Then, local variables of scalar types whose addresses are never taken (using the `&` operator) are identified and turned into temporary variables; all other local variables are allocated in the stack frame. Finally, all type-dependent behaviors of C (overloaded arithmetic operators, implicit conversions, layout of data structures) are made explicit through the insertion of explicit conversions and address computations. The front-end phase outputs Cminor code. Cminor is a simple, untyped intermediate language featuring both structured (`if`/`else`, loops) and unstructured control (`goto`).

Back-end compiler This third phase comprises 12 of the passes of CompCert, including all optimizations and most dependencies on the target architecture. It bridges the gap between the output of the front-end and the assembly code by progressively refining control (from structured control to control-flow graphs to labels and jumps) and function-local data (from temporary variables to hardware registers and stack frame slots). The most important optimization performed is register allocation, which uses the sophisticated Iterated Register Coalescing algorithm [7]. Other optimizations include function inlining, instruction selection, constant propagation, common subexpression elimination (CSE), and redundancy elimination. These optimizations implement several strategies to eliminate computations that are useless or redundant, or to turn them into equivalent but cheaper instruction sequences. Loop optimizations and instruction scheduling optimizations are not implemented yet.

Optimization passes exploit the results of two intraprocedural static analyses: a forward "value" analysis, that infers variation intervals for integer variables, known values for floating-point variables, and nonaliasing information for pointer variables; and a backward "neededness" analysis, generalizing liveness analysis by identifying the bits of a variable or memory location that do not contribute to the final results of a function.

Optimizations and static analyses are performed over the RTL intermediate representation, consisting of control-flow graphs of "three-address" machine-like instructions. RTL code is not in Single Static Assignment (SSA) form. Consequently, static analyses are more costly and optimizations slightly less aggressive than what is possible with SSA-based compilation algorithms. The reason for not using SSA is that, at the beginning of the CompCert project, the semantic properties of SSA and the soundness arguments for SSA-based algorithms were poorly understood. The recent work of Demange *et al.* [2] and Zhao *et al.* [15] provides formal foundations for SSA-based optimizations.

Assembling The final phase of CompCert takes the AST for assembly language produced by the back-end, prints it in concrete assembly syntax, adds DWARF debugging information coming from the parser (cf. Sec. 5),

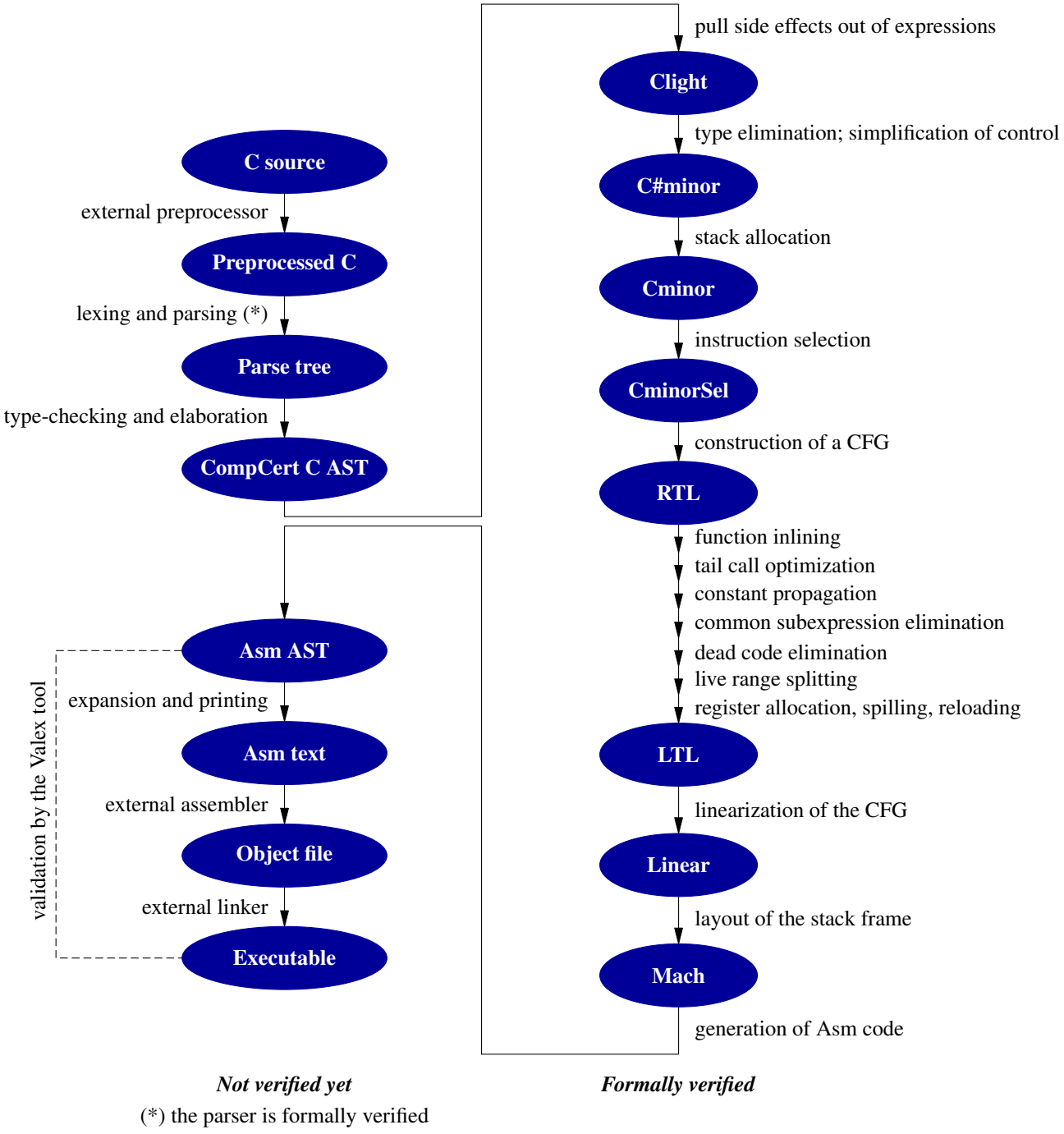

Figure 1: General structure of the CompCert C compiler

and calls into an off-the-shelf assembler and linker to produce object files and executable files. To improve confidence, CompCert provides an independent tool that re-checks the ELF executable file produced by the linker against the assembly language AST produced by the back-end.

3 The CompCert Proof

The CompCert front-end and back-end compilation passes are all formally proved to be free of miscompilation errors; as a consequence, so is their composition. The property that is formally verified is *semantic preservation* between the input code and output code of every pass. To state this property with mathematical precision, we give formal semantics for every source, intermediate and target language, from C to assembly. These semantics associate to each program the set of all its possible behaviors. Behaviors indicate whether the program terminates (normally by exiting or abnormally by causing a run-time error such as dereferencing the null pointer) or runs forever. Behaviors also contain a trace of all observable input/output actions performed by the program, such as system calls and accesses to "volatile" memory areas that could correspond to a memory-mapped I/O device. Arithmetic operations and non-volatile memory accesses are not observable.

Technically, the semantics of the various languages are specified in small-step operational style as labeled transition systems (LTS). A LTS is a mathematical relation *current state* $\stackrel{trace}{\longrightarrow}$ *next state* that describes one step of execution of the program and its effect on the program state. For assembly-style languages, the state of the program comprises the values of processor registers and the contents of memory. The step of computation is the execution of the instruction pointed to by the program counter (PC) register, which updates the contents of registers and possibly of memory. For higher-level languages such as C, states have a richer structure, including not just memory contents and an abstract program point designating the statement or expression to execute next, but also environments mapping variables to memory locations, as well as an abstraction of the stack of function calls.

A generic construction defines the observable behaviors from these transition systems, by iterating transitions from an initial state (the initial call to the `main` function): $S_0 \stackrel{t_1}{\rightarrow} S_1 \stackrel{t_2}{\rightarrow} \cdots \stackrel{t_n}{\rightarrow} S_n$. Such sequences of transitions can stop on a state S_n from which no transition is possible. This describes a terminating execution, where the program terminates either normally (on returning from the `main` function) or on a run-time error (e.g. dereferencing the null pointer, or dividing by zero). Alternatively, an infinite sequence of transitions describes a program execution that runs forever. In both cases, the concatenation of the traces $t_1.t_2\ldots$ describes the I/O actions performed. Several behaviors are possible for the same program if non-determinism is involved. This can be internal non-determinism (e.g. multiple possible evaluation orders in C) or external non-determinism (e.g. reading from a memory-mapped device can produce multiple results depending on I/O behaviors).

To a first approximation, a compiler preserves semantics if the generated code has exactly the same set of observable behaviors as the source code (same termination properties, same I/O actions). This first approximation fails to account for two important degrees of freedom left to the compiler. First, the source program can have several possible behaviors: this is the case for C, which permits several evaluation orders for expressions. A compiler is allowed to reduce this non-determinism by picking one specific evaluation order. For example, consider the following C program:

```
#include <stdio.h>
int f() { printf("f"); return 1; }
int g() { printf("g"); return 2; }
int main() { return f() + g(); }
```

According to the C semantics, two behaviors are allowed, producing `fg` or `gf` as output, depending on whether the call to `f` or the call to `g` occurs first. CompCert chooses to call `g` first, hence the compiled code has only one behavior, with `gf` as output.

Second, a C compiler can "optimize away" run-time errors present in the source code, replacing them by any behavior of its choice. (This is the essence of the notion of "undefined behavior" in the ISO C standards.) Consider an out-of-bounds array access:

```
int main(void)
{ int t[2];
  t[2] = 1;  // out of bounds
  return 0;
}
```

This is undefined behavior according to ISO C, and a run-time error according to the formal semantics of CompCert C. The generated assembly code does not check array bounds and therefore writes 1 in a stack location. This location can be padding, in which case the compiled program terminates normally, or can contain the return address for "main", smashing the stack and causing execution to continue at PC 1, with unpredictable effects. Finally, an optimizing compiler like CompCert can notice that the assignment to `t[2]` is useless (the `t` array is not used afterwards) and remove it from the generated code, causing the compiled program to terminate normally.

To address the two degrees of flexibility mentioned above, CompCert's formal verification uses the following definition of semantic preservation, viewed as a refinement over observable behaviors:

> If the compiler produces compiled code *C* from source code *S*, without reporting compile-time errors, then every observable

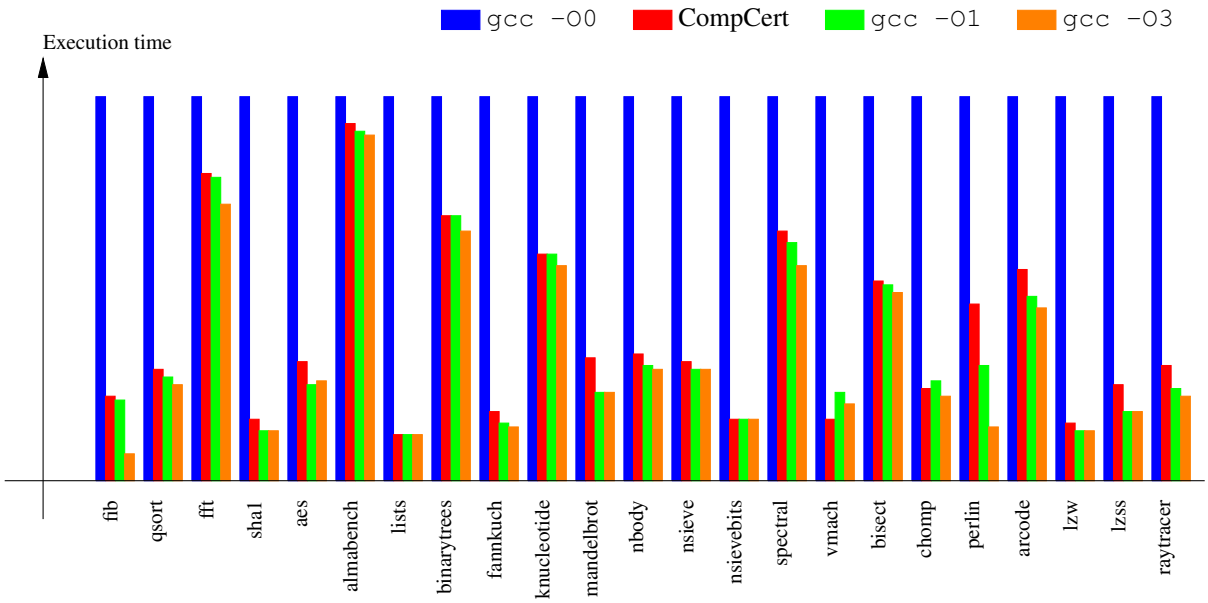

Figure 2: Performance of CompCert-generated code relative to GCC 4.1.2-generated code on a Power7 processor. Shorter is better. The baseline, in blue, is GCC without optimizations. CompCert is in red.

> behavior of *C* is either identical to an allowed behavior of *S*, or improves over such an allowed behavior of *S* by replacing undefined behaviors with more defined behaviors.

In the case of CompCert, the property above is a corollary of a stronger property, called a simulation diagram, that relates the transitions that *C* can make with those that *S* can make. First, 15 such simulation diagrams are proved independently, one for each pass of the front-end and back-end compilers. Then, the diagrams are composed together, establishing semantic preservation for the whole compiler.

The proofs are very large, owing to the many passes and the many cases to be considered – too large to be carried using pencil and paper. We therefore use machine assistance in the form of the Coq proof assistant. Coq gives us means to write precise, unambiguous specifications; conduct proofs in interaction with the tool; and automatically re-check the proofs for soundness and completeness. We therefore achieve very high levels of confidence in the proof. At 100 000 lines of Coq and 6 person-years of effort, CompCert's proof is among the largest ever performed with a proof assistant.

4 Practical Experience

CompCert targets the following three architectures: 32-bit PowerPC, ARMv6 and above, and IA32 (i.e. Intel/AMD x86 in 32-bit mode with SSE2 extension). On PowerPC and ARM, the code generated by CompCert runs at least twice as fast as the code generated by GCC without optimizations, and approximately 10% slower than GCC 4 at optimization level 1, 15% slower at optimization level 2 and 20% slower at optimization level 3. These numbers were obtained on the benchmark suite shown in figure 2, along with the performance numbers for a Power7 processor. This suite comprises computational kernels from various application areas: signal processing, physical simulation, 3d graphics, text compression, cryptography. By lack of aggressive loop optimizations, performance is lower on HPC codes involving dense matrix computations. On IA32, due to its paucity of registers and its specific calling conventions, CompCert is approximately 20% slower than GCC 4 at optimization level 1.

CompCert provides a general mechanism to attach free-form annotations (text messages possibly mentioning the values of variables) to C program points, and have these annotations transported throughout compilation, all the way to the generated assembly code, where the variable names are expressed in terms of machine code addresses and machine registers. Apart from improving traceability this source annotation mechanism enables WCET tools to compute more precise WCET bounds. Indeed, WCET tools like aiT [6] operate directly on the executable code, but they sometimes require programmers to provide additional information (e.g., the bound of a while loop) that cannot easily be reconstructed from the machine code alone. When using CompCert, such information can be safely extracted from annotations inserted at the source code level. A tool automating this task was developed by Airbus: it generates a machine-level annotation file usable by the aiT WCET analyzer. Compiling a whole flight control software from Airbus (about 4 MB of assembly code) with CompCert resulted in significantly improved performance in terms of WCET bounds and code size [3].

5 Industrial Application

An industrial application of CompCert is not only attractive because of the high confidence in the correctness of the compilation process. As mentioned in Sec. 1 it opens up the possibility to use optimized code even in highly critical avionics projects, hence enabling higher software performance. Furthermore, CompCert's annotation mechanism as described in Sec. 4 can contribute to further improving confidence by providing proven traceability information [3].

A prerequisite for industrial use of a compiler is the possibility to debug the program under compilation. While previous versions of CompCert only provided rudimentary debugging support, CompCert now is able to produce debug information in the Dwarf 2 format. It generates debugging information for functions and variables, including information about their type, size, alignment and location. This also includes local variables so that the values of all variables can be inspected during program execution in a debugger. To this end CompCert introduces an additional pass which computes the live ranges of local variables and their locations throughout the live range. Furthermore CompCert keeps track of the lexical scopes in the original C program and creates corresponding Dwarf 2 lexical scopes in the debugging information.

The CompCert sources can be downloaded from Inria[1] free of charge; the current state of the development can be viewed on Github [2]. In addition, a revisioned distribution is available from AbsInt, either as a source code download or as pre-compiled binary for Windows and Linux. To create the pre-compiled binary, Coq is executed under Linux to create the OCaml source files and the corresponding correctness proof. This provides a proven-in-use argument for Coq usage since there is a wide community using Coq under Linux/Unix. Furthermore it makes sure the Windows and Linux versions operate on the same sources. Under Windows the OCaml sources are compiled with a native Windows compiler; to execute CompCert no additional libraries (e.g., cygwin or mingw) are needed. The package also contains pre-configured setup files for the compiler driver to control the cooperation between the CompCert executable and the external cross compiler required for preprocessing, assembling and linking.

Translation Validation. Currently the verified part of the compile tool chain ends at the generated assembly code. In order to bridge this gap we have developed a tool for automatic translation validation, called *Valex*, which validates the assembling and linking stages a posteriori.

Valex checks the correctness of the assembling and linking of a statically and fully linked executable file against the internal abstract assembly representation produced by CompCert. In order to use Valex, the C source files for the program must be compiled by CompCert and the command-line option `-sdump` must be specified. This option instructs CompCert to serialize the internal abstract assembly representation in JSON [4] format.

The generated `.json`-files as well as the linked executable are then passed as arguments to the Valex tool. The main goal is to verify that every function defined in a C source file compiled by CompCert and not optimized away by it can be found in the linked executable and that its disassembled machine instructions match the abstract assembly code. To that end, after parsing the abstract assembly code Valex extracts the symbol table and all sections from the linked executable. Then the functions contained in the abstract assembly code are disassembled. Extraction and disassembling is done by two invocations of exec2crl, the executable reader of aiT [1].

exec2crl tries to decode all symbols in the linked executable which have the same name as a function symbol contained in any of the input `.json`-files. If one of these symbols in the linked executable is not a function, exec2crl reports a warning that it cannot decode the given symbol. The decoded control-flow graph is linearized to a sequential list of its instructions sorted by their address.

The main stages of Valex deal with function and variable usage. For every function in the abstract assembly code, Valex matches the instructions in the abstract assembly code against the instructions contained in the linked executable. Valex supports both the cases that an abstract assembly instruction directly corresponds to one machine instruction, and that it corresponds to a sequence of machine instructions. It checks whether the arguments of the instructions are equivalent and the intended instruction mnemonic is used.

For every variable defined in a C source file compiled by CompCert Valex checks whether the corresponding symbol can be found in the symbol table. It also checks that the corresponding size and initialization data is contained in the linked executable and matches the initialization data in the abstract assembly. Valex reports an error if one of these checks fails.

In a further stage Valex reconstructs the mapping from symbolic names and labels to machine addresses computed by the linker, checks that there is an address for every symbolic name and label and ensures that all occurrences are always mapped to the same address.

Additionally, Valex tests for variables and functions whether they are placed in their corresponding sections, which are either the default sections, or the sections specified by the user in the C files using `#pragma` section. When using the GCC tool chain, Valex also checks whether uninitialized variables that were placed into one of CompCert's internal sections `Data`, `Small Data`,

[1] http://compcert.inria.fr/download.html
[2] https://github.com/AbsInt/CompCert

`Const` or `Small Const` are contained in the `.bss` or `.sbss` section of the executable.

Currently Valex can check linked PowerPC executables that have been produced from C source code by the CompCert C compiler using the Diab assembler and linker from Wind River Systems, or the GCC tool chain (version 4.8, together with GNU binutils 2.24).

6 The Confidence Argument

As described in Sec. 3 all of CompCert's front-end and back-end compilation passes are formally proved to be free of miscompilation errors. These formal proofs bring strong confidence in the correctness of the front-end and back-end parts of CompCert. These parts include all optimizations – which are particularly difficult to qualify by traditional methods – and most code generation algorithms.

The formal proofs do not cover the following aspects:

1. Correctness of the specifications used for the formal proof, i.e. the formal semantics of C and assembly.
2. The parsing phase, i.e. the transformation from the input C program to CompCert's abstract syntax.
3. The assembling and linking phase.

Those aspects can be handled well by traditional qualification methods, i.e. via a validation suite, to complement the formal proofs. A validation suite for CompCert is currently in development and will be available from AbsInt.

Especially the parsing phase (item 2) can be seen as a straightforward code generation pass which does not include any optimizations and only performs local transformations. Since the internal complexity of this stage is low, systematic testing provides good confidence. CompCert can print the result of parsing in concrete C syntax, facilitating comparison with the C source.

However, it is possible to provide additional confidence beyond the significance of the validation suite, in particular for items 1 and 3. CompCert provides a reference interpreter, proved correct in Coq, that can be used to systematically test the C semantics on which the compiler operates. Likewise, the Valex validator described in Sec. 5 provides confidence in the correctness of the assembling and linking phase. It performs translation validation of the generated code which is a widely accepted validation method [13].

At the highest assurance levels, qualification arguments may have to be provided for the tools that produce the executable CompCert compiler from its verified sources, namely the "extraction" mechanism of Coq, which produces OCaml code from the Coq development, combined with the OCaml compiler. We are currently experimenting with an alternate execution path for CompCert that relies on Coq's built-in program execution facilities, bypassing extraction and OCaml compilation. This alternate path runs CompCert much more slowly than the normal path, but fast enough that it can be used as a validator for selected runs of normal CompCert executions.

In summary, CompCert provides unprecedented confidence in the correctness of the compilation phase: the 'normal' level of confidence is reached by providing a validation suite, which is currently accepted best practice; the formal proofs provide much higher levels of confidence concerning the correctness of optimizations and code generation strategies; finally, the Valex translation validator provides additional confidence in the correctness of the assembling and linking stages.

7 Conclusions

CompCert is a formally verified optimizing C compiler: the executable code it produces is proved to behave exactly as specified by the semantics of the source C program. Experimental studies and practical experience demonstrate that it generates efficient and compact code. Further requirements for industrial application, notably the availability of debug information, and support for Linux and Windows platforms have been established. Explicit traceability mechanisms enable a seamless mapping from source code properties to properties of the executable object code. We have summarized the confidence argument for CompCert, which makes it uniquely well-suited for highly critical applications.

References

[1] AbsInt GmbH, Saarbrücken, Germany. *AbsInt Advanced Analyzer for PowerPC*, October 2015. User Documentation.

[2] G. Barthe, D. Demange, and D. Pichardie. Formal verification of an SSA-based middle-end for CompCert. *ACM Trans. Program. Lang. Syst.*, 36(1):4, 2014.

[3] R. Bedin França, S. Blazy, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS 2012: Embedded Real Time Software and Systems*, 2012.

[4] ECMA International. Standard ECMA-404 The JSON Data Interchange Format. Technical report, ECMA International, Oct. 2013.

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

[6] C. Ferdinand and R. Heckmann. aiT: Worst-Case Execution Time Prediction by Static Programm Analysis. In R. Jacquart, editor, *Building the Information Society. IFIP 18th World Computer Congress*, pages 377–384. Kluwer, 2004.

[7] L. George and A. W. Appel. Iterated register coalescing. *ACM Trans. Prog. Lang. Syst.*, 18(3):300–324, 1996.

[8] ISO. International standard ISO/IEC 9899:1999, Programming languages – C, 1999.

[9] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *ESOP 2012: 21st European Symposium on Programming*, volume 7211 of *LNCS*, pages 397–416. Springer, 2012.

[10] J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Mathematical Aspects of Computer Science*, volume 19, pages 33–41, 1967.

[11] Motor Industry Software Reliability Association. MISRA-C: 2004 – Guidelines for the use of the C language in critical systems, 2004.

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

[13] A. Pnueli, M. Siegel, and E. Singerman. Translation validation. In *TACAS'98: Tools and Algorithms for Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

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

[15] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal verification of SSA-based optimizations for LLVM. In *PLDI'13: ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 175–186. ACM, 2013.