CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler

Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, Sandrine Blazy

CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler

Daniel Kästner[3] Jörg Barrho[4] Ulrich Wünsche[4] Marc Schlickling[4]
Bernhard Schommer[5] Michael Schmidt[3] Christian Ferdinand[3]
Xavier Leroy[1] Sandrine Blazy[2]

1: Inria Paris, 2 rue Simone Iff, 75589 Paris, 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
4: MTU Friedrichshafen GmbH, Maybachplatz 1, D-88048 Friedrichshafen, Germany
5: Saarland University, Saarland Informatics Campus, 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 use of CompCert to gain certification credits for a highly safety-critical industry application, certified according to IEC 60880 [7]. We will briefly introduce the target application, illustrate the process of changing the existing compiler infrastructure to CompCert, and discuss performance characteristics. The main part focuses on the tool qualification strategy, in particular on how to take advantage of the formal correctness proof in the certification process.

1 Introduction

A compiler translates the source code written in a given programming language into executable object code of the target processor. Due to the complexity of the code generation and optimization process compilers may contain bugs. In fact, studies like [23, 5] and [25] 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.

In 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 identify and track down. Furthermore 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. In consequence, 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.

Since 2015 the CompCert compiler has been commercially available. CompCert 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. CompCert is the first formally verified compiler on the market; it provides an unprecedented level of confidence in the correctness of the compilation process. In general, 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 since the correctness proof of CompCert C guarantees that all safety properties verified on the source code automatically hold as well for the generated executable. 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.

In [19] we have given an overview of the design and the proof concept of CompCert and have presented an evaluation of its performance on the well-known SPEC benchmarks. In this article we report on practical experience with replacing a legacy compiler by CompCert for a highly critical control system from MTU in the nu-

clear power domain.

The article is structured as follows: in Sec. 2 we give an overview of the MTU application for which CompCert is used; Sec. 3 describes the relevant considerations for applying a traditional non-verified compiler. In Sec. 4 we briefly summarize the CompCert design and its proof concept. Sec. 5 describes the integration of CompCert into the development process, and the performance gains observed. The tool qualification strategy is detailed in Sec. 6, Sec. 7 concludes.

2 The Application

MTU develops diesel engines that are deployed in civil nuclear power plants as drivers for emergency generators to generate electrical power. Such engines are available to the market diversely as either common rail or fuel rack controlled engines with capabilities to produce up to 7 MW electrical power per unit.

In case of failures in the electrical grid of a nuclear power plant one or more of these units are requested to provide power to support the capability to control the nuclear plant core and cooling systems. It is obvious that the functional contribution may be mission critical to the overall plant.

The engines are controlled by an MTU-developed digital engine control unit (ECU). This ECU performs only safety functions and in particular maintains the safe state requested by the plant operator. This safe state ensures that the engine stands still if required and is controlled to maintain the demanded engine speed if required.

Software decomposition The software of the ECU runs on top of a handwritten runtime environment, written in assembler, specific to the controller in use. The application consists of handwritten C-code and generated C-code derived from SCADE models.

The handwritten C-code implements a scheduler, a hardware abstraction layer, and self-supervision capabilities. The hardware abstraction layer polls physical sensor inputs, controls hardware actuators, and provides hardware related self supervision mechanisms which must not interfere with the two former objectives in fixed timing intervals. Such fixed intervals must be small enough to acquire all relevant events and to maintain sensor acquisitioning sampling theorems.

The scheduler provides safe data and control flow interfacing between the concurrent hardware access thread and the main control loop. Such interfacing limits the amount of required race condition considerations and allows for maintaining safe timing constraints of the threads. Based on safe over approximations of timing envelopes it is possible to prove that all scheduling constraints are always maintained.

The SCADE model provides the engine controller algorithms. The monolithic model strictly follows the synchronous paradigm by separating input acquisition, processing and generating output. The entire model execution is provided in SCADE which is a prerequisite to make further statements on the model integrity.

Development constraints Software and development process comply with the international standards IEC60880 [7] and IEC61508:2010, part 3 (SCL3 for software) [8].

C was chosen as programming language because of the abundant availability of translators for the targeted PowerPC architecture. Code generators from model driven approaches to C are well introduced and the SCADE generator is validated to translate correctly to a defined language subset.

C subset All C-code is produced in a subset of ISO/IEC9899:1999 [9]. Its capability is sufficient for all of the outlined application requirements. This version of the standard is considered so widely used that the standard and its deficiencies are well understood and compilers are more likely to fully comply.

Emphasis is put on the objective to enhance robustness, to provide exactly one method to solve a problem and to avoid potentially error-prone constructs. The MISRA:2004 [22] standard is a good starting point for choosing such a language subset. In addition continuous research on actual and potential coding defects has been considered. Lastly the subset is formed by complementing cultural development among users and testers of the application in question in a structured process.

With each programming project an assessment of perceived risks regarding frequency, potential consequences and chances of detection is carried out. During development, continuous discussions are encouraged to support risk consciousness.

All of these risk considerations are condensed into a set of in-house coding guidelines also reflecting the current project team's language proficiency.

Data types are basically restricted to the use of integer arithmetics with as few type conversions as possible. Thus compiler behavior is as explicit as possible mending some of the inherent type unsafety of C. Enums, unions and bit fields are not part of the language subset. The language subset is also designed to be well covered by automatic checking tools. The sound static runtime error analyzer Astrée [19, 16] also includes a coding guideline checker, called RuleChecker, which is suitable for the subset chosen.

For the defined specific rule set it provides a coding guideline coverage of more than 85 %. The remaining 15 % are inevitably attributed to such objectives requiring human involvement as to avoid tricky programming, to choose understandable identifier names and to provide helpful comments.

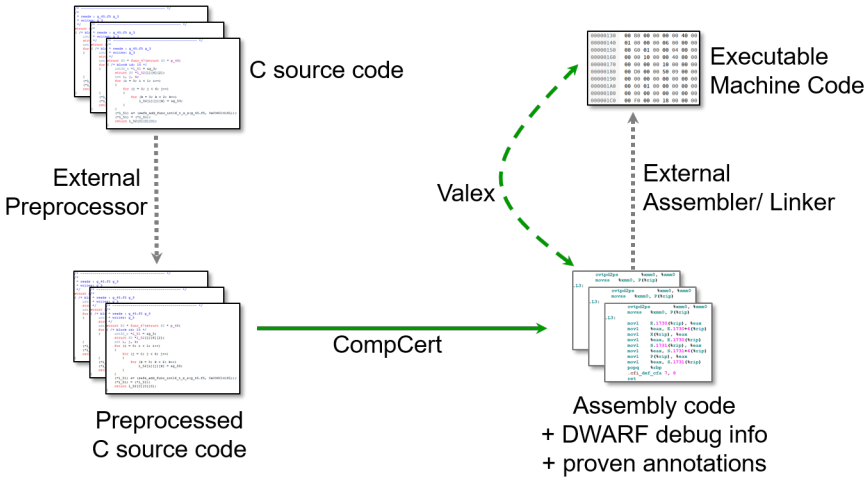

Figure 1: CompCert Workflow

3 The Past: Using a Non-Verified Compiler

Compiling source code which becomes part of safety software in production use is inherently flagged as a critical task. For such critical tasks a tool must be qualified as suitable by fulfilling a number of criteria defined by the user. MTU only uses critical tools in safety applications if such a tool has been developed within a structured process. It must provide sufficient evidence for reliable operation and user experience must have a positive record. MTU's tool qualification strategy is depicted in Fig. 2.

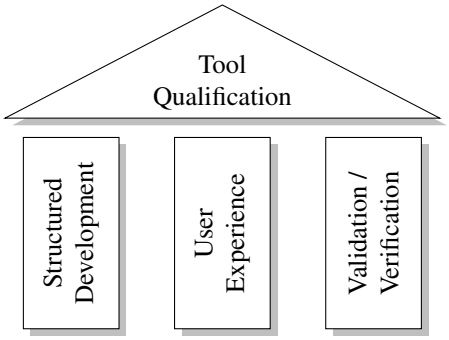

Figure 2: MTU tool qualification strategy

Historically MTU has used a traditional commercially available C-compiler well proven in use. Use of this compiler requires some maintenance effort due to the sporadic appearance of new bugs. Each of these bugs requires evaluation and eventually code changes and changes of code review checklists for fully standard compliant source code.

When such a proven in use compiler is removed from standard supplier support there are two options. The supplier may offer the service to check if bugs in later compiler versions already existed in the used version. However the supplier may charge substantial fees for such service. Alternatively the user may decide to qualify a newer compiler version with a commercial validation suite. This also induces substantial effort and external costs. Neither of these alternatives is satisfactory.

4 The CompCert Compiler

In the following we will give a brief overview of the design and proof concept of CompCert; more details can be found in [19]. Fig. 1 shows the CompCert-based workflow. The input to the compilation process is a set of C source and header files. CompCert itself focuses on the task of compilation and includes neither preprocessor, assembler, nor linker. Therefore it has to be used in combination with a legacy compiler tool chain. Since preprocessing, assembling and linking are well-established stages there are no particular tool chain requirements.

While early versions of CompCert were limited to single-file inputs, CompCert now also supports separate compilation [14]. It reads the set of preprocessed C files emitted by the legacy preprocessor, performs a series of code generation and optimization steps and emits a set of assembly files enhanced by debug information.

CompCert generates DWARF2 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 a dedicated pass which computes the live ranges of local variables and their locations throughout the live range.

The generated assembly code can contain formal CompCert annotations which can be inserted at the C code level and are carried throughout the code generation process. This way, traceability information, or semantic information to be passed to other tools can be transported to the machine code level. Since they are fully covered by the CompCert proof the information is

reliable and provides proven links between the machine code and the source code level.

After assembling and linking by the legacy tool chain the final executable code is produced. To increase confidence in the assembling and linking stages CompCert provides a tool for translation validation, called Valex, which performs equivalence checks between assembly and executable code (cf. Sec. 4.4).

4.1 Design Overview

CompCert is structured as a pipeline of 20 compilation passes that bridge the gap between C source files and object code, going through 11 intermediate languages. The passes can be grouped in 4 successive phases:

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 [11].

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. Implicit type conversions, operator overloading, address computations, and other type-dependent behaviors are made explicit; loops are simplified. 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. The most important optimization performed is register allocation, which uses the sophisticated Iterated Register Coalescing algorithm [6]. 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.

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, 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, called *Valex* (cf. Sec. 6), that re-checks the ELF executable file produced by the linker against the assembly language AST produced by the back-end.

4.2 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 runtime 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.

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. Second, a C compiler can "optimize away" runtime 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.) As an example 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 runtime 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:

Definition 1 (Semantic preservation) *If the compiler produces compiled code C from source code S, without reporting compile-time errors, then every observable 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.*

The semantic preservation property 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, the 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.

4.3 Proving the Absence of Runtime Errors

In safety-critical systems, the use of dynamic memory allocation and recursions is typically forbidden or only used in limited ways. This simplifies the task of static analysis such that for safety-critical embedded systems it is possible to formally prove the absence of runtime errors, or report all potential runtime errors which still exist in the program. Such analyzers are based on the theory of abstract interpretation [4], a mathematically rigorous formalism providing a semantics-based methodology for static program analysis. Abstract interpretation supports formal correctness proofs: it can be proved that an analysis will terminate and that it is *sound*, i.e., that it computes an over-approximation of the concrete semantics. If no potential error is signaled, definitely no runtime error can occur: there are no false negatives. If a potential error is reported, the analyzer cannot exclude that there is a concrete program execution triggering the error. If there is no such execution, this is a false alarm (false positive). This imprecision is on the safe side: it can never happen that there is a runtime error which is not reported.

One example of a sound static runtime error analyzer is the Astrée analyzer [20, 15]. It reports program defects caused by unspecified and undefined behaviors according to the C norm (ISO/IEC 9899:1999 (E)) [9], program defects caused by invalid concurrent behavior, violations of user-specified programming guidelines, and computes program properties relevant for functional safety. Users are notified about: integer/floating-point division by zero, out-of-bounds array indexing, erroneous pointer manipulation and dereferencing (buffer overflows, null pointer dereferencing, dangling pointers, etc.), data races, lock/unlock problems, deadlocks, integer and floating-point arithmetic overflows, read accesses to uninitialized variables, unreachable code, nonterminating loops, violations of optional user-defined static assertions. Astrée also provides a module for checking coding rules, called *RuleChecker*, which supports various coding guidelines (MISRA C:2004 [22], MISRA C:2012 [21], ISO/IEC TS 17961 [10], SEI CERT C [2, 3], CWE [24]), computes code metrics and checks code metric thresholds. RuleChecker is also available as a standalone product, but when used in combination with Astrée it can access the results of the sound static runtime analysis and, hence, can achieve zero false negatives even on semantic rules.

4.4 Translation Validation

Currently the verified part of the compilation 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.

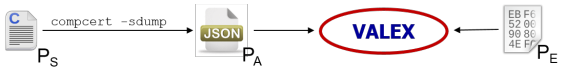

Figure 3: Translation Validation with Valex

Valex checks the correctness of the assembling and linking of a statically and fully linked executable file P_E against the internal abstract assembly representation P_A produced by CompCert from the source C program P_S. The internal abstract assembly as well as the linked executable are 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 and StackAnalyzer [1]. Apart from matching the instructions in the abstract assembly code against the instructions contained in the linked executable Valex also checks whether symbols are used consistently, whether variable size and initialization data correspond and whether variables are placed in the right sections in 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).

5 Integration and Performance

Integration The ECU control software uses a limited set of timing interrupts which does not impair worst-case execution time estimations. The traditional compiler accepts pragma indications to flag C-functions so they can be called immediately from an interrupt vector. The compiler then adds code for saving the system state and more registers than used in a standard PowerPC EABI function call.

CompCert does not accept this compiler-dependent pragma nor inline assembly so the user must hand-code the mechanism outlined in the previous paragraph in assembler language in separate assembly files. Such assembler code can be placed in the runtime environment module. Some system state recovery contained in a fallback exception handler is also transferred to the runtime environment.

The strategy of using a minimum sufficient subset as discussed in Sec. 2 above is fully confirmed since only one related change to the source code was necessary. For more than five years CompCert has fully covered the chosen range of constructs even during earlier phases of its development.

Behaviors undefined according to the C semantics are not covered by the formal correctness proof of CompCert. Only code that exhibits no numeric overflows, division by zero, invalid memory accesses or any other undefined behavior can possibly be functionally correct. The sound abstract interpretation based analyzer Astrée can prove the absence of runtime errors including any undefined behaviors [18, 19]. Therefore we use Astrée to complement the formal correctness argument of CompCert.

Further minor modifications were necessary to adapt the build process to the CompCert compiler options. Also the linker control file required some changes since CompCert allocates memory segments differently from some traditional popular compilers.

In the final step an MTU specific flashing tool assigns code, constant data as well as initialized and non-initialized data as required by the C runtime environment specific to the target architecture.

Testability Testing functional behaviour on the target platform can be tedious. Potentially concurrent software interacts with hardware which does not necessarily behave according to the synchronous paradigm. The hardware in turn interacts with the noise charged physical environment. In addition some of that interaction only works properly under hard real time restrictions. Thus typical module or software tests in the target environment suffer from the necessity to impose severe restrictions on the behaviors expected in reality.

It is thus desirable to test software components reaching a maximum coverage of real world interaction noise.

If such components are specified to expose defined complete and non contradicting behaviour on their boundaries and are written as generically as possible, abstract testing comes into reach. Generic behaviour does not depend on underlying processor properties such as endianness and hardware register allocation. On the compiler side it does not depend on compiler specific or undefined behaviour. Coding guidelines and architectural constraints may ensure compliance with such rules.

If software artifacts comply with these constraints they may be tested independently from hardware and specific compilation tool chain. CompCert is available for ARM, x86 and PowerPC architectures so that properties acquired on one platform hold on the other.

Code Performance The code generated by CompCert was subjected to the Valex tool and shows no indications of incompliance. The generated code was integrated into the target hardware and extensively tested in a simulated synthetic environment which is a precondition to using the integrated system on a real engine. If simulator test and engine test are passed they jointly provide behavioral validation coverage of every aspect of the functional system requirements.

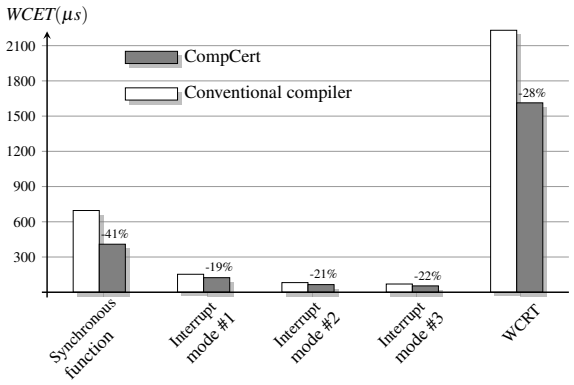

Figure 4: WCET estimates for MTU application

All building processes were completed successfully; all functional tests passed. Thus these tests – on an admittedly minimized and robust language subset – exposed no indication of compiler flaws.

To assess the performance of the CompCert compiler further we have investigated the size and the worst-case execution time of the generated code.

To determine the memory consumption by code and data segments we have analyzed the generated binary file. Compared to the conventional compiler the code segment in the executable generated by CompCert is slightly smaller. The size of the data segment size is almost identical in both cases. These observations are consistent with our expectations since in CompCert we have used more aggressive optimization settings. The traditional compiler was configured not to use any opti-

mization to ensure traceability and to reduce functional risks introduced by the compiler itself during the optimization stage.

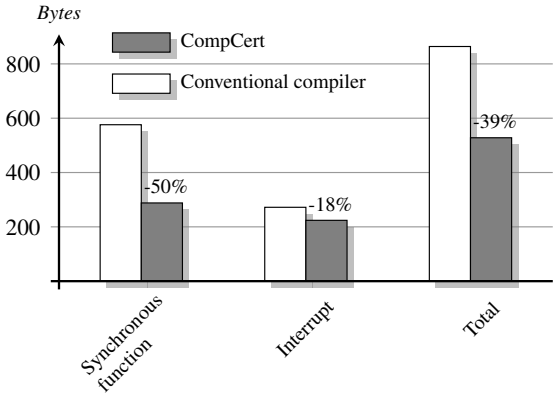

Figure 5: Worst-case stack usage for MTU application

With the verified compiler CompCert at hand the design decision was made to lift this restriction. CompCert performs register allocation to access data from registers and minimizes memory accesses. In addition, as opposed to the traditional compiler it accesses memory using small data areas. That mechanism lets two registers constantly reference base memory addresses so that address references require two PowerPC assembler instructions instead of three as before.

The maximum execution time for one computational cycle is assessed with the static WCET (worst-case execution time) analysis tool aiT [17]. When configured correctly this tool delivers safe upper execution time bounds. All concurrent threads are mapped into one computation cycle under worst-case conditions. The precise mapping definition is part of the architectural software design on the bare processor.

Analyses are performed on a normal COTS PC, each entry (synchronous function, interrupt) has been analyzed separately. Analysis of timing interrupt is split in several modes, and finally, the WCRT (worst-case response time) for one computational cycle is calculated. The results for the MTU application are shown in Fig. 4. The computed WCET bounds lead to a total processor load which is about 28% smaller with the CompCert-generated code than with the code generated by the conventional compiler. The main reason for this behaviour is the improved memory performance. The result is consistent with our expectations and with previously published CompCert research papers.

We have also determined a safe upper bound of the total stack usage in both scenarios, using the static analyzer StackAnalyzer [13]. The results are shown in Fig. 5. When providing suitable behavioral assumptions about the software to the analyzer the overall stack usage is around 40% smaller with the CompCert-generated code than the code generated by the conventional compiler.

6 Tool Qualification

MTU's qualification strategy is built on three columns, namely providing evidence of a structured tool development, sufficient user experience, and confirmation of reliable operation via validation (cf. Sec. 3 and Fig. 2). This strategy has also been applied to qualify CompCert for use within a highly safety-critical application.

Compilation As described in Sec. 4 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 proof does not cover some elements of the parsing phase, nor the preprocessing, assembling and linking (cf. [19]) for which external tools are used. Therefore we complement the formal proof by applying a publically available validation suite.

The overall qualification strategy for CompCert is depicted in Fig. 6. In contrast to validating the correlation of source files and the resulting fully linked executable file, qualification of the compiler toolchain is split in three phases: traditional testsuite validation, formal verification, and translation validation.

Preprocessor Source-code preprocessing is mandated to a well-used version of gcc. The selected version is validated using a preprocessor testsuite, for which the correlation to the used language subset is manually proven. MTU uses strict coding rules limiting the use of C-language [**?**] constructs to basic constructs known to be widely in use. Also usage of C preprocessing macros is limited by these rules to very basic constructs. The testsuite is tailored to fully cover these demands.

It must be ensured that source files and included header files only use a subset of the features which are validated by the above procedure. This may be accomplished by establishing a suitable checklist and manually applying it to each and every source file.

Effort may however be reduced and the reliability of that process be vastly improved if a coding guideline checker is used. That tool must again be validated to provide alarms for every violation of any required rule.

As described above Astrée includes a code checker, called *RuleChecker*, which analyzes each source file for compliance with a predefined set of rules, including MISRA:2004 [22]. It also provides a Qualification Support Kit and Qualification Software Life Cycle Data reports which facilitate the tool qualification process.

Assembling and Linking Cross-assembling and cross-linking is also done by gcc. To complement the proven-in-use argument and the implicit coverage by the validation suite we use the translation validation tool *Valex* shipped with CompCert which provides

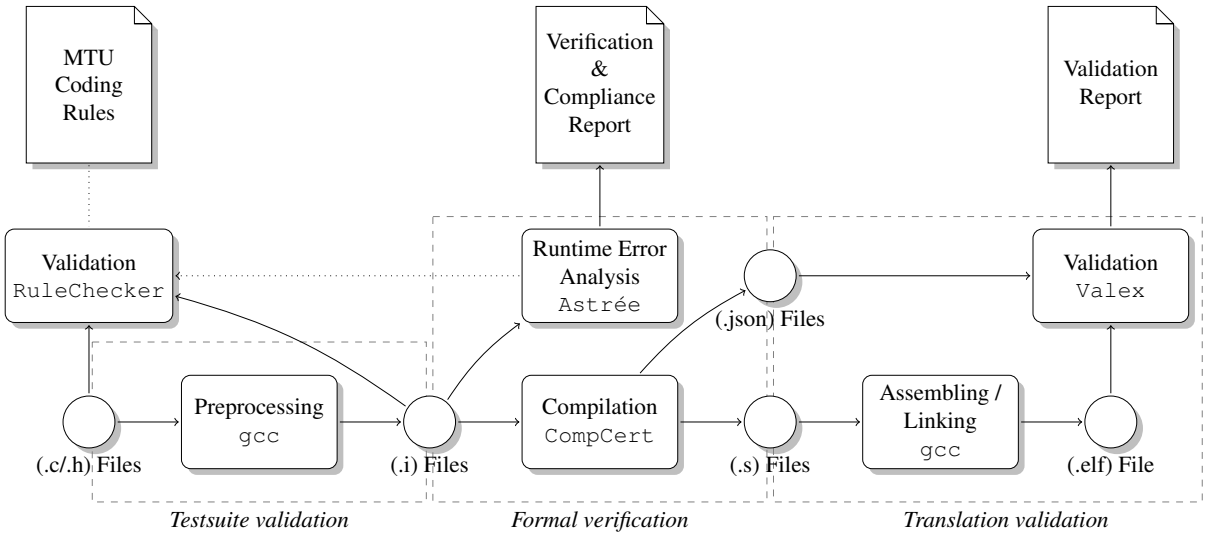

Figure 6: CompCert qualification

additional confidence in the correctness of assembler and linker. Each source file is compiled with CompCert using a dedicated option, s.t. CompCert is instructed to serialize its internal abstract assembly representation in JSON format [12]. The generated `.json`-files as well as the fully linked executable are then passed to the Valex tool. As described in Sec. 4.4 Valex checks the correctness of the assembling and linking of the executable file against the internal abstract assembly representation produced by CompCert.

Tools used in the process of qualifying CompCert, namely Astrée and Valex, are also qualified using the qualification strategy described above. By dividing the qualification of CompCert into steps and applying strict coding rules throughout the development, complexity of compiler qualification tremendously decreases making use of CompCert feasible also within a highly safety-critical industrial application.

7 Conclusion

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. This article reports on practical experience obtained at MTU with replacing a non-verified legacy compiler by CompCert for a highly critical control software of an emergency power generator. We have described the necessary steps to integrate CompCert in the development process, and outlined our tool qualification strategy. The main benefits are higher confidence in the correctness of the generated code, and significantly improved system performance.

References

[1] AbsInt GmbH, Saarbrücken, Germany. *AbsInt Advanced Analyzer for PowerPC*, April 2016. User Documentation.

[2] CERT – Software Engineering Institute. *SEI CERT C Coding Standard – Rules for Developing Safe, Reliable, and Secure Systems*. Carnegie Mellon University, 2016.

[3] CERT – Software Engineering Institute, Carnegie Mellon University. SEI CERT Coding Standards Website.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th POPL*, pages 238–252, Los Angeles, CA, 1977. ACM Press.

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

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

[7] IEC 60880. Nuclear power plants instrumentation and control systems important to safety software aspects for computer-based systems performing category a functions, 2006.

[8] IEC 61508. Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

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

[10] ISO/IEC. Information Technology – Programming Languages, Their Environments and System Software Interfaces – Secure Coding Rules (ISO/IEC TS 17961), Nov 2013.

[11] 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.

[12] The JSON Data Interchange Format. Technical Report Standard ECMA-404 1st Edition / October 2013, ECMA, Oct. 2013.

[13] D. Kästner and C. Ferdinand. Proving the Absence of Stack Overflows. In *SAFECOMP '14: Proceedings of the 33th International Conference on Computer Safety, Reliability and Security*, volume 8666 of *LNCS*, pages 202–213. Springer, September 2014.

[14] D. Kästner, X. Leroy, S. Blazy, B. Schommer, M. Schmidt, and C. Ferdinand. Closing the gap – the formally verified optimizing compiler CompCert. In *SSS'17: Developments in System Safety Engineering: Proceedings of the Twenty-fifth Safety-critical Systems Symposium*, pages 163–180. CreateSpace, 2017.

[15] D. Kästner, A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, A. Schmidt, H. Hille, S. Wilhelm, and C. Ferdinand. Finding All Potential Runtime Errors and Data Races in Automotive Software. In *SAE World Congress 2017*. SAE International, 2017.

[16] D. Kästner, A. Miné, A. Schmidt, H. Hille, L. Mauborgne, S. Wilhelm, X. Rival, J. Feret, P. Cousot, and C. Ferdinand. Finding All Potential Run-Time Errors and Data Races in Automotive Software. In *Proceedings of the SAE World Congress 2017 (SAE Technical Paper)*. SAE International, 2017.

[17] D. Kästner, M. Pister, G. Gebhard, M. Schlickling, and C. Ferdinand. Confidence in Timing. *Safecomp 2013 Workshop: Next Generation of System Assurance Approaches for Safety-Critical Systems (SASSUR)*, September 2013.

[18] D. Kästner, S. Wilhelm, S. Nenova, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Astrée: Proving the Absence of Runtime Errors. *Embedded Real Time Software and Systems Congress ERTS* [2], 2010.

[19] X. Leroy, S. Blazy, D. Kästner, B. Schommer, M. Pister, and C. Ferdinand. CompCert - A Formally Verified Optimizing Compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, Toulouse, France, Jan. 2016. SEE.

[20] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. *Embedded Real Time Software and Systems Congress ERTS*[2], 2016.

[21] MISRA Working Group. *MISRA-C:2012 Guidelines for the use of the C language in critical systems*. MISRA Limited, Mar. 2013.

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

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

[24] The MITRE Corporation. CWE – Common Weakness Enumeration.

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