Closing the Gap – The Formally Verified Optimizing Compiler CompCert

Daniel Kästner[1], Xavier Leroy[2], Sandrine Blazy[3], Bernhard Schommer[4], Michael Schmidt[1], Christian Ferdinand[1]

1: AbsInt GmbH, Saarbrücken, Germany
2: Inria Paris-Rocquencourt, Le Chesnay, France
3: University of Rennes 1 - IRISA, Rennes, France
4: Saarland University, Saarbrücken, Germany

Abstract *CompCert is the first commercially available optimizing compiler that is formally verified, using machine-assisted mathematical proofs, to be free from miscompilation. The executable code it produces is proved to behave exactly as specified by the semantics of the source C program. CompCert's intended use is the compilation of safety-critical and mission-critical software meeting high levels of assurance. This article gives an overview of the design of CompCert and its proof concept, summarizes the resulting confidence argument, and gives an overview of relevant tool qualification strategies. We briefly summarize practical experience and give an overview of recent CompCert developments.*

1 Introduction

Modern compilers are highly complex software systems which contain many highly tuned and sophisticated algorithms; however these can contain bugs. Studies like (NULLSTONE Corporation 2007, Eide and Regehr 2008) and (Yang et al.2011) have found numerous bugs in all investigated open source and commercial compilers, including compiler crashes and miscompilation[1] 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 DO-178B/C, ISO-26262, or IEC-61508 require identification of potential hazards and

[1] Miscompilation means that the compiler silently generates incorrect machine code from a correct source program.

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.

The first attempts to formally prove the correctness of a compiler date back to the 1960's (McCarthy and Painter 1967). 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. 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.

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 compiler optimizations were often completely switched off, using optimized code now becomes feasible.

The paper is structured as follows: in section 2 we give a top-level overview of the CompCert compiler and its tool flow. Section 3 summarizes the main code generation and optimization stages of CompCert and its annotation concept. The formal CompCert proof is outlined in section 4. Section 5 presents the Valex tool for a posteriori validation of assembly and linking phases. Section 6 describes the reference interpreter provided by CompCert for testing and semantic validation purposes. Section 7 summarizes the confidence argument for CompCert and adequate tool qualification strategies. Section 8 summarizes experimental results and practical experience obtained with the CompCert compiler.

2 CompCert Overview

An overview of the CompCert-based workflow is given in Fig. 1. 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; as an example CompCert has successfully been used with the GCC and Diab compilers.

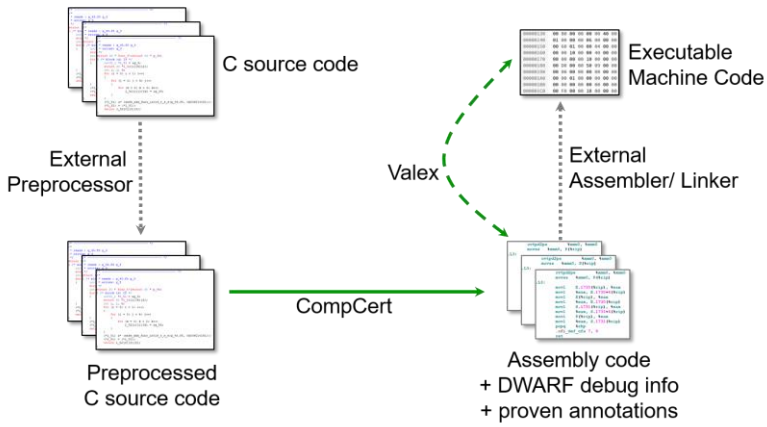

Fig. 1. CompCert Workflow

While early versions of CompCert were limited to single-file inputs, CompCert now also supports separate compilation (cf. Sec. 4.3). It reads the set of preprocessed C files produced by the legacy preprocessor, performs a series of code generation and optimization steps (cf. Sec. 3.1) and produces a set of assembly files enhanced by debug information.

CompCert generates DWARF2[2] 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 (cf. Sec. 3.2).

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. 5).

[2] cf. DWARF Debugging Standard Website (http://dwarfstd.org).

2.1 Availability

The CompCert sources can be downloaded from Inria[3] free of charge; the current state of the development can be viewed on Github[4]. In addition, a released distribution with long-term support is available from AbsInt, either as a source code download or as pre-compiled binary for Windows and Linux. 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.

3 CompCert Design

Like other compilers, CompCert is structured as a pipeline of compilation passes, depicted in Fig. 2 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 four successive phases, described in the following sections.

3.1 CompCert 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 parser is automatically generated from the grammar of the C language by the Menhir parser generator, along with a Coq[5] proof of correctness of the parser (Jourdan et al.2012). 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 (Motor Industry Software Reliability Association 2004) and almost all of ISO C99 (ISO 1999), with the exception of variable-length arrays and unstructured, non-MISRA `switch` statements (e.g. Duff's device[6]).

[3] http://compcert.inria.fr/download.html

[4] https://github.com/AbsInt/CompCert

[5] Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs (http://coq.inria.fr).

[6] http://en.wikipedia.org/wiki/Duff's_device

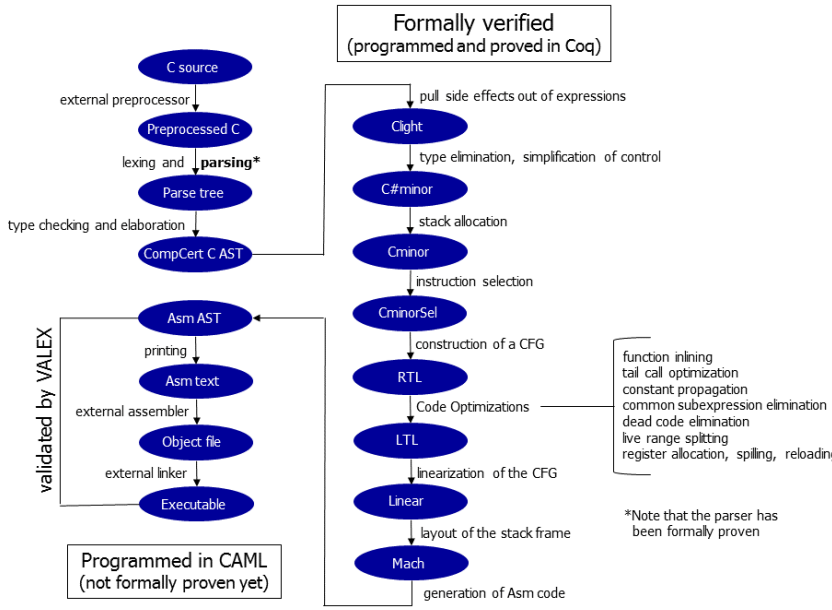

Fig. 2. CompCert Phases

C front-end compiler The second phase first re-checks 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 behaviours 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[7] code.

Back-end compiler The 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 (George

[7] Cminor is a simple, untyped intermediate language featuring both structured (`if/else`, loops) and unstructured control (`goto`).

and Appel 1996). 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 an off-the-shelf assembler and linker to produce object files and executable files. To improve confidence, the translation validation tool Valex re-checks the executable file produced by the linker against the assembly language AST produced by the back-end.

3.2 CompCert Annotations

CompCert provides a general mechanism to attach free-form annotations with formal semantics (plain text possibly mentioning the values of variables) to C program points. The annotations are transported throughout compilation, all the way to the generated assembly code, where variable names are expressed in terms of machine code addresses and machine registers. A simple example is the annotation:

```
__builtin_annot("x is %1 and y is %2", x, y);
```

The formal semantics of such an annotation is that of a *pro forma* "print" statement: when executed, an observable event is added to the trace of I/O operations which records the text of the annotation and the values of the argument variables `x` and `y`. In the generated machine code, annotations produce no instructions, just an assembler comment or debugging information consisting of the text of the annotations where the escapes (`%1` and `%2`) are replaced by the actual locations (in registers or in memory) where the argument variables `x` and `y` were placed by the compiler. Hence we obtain:

```
# annotation: x is r7 and y is mem(word,r1+16)
```

if `x` was allocated to register `r7` and `y` was allocated to a stack location at offset 16 from the stack pointer `r1`.

A first advantage of this mechanism is that it provides proven traceability: the link between machine-level storage cells and source-level variables is covered by the proof. Another typical use of annotations is to track pieces of code such as library function symbols. We can put annotations at the beginning and the end of every library function symbol, recording the values of its arguments and result var-

iables. The semantic preservation proof therefore guarantees that symbols are entered and finished in the same order and with the same arguments and results, both in the source and generated code. This ensures in particular that the compiler did not reorder or otherwise alter the sequence of symbol invocations present in the source program -- a guarantee that cannot be obtained by observing system calls and volatile memory accesses only.

A third application of the annotation mechanism is to enable WCET tools to compute more precise worst-case execution time (WCET) bounds. Indeed, WCET tools like aiT (Ferdinand and Heckmann 2004) 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 Analyser. 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 (Bedin Franca et al.2012).

4 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.

4.1 Operational Semantics

To state the semantic preservation 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 behaviours. These behaviours 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. Behaviours also contain a trace of all observable input/output actions performed by the program, such as system calls, annotations as described in Sec. 3.2., and accesses to "volatile" memory areas that could correspond to a memory-mapped I/O device.

Technically, the semantics of the various languages are specified in small-step operational style as labelled transition systems (LTS). A LTS is a mathematical relation $currentstate \xrightarrow{trace} nextstate$ that describes one step of execution of the program and its effect on the program state. For assembly languages, program states

are just the contents of processor registers and memory locations. For higher-level languages such as C, program states have a richer structure, including memory contents, an abstract program point designating the statement or expression to execute next, environments mapping variables to memory locations, as well as an abstraction of the stack of function calls.

A generic construction defines the observable behaviours from these transition systems, by iterating transitions from an initial state (the initial call to the `main` function): $S_0 \xrightarrow{t1} S_1 \xrightarrow{t2} \ldots$ Such sequences of transitions can go on infinitely, denoting a program that runs forever, or stop on a state S_n from which no transition is possible, denoting a terminating execution. The concatenation of the traces $t_1.t_2\ldots$ describes the I/O actions performed. Several behaviours 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 behaviours).

4.2 Semantic Preservation

To a first approximation, a compiler preserves semantics if the generated code has exactly the same set of observable behaviours 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 behaviours: 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" run-time errors present in the source code, replacing them by any behaviour of its choice. (This is the essence of the notion of "undefined behaviour" in the ISO C standards.) As an example consider an out-of-bounds array access:

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

This is undefined behaviour 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 address 1, with unpredictable effects. Finally, an optimizing compiler like Comp-

Cert 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 behaviours:

Definition 1 (Semantic Preservation): *If the compiler produces compiled code C from source code S, without reporting compile-time errors, then every observable behaviour of C is either identical to an allowed behaviour of S, or improves over such an allowed behaviour of S by replacing undefined behaviours with more defined behaviours.*

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, 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.

4.3 Separate Compilation and Linking

In Definition 1, semantic preservation is stated in terms of *whole programs*: the source program *S* is compiled in one run of the compiler to an executable program *C*, whose semantics is then related to that of *S*. This is not how compilers are used in practice: the source program is composed of several *compilation units* residing in different files; each unit is *separately compiled* to an object file; finally, the executable is obtained by *linking* together the object files.

The implementation of CompCert supports this familiar separate compilation scenario (the `-c` command-line option). However, until release 2.7, the proof of semantic preservation did not cover this scenario, leaving open the possibility that CompCert could miscompile when used for separate compilation. Kang et al. (Kang et al.2016) found an example of this problem in CompCert 2.5. Consider the declaration:

```
int * const p;
```

If this is the only declaration of `p` in the program, it gets initialized to the default value for a pointer, namely the null pointer. Since `p` is `const`, it keeps this value through the execution of the program. The alias analysis of CompCert 2.5 was building on those observations to conclude that `p` has an empty points-to set. Memory accesses were then optimized under this assumption. All this is correct in a whole-program scenario, where the compiler sees that the declaration above is the only declaration of `p`. However, after separate compilation of the unit containing the declaration above, the unit can be linked with another unit that declares `p` with a non-null initialization:

```
int x; int * const p = &x;
```

In the resulting executable program, `p` is not the null pointer, and the optimizations performed by CompCert 2.5 can be wrong.

This particular issue was fixed in CompCert 2.6 by making the alias analysis more conservative. However, we still missed formal evidence that all CompCert optimizations are correct in the presence of separate compilation. To this end, and following the approach invented by Kang et al. (Kang et al.2016), CompCert 2.7 strengthens the statement and proof of semantic preservation to take separate compilation and linking into account:

Definition 1 (Semantic preservation with separate compilation): *Consider n source compilation units S_1, ... S_n that compile separately to compiled units C_1, ..., C_n without reporting compile-time errors. Assume that the source units link together without error to a whole source program $S = S_1 \oplus ... \oplus S_n$. Then, the compiled units link without errors to a whole compiled program $C = C_1 \oplus ... \oplus C_n$. Moreover, every observable behaviour of C is either identical to or improved upon an allowed behaviour of S, as in Definition 1.*

This approach of Kang et al. relies on a notion of syntactic linking between two or more compilation units, written $\oplus$ in the definition above, that extends the operations traditionally performed over object files by linkers to all the source, intermediate, and target languages of CompCert. For instance, in the case of the source CompCert C language, syntactic linking is defined by considering all declarations of identically-named global variables and functions. If the declarations are compatible, as in `extern int x` and `int x = 1`, the most precise declaration is retained (`int x = 1`). If two declarations are incompatible, such as `int x = 1` and `int x = 2`, syntactic linking fails.

A limitation of this approach is that it describes only linking between compilation units written in the same language, but not linking between, say, a C source file and a hand-written assembly file. Formalizing and reasoning upon such cross-language linking and interoperability is a difficult, active research problem (Ahmed 2015, Neis et al.2015, Stewart et al.2015).

5 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.

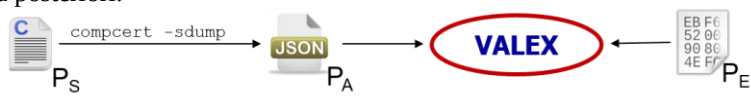

Fig. 3. Translation Validation with Valex

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 (Abs 2016). 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).

6 The Reference Interpreter

The CompCert compiler also provides an interpreter that can execute simple C programs without compilation. More precisely, preprocessing, parsing and initial elaboration are performed; the resulting CompCert C abstract syntax is then executed by interpretation.

This is a *reference* interpreter, meaning that it implements exactly the formal semantics of CompCert C against which CompCert is proved correct. In particular, all behaviours that are undefined in the formal semantics are reported as such by the interpreter. In contrast, compiling a program that invokes undefined behaviour often causes this behaviour to become defined or be optimized away, making it impossible to observe by running the compiled executable. Likewise, the reference interpreter can explore all evaluation orders allowed by the CompCert C formal semantics, while CompCert, as a compiler, implements only one of the possible evaluation orders. This makes the reference interpreter very useful to explore the CompCert C semantics and test C code fragments for undefined behaviours.

Here is an example of such an exploration. Consider the program:

```
#include <stdio.h>
int x[2] = { 12, 34 };
int main(void)
{
  int i = 65536 * 65536 + 2; // will overflow
  printf("i = %d\n", i);
  printf("x[i] = %d\n", x[i]);
}
```

Running it with the `-interp -quiet` options through CompCert, we obtain:

```
i = 2
Stuck state: in function main, expression
        printf(<ptr __stringlit_2>, <loc x+8>)
Stuck subexpression: <loc x+8>
ERROR: Undefined behaviour
```

The first line (`i = 2`) is the output of the `printf` statement. It shows that the arithmetic overflow in the computation of `i` is not undefined behaviour in CompCert C but is defined modulo 2^{32}. The following lines diagnose an undefined behaviour, namely accessing the array `x` outside of its bounds. (Here, `<loc x + 8>` means dereferencing the memory location 8 bytes past the beginning of `x`.) A `-trace` option is available which provides a full trace of interpretation, showing every execution step taken and every intermediate state.

Technically, the reference interpreter is obtained and proved correct as follows. In Coq, a computable function `step` from execution states to sets of (observable events, execution states) pairs is defined, then proved sound and complete with respect to the transition relation of the CompCert C operational semantics:

$$S \xrightarrow{t} S' \Leftrightarrow (t, S') \in step(S)$$

The `step` function is then extracted to OCaml[8] code and linked with handwritten code that iterates `step` to form transition sequences. By default, only one successor state in `step`(*S*) is deterministically chosen, but the `-random` option causes this choice to be made randomly between all possible successors, and the `-all` option triggers an exhaustive breadth-first exploration instead.

There are some limitations with using CompCert in reference interpreter mode. First, the only standard C library functions supported are `printf`, `malloc` and `free`. Hence, the only programs that can currently be interpreted are self-contained tests with fixed inputs. Second, interpretation is 10^5 to 10^6 times slower than execution of compiled code, unless exhaustive exploration is requested, in which case interpretation is exponentially slower.

Despite these limitations, we found the reference interpreter of CompCert useful: first, to animate the formal semantics of CompCert C, helping build confidence in it; second, to test code fragments for undefined behaviours.

7 The Confidence Argument

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. As described in Sec. 2.1 the source code and the corresponding proofs are freely available, as is the proof assistant Coq. So the source code is amenable to manual review, the proof is reproducible for everybody and can be manually reviewed as well.

The formal proofs do not cover the following aspects:

1. The preprocessing phase
2. The correctness of the specifications used for the formal proof, i.e. the formal semantics of C and assembly,
3. Elements of the parsing phase, mostly lexing, type checking and elaboration
4. The assembly 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 (cf. item 3) 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 concreteC syntax, facilitating comparison with the C source.

[8] https://ocaml.org

However, it is possible to provide additional confidence beyond the significance of the validation suite, in particular for items 1 and 4. The CompCert reference interpreter described in Sec. 6 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 (Pnueli et al.1998).

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 so 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.

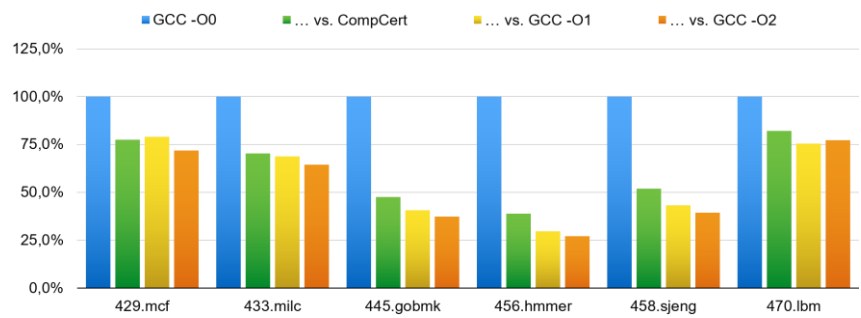

Fig. 4. Execution Time Comparison for SPEC Benchmarks on PowerPC

8 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). The

result of the SPEC CPU2006[9] benchmarks measured on a PowerPC G5 are illustrated in Fig. 4 and Fig. 5, where Fig. 4 shows the execution time of the generated code and Fig. 5 its size. The experiments show that the code generated by CompCert runs about 40% faster than the code generated by GCC without optimizations, approximately 12% slower than GCC 4 at optimization level 1, and 20% slower than GCC 4 at optimization level 2.

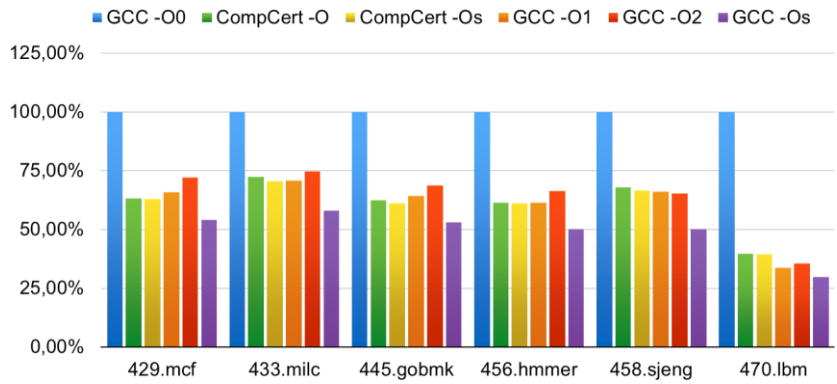

Fig. 5. Code Size Comparison for SPEC Benchmarks on PowerPC.

Regarding code size, the code generated by CompCert in modes -Os is about 1% smaller than in mode -O; it is about 40% smaller than the code generated by GCC -O0, similar to the code size of GCC -O1 (less than 1% difference), 5% smaller than GCC -O2, and about 20% larger than the code of GCC -Os.

Since SPEC is a general-purpose compiler benchmark we also considered another benchmark which is more oriented towards embedded computing. This suite comprises computational kernels from various application areas: signal processing, physical simulation, 3d graphics, text compression, and cryptography.

The results are similar than with the SPEC benchmarks: executing the code generated by CompCert -O reduces execution time to 48% compared to GCC -O0, GCC -O1 achieves 45%, and GCC -O2 42%. Hence the code generated by CompCert runs about 52% faster than the code generated by GCC without optimizations, approximately 11% slower than GCC 4 at optimization level 1, and 23% slower than GCC 4 at optimization level 2.

Regarding code size, the code generated by CompCert in modes -Os here is less than 1% smaller than in mode -O; it is about 17% smaller than the code generated by GCC -O0, 4% larger than the code size of GCC -O1, similar to GCC-O2 (difference smaller than 1%), and about 5% larger than the code of GCC -Os.

In general, due to lack of aggressive loop optimizations, performance is lower on HPC[10] codes involving dense matrix computations. This is also the main reason

[9] http://www.spec.org/cpu2006

[10] High-Performance Computing

for the difference in execution time between CompCert and GCC with high optimization levels.

The performance of CompCert on ARM is similar to the PowerPC architecture. 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 on the benchmark suite.

9 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. 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

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

[Ahmed 2015] Amal Ahmed. Verified compilers for a multi-language world. In *SNAPL 2015: 1st Summit on Advances in Programming Languages*, volume 32 of *LIPIcs*, pages 15–31. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.

[Bedin Franca *et al.*2012] Ricardo Bedin Franca, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *ERTS 2012: Embedded Real Time Software and Systems*, 2012.

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

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

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

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

[Jourdan *et al.*2012] Jacques-Henri Jourdan, FranÃ§ois Pottier, and Xavier Leroy. Validating LR(1) parsers. In *ESOP 2012: 21st European Symposium on Programming*, volume 7211 of *LNCS*, pages 397–416. Springer, 2012.

[Kang *et al.*2016] Jeehoon Kang, Yoonseung Kim, Chung-Kil Hur, Derek Dreyer, and Viktor Vafeiadis. Lightweight verification of separate compilation. In *POPL 2016: 43rd symposium on Principles of Programming Languages*, pages 178–190. ACM, 2016.

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

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

[Neis *et al.*2015] Georg Neis, Chung-Kil Hur, Jan-Oliver Kaiser, Craig McLaughlin, Derek Dreyer, and Viktor Vafeiadis. Pilsner: a compositionally verified compiler for a higher-order imperative language. In *ICFP 2015: 20th International Conference on Functional Programming*, pages 166–178. ACM, 2015.

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

[Pnueli *et al.*1998] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS'98: Tools and Algorithms for Construction and Analysis of Systems*, volume 1384 of *LNCS*, pages 151–166. Springer, 1998.

[Stewart *et al.*2015] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *POPL 2015: 42rd symposium on Principles of Programming Languages*, pages 275–287. ACM, 2015.

[Yang *et al.*2011] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI '11*, pages 283–294. ACM, 2011.