Formally verified optimizing compilation in ACG-based flight control software

Ricardo Bedin França*†, Sandrine Blazy‡, Denis Favre-Felix*, Xavier Leroy§, Marc Pantel† and Jean Souyris*
*AIRBUS Operations SAS
316 Route de Bayonne, Toulouse, France
{ricardo.bedin-franca,denis.favre-felix,jean.souyris}@airbus.com
†Institut de Recherche en Informatique de Toulouse
2 Rue Charles Camichel, Toulouse, France
{ricardo.bedinfranca,marc.pantel}@enseeiht.fr
‡IRISA - Université de Rennes 1
Campus de Beaulieu, Rennes, France
sandrine.blazy@irisa.fr
§ INRIA Rocquencourt
Domaine de Voluceau, Le Chesnay, France
xavier.leroy@inria.fr

***Abstract*—This work presents an evaluation of the CompCert formally specified and verified optimizing compiler for the development of DO-178 level A flight control software. First, some fundamental characteristics of flight control software are presented and the case study program is described. Then, the use of CompCert is justified: its main point is to allow optimized code generation by relying on the formal proof of correctness and additional compilation information instead of the current un-optimized generation required to produce predictable assembly code patterns. The evaluation of its performance (measured using WCET and code size) is presented and the results are compared to those obtained with the currently used compiler.**

***Keywords*-Safety critical systems, Optimized code generation, Toolset performance evaluation**

I. Introduction

Flight Control Software (FCS) development is a very challenging task: not only it has the typical constraints of other software projects, such as budget, delivery schedule and available hardware, but also those that apply to hard real-time, safety-critical software. There are specific regulations for the avionics domain - in particular, the DO-178/ED12 [1] that enforce rigorous development of embedded software in avionics systems.

In this context, it is not trivial to develop software that can perform optimally while complying with all requirements. This paper focuses on the compilation, which is a very important step in the generation of good performance critical software: compilers are very complex tools that carry out the delicate task of generating low-level code that behaves exactly as expected from the high-level source code. Thus, the compiler has clear influence over software performance and safety, but it is commonly developed by third parties (Commercial Off-The-Shelf "COTS" software) and are not necessarily oriented towards the needs of avionics software.

We present experiments carried out by Airbus with the use of a formally-verified compiler, CompCert, in flight control software development. The goal of these experiments is to evaluate its performance in a realistic environment, with software modules similar to actual FCS ones and the same development and verification tools as real FCS. This paper extends the performance evaluation presented in [2], using a new version of CompCert that includes an annotation mechanism (for traceability and timing analysis purposes) and using more criteria to compare a CompCert-based development with the currently used approach. This paper compares Worst-Case Execution Times (WCET) taking into account the size and function of the modules.

The paper is structured as follows: Section II presents the fundamentals of flight control software development and the method we use to assess software performance. Section III presents the CompCert compiler and its features that led to its choice for this work. Section IV presents the results of its performance evaluation and Section V draws conclusions and research perspectives.

II. Flight Control Software, Compilers and Performance

A. An Overview of Flight Control Software

While older airplanes had only mechanical, direct links between the pilots' inputs and their actuators, modern aircraft rely on computers and electric connections to transmit these inputs. As Traverse and Brière [3] describe, digital, electrical flight control systems are used on all aircraft control surfaces since the development of the Airbus A320.

Hardware and software used in flight control systems are subject to very strict requirements, just as any other component. Every kind of avionics software is subject to the DO-178 (currently, version B) regulations, and the DO-178B guidelines become more numerous and more stringent according to the criticality of a given software program. The correct operation of flight control software is essential to a safe flight, hence they belong to "software level A" and their planning, development, verification and project management must comply with very strict regulations.

In addition, aircraft manufacturers usually have their own internal development constraints, ranging from additional safety considerations (e.g. dissymetry and redundancy) to industrial ones, such as delivery delays.

B. The Case Study

In order to carry out a realistic study, we have chosen to use a case study that closely resembles actual flight control software: not only the source code is representative (in functionalities and size) of flight control laws, but the target computer is also representative of true flight control hardware, so as to illustrate typical constraints on hardware usage. The hardware and software used in this work are similar to those described in [4]: the relevant hardware in the scope of this work comprises the MPC755 microprocessor, and an external RAM memory. The MPC755 is a single-core microprocessor, which is much less complex than modern multi-core ones but does have pipelines, an internal cache and superscalar architecture, three elements that make its behavior less predictable. Naive timing analysis of this microprocessor could lead to the "timing anomalies" described by Lundqvist and Stenström [5].

It must be noted that the choices of hardware and software are led by a combination of factors – besides performance needs, there are other constraints, such as weight, size, power dissipation, cost and – most importantly in the scope of this paper – verifiability. Thus, choosing the MPC755 for this case study is consistent with flight control systems of modern aircraft. The development process described below also reflects the intent of developing deterministic and verifiable software.

The development described in the next paragraphs follows the basic steps that are recommended by the DO-178B: specification, design, coding/integration and verification. One must take into account, though, that specification and design are treated together, due to the highly detailed software specification.

1) Specification and Design: The largest part of the software program – the "application" subset, which contains the implementation of the flight control laws – is specified as a set of sheets with the graphical formalism SCADE, each sheet being composed of interconnected basic operators (addition, filter, etc). There is no "main sheet": they all belong to the same hierarchical level and they communicate via their input and output parameters. In order to simplify the specification and the code generation, all the symbols used in the sheets are custom-made by the developers. SCADE (V6) state machines are not used, mainly for determinism purposes – conditional statements are kept inside some library symbols.

Figure 1 depicts instances of the custom-made symbols ABS, HWACQDSI, BPO and AFDX_FSOUT. Each symbol has inputs and outputs, which are connected to their left and right sides, respectively. At the bottom of the HWACQDSI and AFDX_FSOUT symbols, there are some "hidden inputs", which have the same semantics as a normal input but are used to underline symbol parameters (in our example, integer constants) that are not related to the data flow.

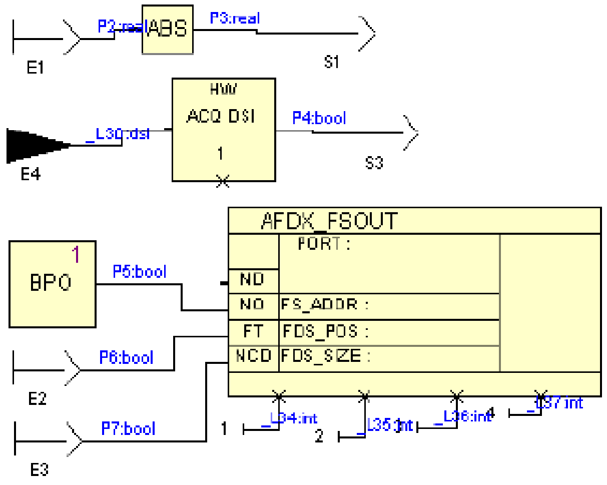


Figure 1. SCADE custom-made operators

The program also contains a manually-coded part that goes through distinct specification and design phases, but further details about this part are beyond the scope of this paper, as it makes no use of automatic code generation.

2) Coding: The specification is translated to source code by an automatic code generator (ACG). Automatic code generation, when applicable, has the advantages of being less error-prone and offering much smaller coding times than manual code generation. In order to lighten the burden of source code verification activities, the ACG is qualified as a development tool, according to the DO-178B standards for level A software.

In this study, we use C as a source code language, as it is widely used in critical systems and there are many development and verification tools for C-coded critical systems. Each symbol is represented in C as a macro: a "symbol library" (a set of macros) is manually coded in order to implement each SCADE operator, and the ACG-generated code consists in a sequence of macro instantiations that respect the data flow specified in SCADE. A SCADE sheet is represented by a C function that contains a sequence of macro instantiations – all the data-flow constraints are taken into account by the ACG in order to make sequential C programs that are consistent with the parallel SCADE ones. It must be noted that all the sheets are activated periodically, but their activation period may vary. Thus, the execution cycle is divided into several sequential "tasks" and each sheet may be activated in one or more tasks during a cycle.

The C code is finally compiled with a COTS compiler[1] and linked to produce an executable file. It must be noted that the compiler – like the vast majority of compilers industrially used – is seen as a "black box" by the development and verification teams. In this case, the object code must be verified thoroughly, and the safest solution to carry out a complete verification taking into account the use of a COTS compiler and the high reactivity of the ACG process is to forbid compiler optimizations in order to force the generation of constant code patterns for each symbol. As our ACG-generated code is a (potentially long) sequence of a limited number of symbols and the symbol library code changes much less often than the application in actual flight control programs, it is less onerous to carry out thorough verification activities over each possible code pattern for this symbols than verifying all code "sheets" in each compilation.

3) Verification: Every development phase must be verified and this verification must meet the DO-178B requirements. As this paper focuses on the compilation, we shall describe the main activities that verify software coding and integration:

- Source Code Verification: The source code must be traceable and compliant to the design (in our case, the SCADE specification). Also, it must respect the software resource and time constraints.
- Object Code Verification: Object code also must be traceable and compliant to the SCADE specification and the integration of software modules, as well as their integration with the target computer, must be verified.

The DO-178B demands requirement-based verification: a program must be verified with respect to its high-level and low-level requirements. In this paper, we suppose that low-level verification is carried out at symbol level (e.g. tests and/or formal proofs of symbol outputs), hence the compiler must not optimize away the symbol outputs, even if they are, indeed, intermediate results of a function.

Usually, these verification activities (especially object code verification) involve testing. For level A software, the whole code must be tested with Multiple Condition/Decision Coverage (MC/DC) and traceability between source code and object code is necessary to validate code coverage:

- If coverage is measured over the source code, traceability is necessary to ensure that there is no added, unverified functionality in the object code. Typical cases of "added, unverified" functionalities could be found in compilers that add array bound checks or that have a complex management for `switch` statements.
- The DO-178B report for clarification [6] states that if coverage is measured over the object code, traceability is necessary to ensure that the measured coverage is equivalent to MC/DC, as the object code (such as Assembly language) may not contain the multiple conditions found in the source code.

Traceability analysis is much less complicated if the object code presents no optimization and no untraceable code added by the compiler. Once again, it is useful to hinder compiler optimizations in order to simplify the verification activities.

C. Estimating Software Performance

Besides being a DO-178B requirement, Worst-Case Execution Time (WCET) analysis is a safe and reliable timing verification in the avionics software context. Hardware and software complexity make the search for an exact WCET nearly impossible; usually one computes time values which are as close as possible to the actual WCET, but always higher than it. In our case study, the main purpose of WCET analysis is to make sure that no application task oversteps its allowed execution time.

As mentioned by Souyris *et al* [4], it was once possible to compute the WCET of avionics software by measurement and analysis, but such method is not feasible in state-of-the-art programs. The current approach at Airbus relies on AbsInt[2]'s automated tool a^3 [7] to compute the WCET via static code analysis of the executable file. In order to obtain accurate results, the tool requires a precise model of the microprocessor and other relevant components; this model was designed in close cooperation between Airbus and AbsInt.

Sometimes it is important or even essential to give a^3 extra information about loop counts or register value bounds to refine its analysis. As described in [4], annotations are necessary when memory access address ranges cannot be computed precisely because of limitations in a^3 value analysis (e.g. floating-point). The imprecisions that arise from such limitations degrade WCET analysis and can go as far as stopping a^3 from completing WCET computation. Such a situation is depicted in Algorithm 1: as a^3 is not yet able to carry out the floating-point comparison, it cannot evaluate the range of addresses that may be accessed in line 6. In this case, a^3 has to continue its computation assuming that the access might occur in any memory address, and the great deal of complexity that is added incurs a strongly overestimated – if not unbounded – WCET. For instance, if it is known that variable i is always within the bounds of the array, this information should be provided to a^3 as an annotation.

In our case, annotations are needed only in a few symbols, so as to compute some addresses more precisely – with a^3,

[1]For confidentiality reasons, the currently used compiler, linker and loader names are omitted.

[2]`www.absint.com`

Algorithm 1 Example of a code that needs annotations

```
register double x;                      // Assume that x fits
                                        // inside the array bounds
register int i;
extern double lookup_table[];
i = (int)x;
register double y = lookup_table[i];
```

this kind of annotation can be assigned only to microprocessor registers, which are depicted in the Assembly code.

Let us assume that Algorithm 1 is part of the C macro of a symbol and that its corresponding (non-optimized) Assembly code is depicted by Algorithm 2. One can notice that the C variable *i* is stored in *r31*, since it is loaded with the resulting value of the floating-point to integer conversion. Thus, if we know that *i* is always between, say, 0 and 9, the annotation should be:

```
instruction "Checkpoint" + 0x14 bytes
is entered with r31 = from 0 to 9;
```

In order to keep the fast pace of the ACG-based approach (and avoid potential human mistakes), an automatic annotation generator was devised to avoid manual activities and keep the efficiency of the development process. Each symbol that needs annotations will need them repeatedly for all of its instances, but it is not difficult to annotate automatically the correct Assembly lines with a non-optimized compilation that always generates similar code patterns for all instances of the symbol. Whenever the macro containing Algorithm 1 is instantiated, an annotation would be needed at the same offset 0x14 from the tag *Checkpoint*. Thus, one has to track the possible code patterns for the symbols that need annotations (to make sure that subtle variations in the code patterns do not change the offset of the instruction that needs an annotation) and find the right offsets to assign those value ranges. This annotation strategy is simple and effective, but would not work if the compiler could optimize the code.

Algorithm 2 Example of a loop that needs annotations

```
    Checkpoint:
00 fctiwz f0,f31
04 stfd f0,8(r1)
08 lwz r31,12(r1)              ▷ a3 cannot infer this value
0c addis r11,r0,lookup_table@ha
10 addi r11,r11,lookup_table@l
14 rlwinm r10,r31,3,0,28       ▷ we should help a3 here
18 lfdx f30,r11,r10
```

Annotations are also used in the manually-coded subsets in order to specify – for instance – the behavior of other hardware components, but those are created manually and are not in the scope of this paper.

III. CompCert: Towards a trusted compiler

One can figure out that, in extremely critical systems, traditional COTS compilers must be used with great caution with respect to code optimization. However, there are recent advances in the compilation field: in the scope of this work, a most promising development is the CompCert[3] compiler. Besides working in a more realistic environment (a large C subset as input language, MPC755 as one of the possible target processors) than other experimental compilers, its development is taking into account the needs of critical systems and its own code is available for study if its end users need to know its internal details in order to devise verification strategies for their software.

As described in [8], CompCert is a multiple-pass, moderately-optimizing compiler that is mostly programmed and proved correct using the Coq proof assistant. Its optimizations are not very aggressive, though: as the compiler's main purpose is to be "trustworthy", it carries out basic optimizations such as constant propagation, common subexpression elimination and register allocation by graph coloring, but no loop optimizations, for instance. As no code optimizations are enabled in the currently used compiler, using a few essential optimization options could already give good performance benefits.

The semantic preservation proof of CompCert guarantees that the generated code behaves as prescribed by the semantics of the source program. The observed behaviors in CompCert include termination, divergence and "going wrong". To strengthen the preservation theorem, behaviors also include a trace of the input-output operations performed during the execution of the program. Input-output operations include system calls (if an operating system is used) as well as memory accesses to global variables declared "volatile" (corresponding in particular to memory-mapped hardware devices). The formal verification of CompCert proves, in effect, that the source program and the generated machine code perform the same input-output operations, in the same order, and with the same arguments and results.

A. CompCert annotation mechanism

To strengthen the guarantees implied by CompCert's formal verification, we have introduced a generic program annotation mechanism enabling programmers to mark source program points and keep track of the values of local variables at these points. Syntactically, annotations are presented as calls to a compiler built-in function, taking a string literal and zero, one or several program variables as arguments:

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

The formal semantics of this statement is that of a *pro forma* "print" statement: when executed, an observable event is added to the trace of I/O operations; this event records

[3]`http://compcert.inria.fr`

the text of the annotation and the values of the argument variables (here, x and y). In the generated machine code, however, annotations produce no instructions, just an assembler comment or debugging information consisting of the text of the annotation where the escapes %1, %2 are replaced by the actual locations (in registers or memory) where the argument variables x, y were placed by the compiler. For example, 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.

Despite executing no instructions, this special comment is still treated, from the standpoint of formal semantics, as a *pro forma* "print", generating an observable event. The semantic preservation proof of CompCert therefore guarantees that annotations are executed in the same order and with the same argument values both in the source C program and in the generated assembly code.

A typical use of annotations is to track pieces of code such as library symbols. We can put annotations at the beginning and the end of every symbol, recording the values of the arguments and result variables of the symbol. 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 codes. 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 systems calls and volatile memory accesses only.

This possibility of finer-grained semantic preservation is most welcome, since some of our verification activities may be carried out at symbol level and semantic preservation needs to be ensured at this level to be useful in our context. In particular, we consider using per-symbol annotations in order to generalize the results of symbol-based tests: the test results for a given symbol remain valid for all possible code patterns generated when instantiating this symbol. This approach is currently under discussion and such discussions are not in the scope of this paper.

Another use of annotations is to communicate additional information to verification tools that operate at the machine code level, such as the WCET analyzer of the a^3 tool suite. Continuing the example of section II-C, we insert a source-level annotation as shown below.

During compilation, this source-level annotation is turned into a special comment in the generated assembly file, where the placeholder %1 is replaced by the machine register containing variable *i*. Algorithm 4 below shows the assembly code generated by CompCert for two successive instantiations of the symbol containing Algorithm 3.

The two instantiations generate significantly different assembly code fragments, since the second instantiation reuses some of the intermediate results computed by the first instantiation (common subexpression elimination). Nonetheless, the two special comments corresponding to the source-level annotation are correctly placed and correctly reveal the location of variable *i*, namely registre *r3*.

Algorithm 3 Adding a source-level annotation to Algorithm 1

```
register double x;
register int i;
extern double lookup_table[];
i = (int)x;
__builtin_annot("a3: entered with %1 = from 0 to 9", i);
register double y = lookup_table[i];
```

Algorithm 4 Generated assembly code for two instantiations

```
10 fctiwz f13, f1
14 stfdu f13, -8(r1)
18 lwz r3, 4(r1)
1c addi r1, r1, r8
20 # annotation: a3: entered with r3 = from 0 to 9
20 rlwinm r4, r3, 3, 0, 28
24 addis r12, r4, (lookup_table)@ha
28 lfd f1, (lookup_table)@l(r12)
...
40 # annotation: a3: entered with r3 = from 0 to 9
20 rlwinm r6, r3, 3, 0, 28
44 addis r12, r6, (lookup_table)@ha
48 lfd f2, (lookup_table)@l(r12)
```

From these special comments and their locations in the assembly listing, an automatic tool can easily extract the information that at points 20 and 40 from the beginning of the current function, register r3 (holding the array index) is in the range $[0, 9]$, and communicate this information to the WCET analyzer.

Some aspects of this annotation mechanism are still under discussion with the CompCert and a^3 developers, but an experimental annotation generator has already been developed and the ease of its development is a testimony to the usefulness of the CompCert annotation mechanism: readily-available, formally-verified variable information simplify the task of automating annotation generation for a^3. One should remember that, in comparison, the annotation generator for the "default" compiler code must be reconfigured for each symbol library change: a new analysis must be carried out in order to verify which are the possible Assembly patterns for all symbols that need annotations, and which are the offsets that need these annotations.

IV. Performance evaluation of CompCert

The evaluation environment is essentially the same as in our previous work [2] and is depicted in Figure 2. CompCert is used only to generate Assembly code from the ACG-coded files, as these files are by far the most voluminous part of the program. Compilation of other software subsets, assembling and linking were done with the compiler, assembler and linker that are used in actual FCS.

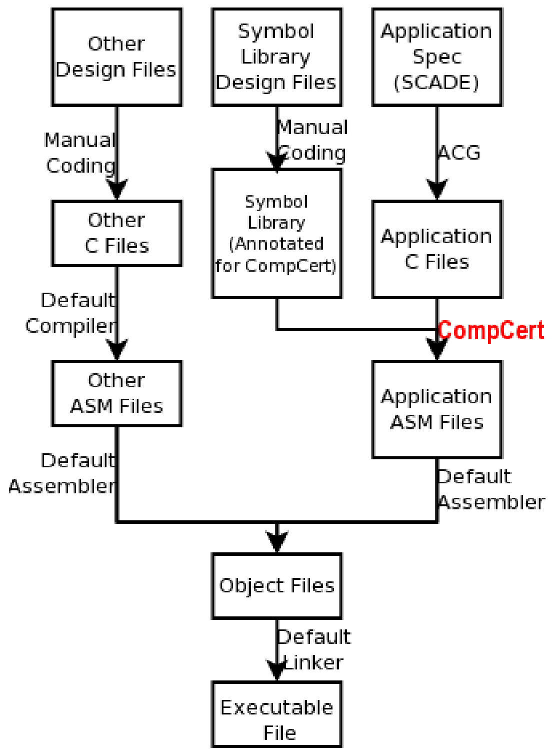
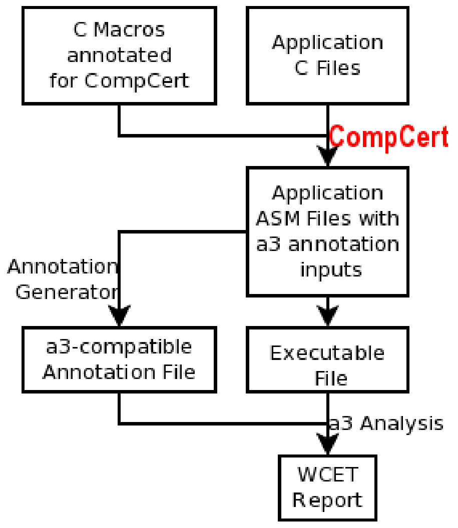


Figure 2. The development chain of the analyzed program

In order to ensure greater realism in the experiments, about 3600 files that are functionally equivalent to a whole flight control program were compiled with CompCert 1.9. These files represent about 3600 SCADE sheets – when compiled with the default compiler, they correspond to 3.96 MB of Assembly code. The symbol library that was used comprises 145 symbols whose sizes vary from less than 10 to more than 100 lines of code. CompCert's source-level annotation mechanism was used to track symbols' inputs and outputs, and also to generate additional information for some variables that need range annotations. As explained in section III-A, this information is available in the generated assembly files, which are examined by the annotation generator to produce an annotation file in the suitable format for a^3.

a^3 was used to compute WCET at two different levels: the most important benchmark is at task level, as it is the measure used for timing analysis in actual programs. While a traditional WCET analysis consists in verifying that each task is performed within its allocated time, we opted to compare the average WCET of all tasks in order to have a synthesis of the results for every task. In addition, we analyze individually the WCET of all SCADE sheets: we do not seek interprocedural optimizations or a register allocation that goes beyond one single module, hence individual WCET computations are meaningful in this context and are useful to find out which kind of algorithms get the most of CompCert's optimizations. The baseline for the benchmark is the WCET of an executable file generated with the default compiler and the compilation options used in a real flight control program. Some analyzed sheets instantiate symbols that need range annotations; CompCert's annotation mechanism was used together with a simple annotation generation script to assign variable ranges for a^3 when needed. Figure 3 depicts the flow of annotation data, from the C macros (where the necessary extra information is specified by the user) to the execution of a^3.



C Macros annotated for CompCert
Application C Files
CompCert
Application ASM Files with a3 annotation inputs
Annotation Generator
a3-compatible Annotation File
Executable File
a3 Analysis
WCET Report

Figure 3. Automatic annotation generation for a^3

In addition to WCET computations, code size measures were carried out as an auxiliary performance indicator – smaller code size often means better-performing code.

The results of the WCET analysis are quite encouraging, as the average WCET improvement per task was 10.6%, which is a significant improvement by flight control software standards. As already pointed out in [2], this is mainly due a better register allocation that saves many loads and stores that had to be performed to keep symbol inputs and outputs on stack.

Figure 4 depicts the WCET computed for all sheets, ordering them according to the WCET obtained when they were compiled with the default compiler. The WCET improvement may change from one region of the graph to another (modules with a very low or very high WCET do not always have a visible improvement, whereas CompCert clearly improved the WCET of those in the middle part of the curves) and even inside a region – the WCET curve for CompCert-compiled modules is not smooth.

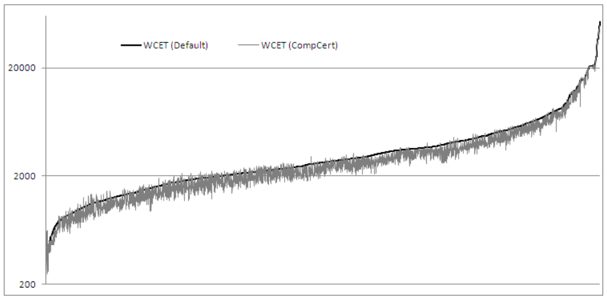


Figure 4. Overall result of WCET comparison

In order to refine the general results obtained by the analysis of this large number of files, special attention was dedicated to files that had extreme values of WCET and code size. The 10% longest and shortest files (in WCET or code size) had results that differed from the average and had specific statistics in order to underline those differences. In addition, some "unexpected" results (e.g. the default compiler performing better than CompCert) were analyzed individually.

The WCET of all analyzed modules was computed for the executable files generated by both compilers, in order to compare them when compiling modules of various WCET and code size value ranges – using the benchmark WCETs and code sizes to classify the modules into categories. The main conclusions from these experiments are:

- In the analyzed program, even if a module is small, there is usually some possibility of optimization but results may vary according to the symbols that are instantiated in a given module. Some symbols have their code vastly improved by CompCert, whereas – in some very exceptional cases – the WCET of a module rises due to the overhead caused by longer function prologues and epilogues. In fact, modules that present very small code size are not quite a reliable source of WCET analysis because even their address in memory becomes a significant factor in WCET analysis.
- Sheets that are not among the fastest or slowest have a slightly better WCET improvement than the overall results. This shows that the optimizations work best when there is enough code to enable their full use, but the code is still compact enough to avoid register spilling.
- A sheet can have a large WCET for two main reasons: either it may have many instructions to execute or it may contain interactions with hardware devices that are time-consuming. In the former case, CompCert usually performs better, except when dealing with spilled variables – the gains become less significant because spilled variables resemble variables compiled with the default compiler. CompCert optimizations can do little or nothing to improve the WCET of a sheet if its symbols spend most of their computation time doing hardware acquisitions and emissions. In our case study, it is more common to have interactions with hardware than register spilling, hence the WCET gain over "long" sheets (larger code size) is more pronounced than the gain over "slow" ones (higher WCET).
- Even with its optimizations turned off, the default compiler sometimes succeeds in selecting more efficient combinations of PowerPC instructions than CompCert. An example is address computations for volatile memory accesses, which CompCert compiles rather naively. We plan to improve the instruction selection phase of CompCert to reduce these inefficiencies.

Table I summarizes the WCET analysis results.

	WCET (CompCert)	Size (CompCert)
All application tasks	-10.6%	-13.8%
Small code size sheets	-2.0%	-14.6%
Small WCET	-10.6%	-12.9%
Average WCET	-12.6%	-14.3%
Average code size	-10.7%	-13.6%
Large code size	-7.7%	-14.2%
Large WCET	-3.8%	-12.4%

Table I
CODE SIZE AND WCET COMPARISON

A. *Verification considerations*

Since the main reason to avoid most optimizing compilers is the ensuing difficulty to verify traceability and compliance of the object code, the performance evaluation was followed by a study of possible verification strategies that could use CompCert's semantic preservation in order to meet the DO-178B requirements without losing the performance gains obtained with its optimizations. This study is currently under way but it is already clear that the "traditional" analysis mentioned in [6] to verify traceability between source code and object code is still feasible with CompCert, as its optimizations remove computational instructions but do not change significantly the code structure (branches, etc). Also, its semantic preservation theorem could be used as a strong argument for traceability and compliance between source code and object code.

V. CONCLUSIONS AND FUTURE WORK

This paper presented an evaluation of the CompCert compiler, based on the characteristics of Airbus flight control software: ACG-based code, modules with different characteristics. Even if we focused on its WCET analysis to assess its performance, CompCert's formal proofs are already seen as a key to bring more confidence in the compilation process, helping to make safe use of code optimizations. Moreover, CompCert's optimizations apply to the vast majority of the modules that are representative of FCS.

The main ongoing work in our CompCert study is the development of a new verification strategy that must be at least as safe as the current one. It is a complex subject on its own but some conclusions drawn from it (e.g. the need for semantic preservation at symbol level) are already being taken into account – as it is likely that we will need semantic preservation at symbol input level, the performance measures were taken using library symbols endowed with CompCert's annotations to preserve the semantics of their inputs and outputs. An important discussion point is the DO-178 interpretation of a tool like CompCert.

The performance evaluation shall not stop at the current state. As the symbol library was coded bearing in mind the current compilation strategy, an interesting work will be recoding it in order to favor optimizing compilation, with fewer intermediate variables and use of Small Data Areas. It is likely that the obtained WCET will be lower and every percent counts if one intends to improve performance.

Another direction for future work is to further improve WCET by deploying additional optimizations in CompCert and proving that they preserve semantics. The WCC project of Falk *et al* [9] provides many examples of profitable WCET-aware optimizations, often guided by the results of WCET analysis. Proving directly the correctness of these optimizations appears difficult. However, equivalent semantic preservation guarantees can be achieved at lower proof costs by *verified translation validation*, whereas each run of a non-verified optimization is verified *a posteriori* by a validator that is proved correct once and for all. For example, Tristan and Leroy [10] show a verified validator for trace scheduling (instruction scheduling over extended basic blocks) that could probably be adapted to handle WCC's superblock optimizations. Rival has experimented the translation validation approach on a wider scope in [11] but, currently, the qualification and industrialization of such a tool seems more complex.

References

[1] *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, Radio Technical Commission for Aeronautics (RTCA) Std., 1982.

[2] R. B. França, D. Favre-Felix, X. Leroy, M. Pantel, and J. Souyris, "Towards Formally Verified Optimizing Compilation in Flight Control Software," in *PPES*, ser. OASIcs, vol. 18. Grenoble, France: Schloss Dagstuhl, 2011, pp. 59–68.

[3] D. Brière and P. Traverse, "AIRBUS A320/A330/A340 Electrical Flight Controls: A Family of Fault-Tolerant Systems," in *FTCS*, 1993, pp. 616–623.

[4] J. Souyris, E. L. Pavec, G. Himbert, V. Jégu, and G. Borios, "Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation," in *Proceedings of the 5th Intl Workshop on Worst-Case Execution Time (WCET) Analysis*, 2005, pp. 21–24.

[5] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*. Washington, DC, USA: IEEE Computer Society, 1999, p. 12.

[6] *Final Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification"*, Radio Technical Commission for Aeronautics (RTCA) Std., 2001.

[7] R. Heckmann and C. Ferdinand, "Worst-case Execution Time Prediction by Static Program Analysis," in *IPDPS 2004*. IEEE Computer Society, 2004, pp. 26–30.

[8] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[9] H. Falk and P. Lokuciejewski, "A compiler framework for the reduction of worst-case execution times," *The International Journal of Time-Critical Computing Systems (Real-Time Systems)*, vol. 46, no. 2, pp. 251–300, 2010.

[10] J.-B. Tristan and X. Leroy, "Formal verification of translation validators: A case study on instruction scheduling optimizations," in *35th symposium Principles of Programming Languages*. ACM Press, 2008, pp. 17–27.

[11] X. Rival, "Symbolic transfer functions-based approaches to certified compilation," in *31st Symposium Principles of Programming Languages*. ACM Press, 2004, pp. 1–13.