The CompCert C verified compiler: Documentation and user's manual

Xavier Leroy

The CompCert C verified compiler

Documentation and user's manual

Version 2.6

Xavier Leroy
INRIA Paris
December 21, 2015

Contents

Introduction

This document is the user's manual for the CompCert C verified compiler. It is organized as follows:

- Chapter 1 gives an overview of the CompCert C compiler and of the formal verification of compilers.
- Chapter 2 explains how to install CompCert C.
- Chapter 3 explains how to use the CompCert C compiler.
- Chapter 4 explains how to use the CompCert C reference interpreter.
- Chapter 5 describes the subset of the ISO C99 language that is implemented by CompCert.
- Chapter 6 describes the supported language extensions: pragmas, attributes, built-in functions, inline assembly.

Chapter 1

CompCert C: a trustworthy compiler

Traduttore, traditore ("Translator, traitor")
(Italian proverb)

CompCert C is a compiler for the C programming language. Its intended use is the compilation of life-critical and mission-critical software written in C and meeting high levels of assurance. It accepts almost all of the ISO C 99 and ANSI C languages, with some exceptions and a few extensions. It produces machine code for the PowerPC, ARM, and IA32 (x86 32-bits) architectures. Performance of the generated code is decent but not outstanding: on PowerPC, about 90% of the performance of GCC version 4 at optimization level 1.

What sets CompCert C apart from any other production compiler, is that it is *formally verified*, using machine-assisted mathematical proofs, to be exempt from *miscompilation* issues. In other words, the executable code it produces is proved to behave exactly as specified by the semantics of the source C program. This level of confidence in the correctness of the compilation process is unprecedented and contributes to meeting the highest levels of software assurance. In particular, using the CompCert C compiler is a natural complement to applying formal verification techniques (static analysis, program proof, model checking) at the source code level: the correctness proof of CompCert C guarantees that all safety properties verified on the source code automatically hold as well for the generated executable.

1.1 Can you trust your compiler?

Compilers are complicated pieces of software that implement delicate algorithms. Bugs in compilers do occur and can cause incorrect executable code to be silently generated from a correct source program. In other words, a buggy compiler can insert bugs in the programs that it compiles. This phenomenon is called *miscompilation*.

Several empirical studies demonstrate that many popular production compilers suffer from miscompilation issues. For example, in 1995, the authors of the NULLSTONE C conformance test suite reported that

> NULLSTONE isolated defects [in integer division] in twelve of twenty commercially available compilers that were evaluated. (http://www.nullstone.com/htmls/category/divide.htm)

A decade later, E. Eide and J. Regehr showed similar sloppiness in C compilers, this time concerning volatile memory accesses:

> We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables. This result is disturbing because it implies that embedded software and operating systems — both typically coded in C, both being bases for many mission-critical and safety-critical applications, and both relying on the correct translation of volatiles — may be being miscompiled. [3]

More recently, Yang *et al* generalized their testing of C compilers and, again, found many instances of miscompilation:

> We created a tool that generates random C programs, and then spent two and a half years using it to find compiler bugs. So far, we have reported more than 325 previously unknown bugs to compiler developers. Moreover, every compiler that we tested has been found to crash and also to silently generate wrong code when presented with valid inputs. [11]

For non-critical, "everyday" software, miscompilation is an annoyance but not a major issue: bugs introduced by the compiler are negligible compared to those already present in the source program. The situation changes dramatically, however, for safety-critical or mission-critical software, where human lives, critical infrastructures, or highly-sensitive information are at stake. There, miscompilation is a non-negligible risk that must be addressed by additional, difficult and costly verification activities such as extra testing and code reviews of the generated assembly code.

An especially worrisome aspect of the miscompilation problem is that it weakens the usefulness of formal, tool-assisted verification of source programs. Increasingly, the development process for critical software includes the use of formal verification tools such as static analyzers, deductive verifiers (program provers), and model checkers. Advanced verification tools are able to automatically establish valuable safety properties of the program, such as the absence of run-time errors (no out-of-bound array accesses, no arithmetic overflows, etc). However, most of these tools operate at the level of C source code. A buggy compiler has the potential to invalidate the safety guarantees provided by source-level formal verification, producing an incorrect executable that crashes or misbehaves from a formally-verified source program.

1.2 Formal verification of compilers

The CompCert project puts forward a radical, mathematically-grounded solution to the miscompilation problem: the formal, tool-assisted verification of the compiler itself. By applying program proof techniques to the source code of the compiler, we can prove, with mathematical certainty, that the executable code produced by the compiler behaves exactly as specified by the semantics of the source C program, therefore ruling out all risks of miscompilation [6].

Compiler verification, as outlined above, is not a new idea: the first compiler correctness proof (for the translation of arithmetic expressions to a stack machine) was published in 1967 [8], then mechanized as early as 1972 using the Stanford LCF proof assistant [9]. Since then, compiler verification has been the topic of much academic research. The CompCert project carries this line of work all the way to a complete, realistic, optimizing compiler than can be used in the production of critical embedded software systems.

Semantic preservation The formal verification of CompCert consists in proving the following theorem, which we take as the high-level specification of a correct compiler:

> **Semantic preservation theorem:**
> For all source programs *S* and compiler-generated code *C*,
> if the compiler, applied to the source *S*, produces the code *C*,
> without reporting a compile-time error,
> then the observable behavior of *C* improves on one of the allowed observable behaviors of *S*.

In CompCert, this theorem has been proved, with the help of the Coq proof assistant, taking *S* to be abstract syntax trees for the CompCert C language (after preprocessing, parsing, type-checking and elaboration), and *C* to be abstract syntax trees for the assembly-level Asm language (before assembling and linking). (See §1.3 for more details.)

There are three noteworthy points in the statement of semantic preservation above:

- First, the compiler is allowed to fail at compile-time and refuse to generate code. This can happen if the source program *S* is syntactically incorrect or contains a type error, but also if the internal capacity of the compiler is exceeded. (For instance, CompCert C will refuse to compile a function having more than 4 Gb of local variables, since such a function cannot be executed on any 32-bit target platform.)
- Second, the compiler is allowed to select one of the possible behaviors of the source program. The C language has some nondeterminism in expression evaluation order; different orders can result in several different observable behaviors. By choosing an evaluation order of its liking, the compiler implements one of these valid observable behaviors.
- Third, the compiler is allowed to improve the behavior of the source program. Here, *to improve* means to convert a run-time error (such as crashing on an integer division by zero) into a more defined behavior. This can happen if the run-time error (e.g. division by zero) was optimized away (e.g. removed because the result of the division is unused). However, if the source program is known to be free of run-time errors, perhaps because it was verified using static analyzers or deductive program provers, improvement as described above never takes place, and the generated code behaves exactly as one of the allowed behaviors of the source program.

What are observable behaviors? In a nutshell, they include everything the user of the program, or the physical world in which it executes, can "see" about the actions of the program, with the notable exception of execution time and memory consumption. More precisely, we follow the ISO C standards in considering that we can observe:

- Whether the program terminates or diverges (runs forever), and if it terminates, whether it terminates normally (by returning from the `main` function) or on an error (by running into an undefined behavior such as integer division by zero).
- All calls to standard library functions that perform input/output, such as `printf()` or `getchar()`.
- All read and write accesses to global variables of `volatile` types. These variables can correspond to memory-mapped hardware devices, hence any read or write over such a variable is treated as an input/output operation.

The observable behavior of a program is, therefore, a *trace* of all I/O and volatile operations it performs, plus an indication of whether it terminates and how it terminates (normally or on an error).

How do we define the possible behaviors of a source or executable program? This is the purpose of a formal semantics for the corresponding languages. A formal semantics is a mathematically-defined relation between programs and their possible behaviors. Several such semantics are defined as part of CompCert's verification, including one for the CompCert C language and one for the Asm language (assembly code for each of the supported target platforms). These semantics can be viewed as mathematically-precise renditions of (relevant parts of) the ISO C 99 standard document and of (relevant parts of) the reference manuals for the PowerPC, ARM and IA32 architectures.

What does semantic preservation tell us about source-level verification? A straightforward corollary of the semantic preservation theorem shows the following:

> Let Σ be a set of acceptable behaviors, characterizing a desired safety or liveness property of the program.
> Assume that a source program S satisfies Σ: all possible observable behaviors of S are in Σ.
> Further assume that the compiler, applied to the source S, produces the code C.
> Then, the compiled code C satisfies Σ: the observable behavior of C is in Σ.

The purpose of a sound source-level verification tool is precisely to establish that a specification Σ holds for all possible executions of a source program S. The specification can be defined by the user, for instance as pre- and post-conditions, or fixed by the tool, for instance the absence of run-time errors. Therefore, a formally-verified compiler guarantees that if a sound source-level verification tool says "yes, this program satisfies this specification", then the compiled code that really executes also satisfies this specification. In other words, using a formally-verified compiler justifies verification at the source level, insofar as the guarantees established over the source program carry over to the compiled code that actually executes in the end.

How do we conduct the proof of semantic preservation? Because of the inherent complexity of an optimizing compiler, the proof is a major endeavor. We split it into 15 separate proofs of semantic preservation, one for each pass of the CompCert compiler. The final semantic preservation theorem, then, follows from the composition of these separate proofs. For every pass, we must prove semantic preservation for all possible input programs and for all possible executions of the input program (there can be many such executions depending on the unpredictable results of input operations). To this end, we need to consider every possible reachable state in the execution of the program and every transition that can be performed from this state according to the formal semantics. The proofs take advantage of the inductive structure of programming languages: for example, to show that a compound expression $a + b$ is correctly compiled, we assume, by induction hypothesis, that the two smaller subexpressions a and b are correctly compiled, then combine these results with reasoning specific to the + operator.

If the compiler proof were conducted using paper and pencil, it would fill hundreds of pages, and no mathematician would be willing to check it. Instead, we leverage the power of the computer: CompCert's proof of correctness is conducted using the Coq proof assistant, a software tool that helps us construct the proof in interaction with the tool, then automatically re-checks the validity of the proof [2]. Such mechanization of the proof brings near-absolute confidence in its validity.

How effective is formal compiler verification? As mentioned above and detailed in §1.3, CompCert is still a work in progress, and complete, end-to-end formal verification has not been achieved yet: as of this writing, about 90% of the compiler's algorithms (including all optimizations and all code generation algorithms) are proved correct in Coq, but the remaining 10% (including elaboration, presimplifications, assembling and linking) are not verified. This can only improve in the future. Nonetheless, this incomplete formal verification already demonstrates major correctness improvements compared with ordinary compilers. Yang *et al* report:

> The striking thing about our CompCert results is that the middleend bugs we found in all other compilers are absent. As of early 2011, the under-development version of CompCert is the only compiler we have tested for which Csmith cannot find wrong-code errors. This is not for lack of trying: we have devoted about six CPU-years to the task. The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework, where safety checks are explicit and machine-checked, has tangible benefits for compiler users. [11]

1.3 Structure of the CompCert C compiler

The general structure of the CompCert C compiler is depicted in Figure 1.1. The compilation of a C source file can be conceptually decomposed into the following phases:

1. Preprocessing: file inclusion, macro expansion, conditional compilation, etc. Currently performed by invoking an external C preprocessor (not part of the CompCert distribution), which produces preprocessed C source code.
2. Parsing, type-checking, elaboration, and construction of a CompCert C abstract syntax tree (AST) annotated by types. In this phase, some simplifications to the original C text are performed to better fit the CompCert C language. Some are mere cleanups, such as collapsing multiple declarations of the same variable. Others are source-to-source transformations, such as pulling block-local `static` variables to global scope, renaming them if needed to keep names unique. (CompCert C has no notion of local `static` variable.) Some of these source-to-source transformations are optional and controlled by command-line options (see §3.2.8).
3. Verified compilation proper. From the CompCert C AST, the compiler produces an Asm code, going through 8 intermediate languages and 15 compilation passes. Asm is a language of abstract syntax for assembly language; it exists in three different versions, one for PowerPC, one for ARM, another for IA32. The 8 intermediate languages bridge the semantic gap between C and assembly, progressively exposing an increasing machine-like view of the program. Each of the 15 passes performs either translation to a lower-level language (re-expressing high level construct into lower-level constructs), or optimizations (rewriting the code so as to improve its performance), or both at the same time. (For more details on the passes and the intermediate languages, see Leroy [6, 7].)
4. Production of textual assembly code, followed by assembling and linking. The latter two passes are performed by an external assembler and an external linker, not part of the CompCert distribution.

As shown in Figure 1.1, only phase 3 (from CompCert C AST to Asm AST) and the parser in phase 2 are formalized and proved correct in Coq. One reason is that some of the other phases lack a mathematical spec-

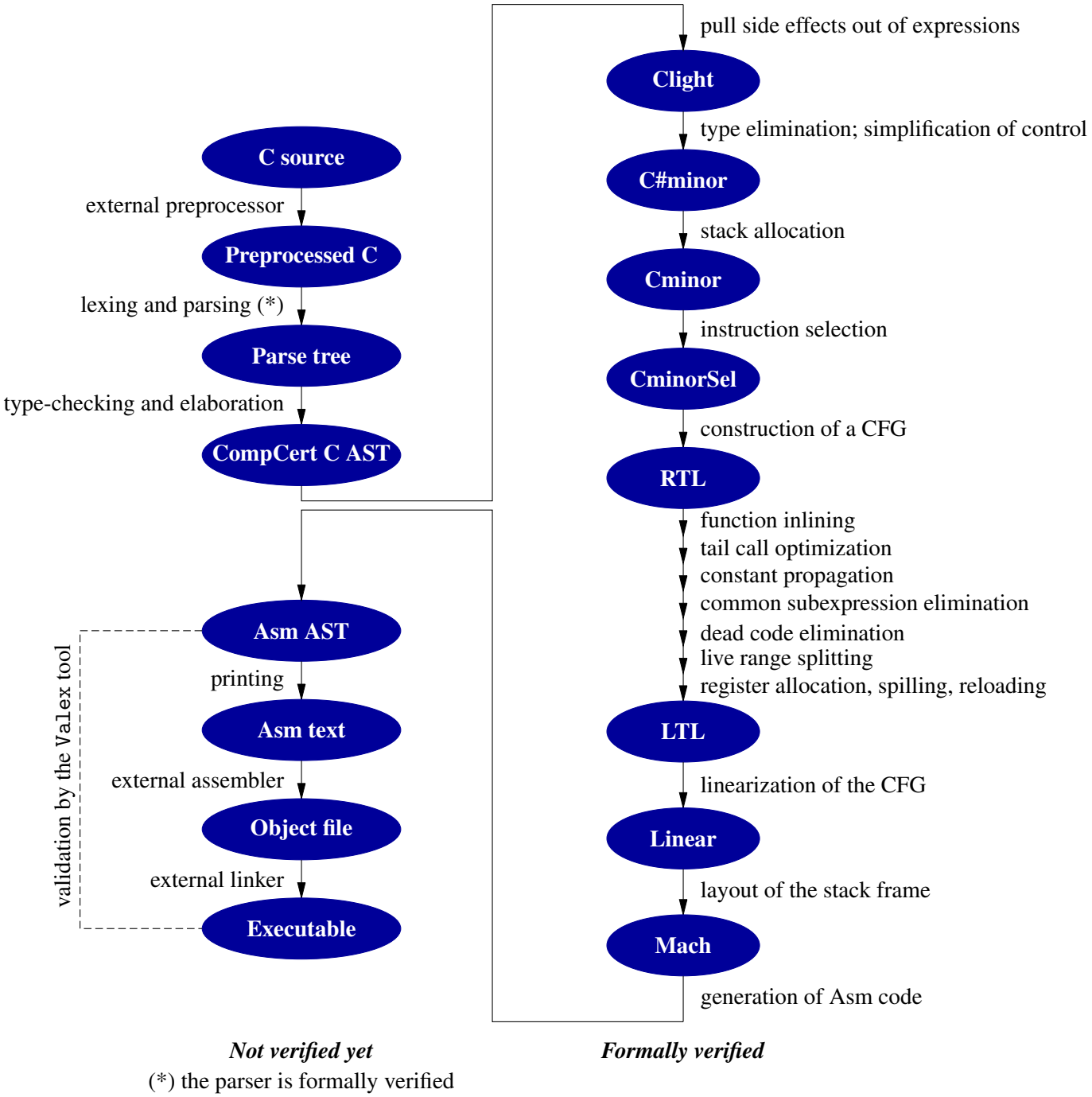

Figure 1.1: General structure of the CompCert C compiler

ification, making it impossible to state, let alone prove, a correctness theorem about them. This is typically the case for the preprocessing phase 1. Another reason is that the CompCert effort is still ongoing, and priority was given to the formal verification of the delicate compilation passes, especially of optimizations, which are all part of the verified phase 3. Future evolutions of CompCert will move more of phase 2 (unverified simplifications) into the verified phase 3. For phase 4 (assembly and linking), we have no formal guarantees yet, but the Valex tool, available from AbsInt, provides additional assurance via *a posteriori* validation of the executable produced by the external assembler and linker.

The main optimizations performed by CompCert are:

- Register allocation using graph coloring and iterated register coalescing, to keep local variables and temporaries in processor registers as much as possible.
- Instruction selection, to take advantage of combined instructions provided by the target architecture (such as "rotate and mask" on PowerPC, or the rich addressing modes of IA32).
- Constant propagation, to pre-evaluate constant computations at compile time.
- Common subexpression elimination, to avoid redundant recomputations and reuse previously-computed results instead.
- Dead code elimination, to remove useless arithmetic operations and memory loads and stores.
- Function inlining, to avoid function call overhead for functions declared `inline`.
- Tail call elimination, to implement tail recursion in constant stack space.

Loop optimizations are not performed yet.

1.4 CompCert C in practice

1.4.1 Supported target platforms

CompCert C provides 3 code generators for the following architectures:

- PowerPC 32 bits and 64 bits (in 32-bit pointers mode);
- ARM v6 and above with VFP coprocessor;
- IA32, also known as Intel/AMD x86 in 32-bit mode, with SSE2 extension (all models since Pentium 4 and Athlon 64).

For each architecture, here are the supported Application Binary Interfaces (ABI) and operating systems:

Architecture	ABI	OS
PowerPC	EABI and ELF/SVR4	Linux (all 32-bit distributions)
ARM	EABI	Debian and Ubuntu GNU/Linux, `armel` architecture
	EABI-HF	Debian and Ubuntu GNU/Linux, `armhf` architecture
IA32	ELF/SVR4	Linux (all distributions), both 32 bits (`i686`) and 64 bits (`x86_64`) in 32-bit emulation
	MacOS	MacOS 10.6 and more recent
	COFF	Microsoft Windows with the Cygwin environment

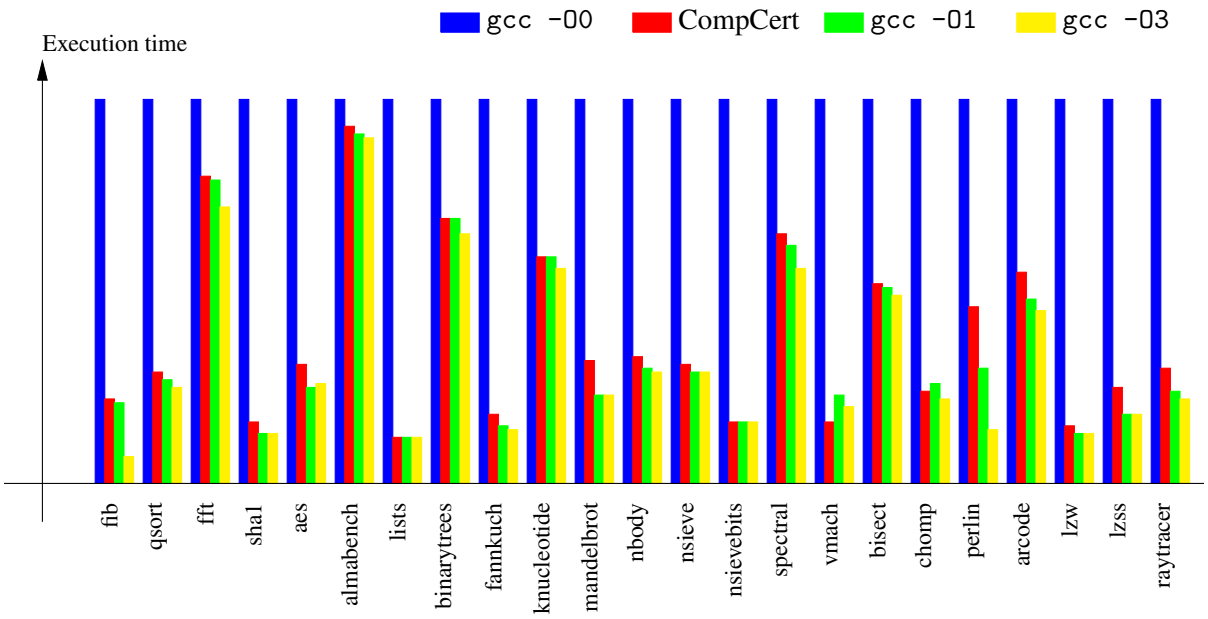

Figure 1.2: Performance of CompCert-generated code relative to GCC 4.1.2-generated code on a Power7 processor. Shorter is better. The baseline, in blue, is GCC without optimizations. CompCert is in red.

Other operating systems that follow one of the ABI above could be supported with minimal effort.

1.4.2 The supported C dialect

Chapter 5 specifies the dialect of the C language that CompCert C accepts as input language. In summary, CompCert C supports all of ISO C 99 [4], with the following exceptions:

- `switch` statements must be structured as in MISRA-C [1]; unstructured `switch`, as in Duff's device, is not supported.
- Variable-length arrays are not supported.
- `longjmp` and `setjmp` are not guaranteed to work.

Consequently, CompCert supports all of the MISRA-C 2004 subset of C, plus many features excluded by MISRA-C, such as recursive functions and dynamic heap memory allocation.

Some extensions to ISO C 99 are supported:

- The `_Alignof` operator and the `_Alignas` attribute from ISO C 2011.
- Pragmas and attributes to control alignment and section placement of global variables.

1.4.3 Performance of the generated code

On PowerPC and ARM, the code generated by CompCert runs at least twice as fast as the code generated by GCC without optimizations (`gcc -O0`), and approximately 10% slower than GCC 4 at optimization

level 1 (`gcc -O1`), 15% slower at optimization level 2 (`gcc -O2`) and 20% slower at optimization level 3 (`gcc -O3`). These numbers were obtained on the homemade benchmark mix shown in Figure 1.2. By lack of aggressive loop optimizations, performance is lower on HPC codes involving lots of matrix computations.

The IA32 architecture, with its paucity of registers and its inefficient calling conventions, is not a good fit for the CompCert compilation model. This results in performance approximately 20% slower than GCC 4 at optimization level 1.

1.4.4 ABI conformance and interoperability

CompCert attemps to generate object code that respects the Application Binary Interface of the target platform and that can, therefore, be linked with object code and libraries compiled by other C compilers. It succeeds to a large extent, as summarized in the following two tables.

Data representation and memory layout:

Data type	ARM	IA32	PowerPC
Not containing `long double` FP numbers	✔	✔	✔
Containing `long double` FP numbers	✔	✘	✘

Calling conventions (passing function arguments and returning function results):

Type of argument or result	ARM EABI	ARM HF	IA32	PowerPC
Scalar types other than `long double`	✔	✔	✔	✔
`long double`	✔	✔	✘	✘
Struct and union types other than the below	✔	✔	✔	✔
Struct types composed of 1 to 4 FP numbers	✔	✘	✔	✔

Here is a more detailed description of the ABI incompatibilities mentioned above:

- On IA32 and PowerPC, CompCert maps the `long double` type to 64-bit FP numbers, while the IA32 ABI mandates 80-bit FP numbers and the PowerPC EABI mandates 128-bit FP numbers. This causes ABI incompatibilities for the layout of structs having fields of type `long double`, as well as for passing function arguments or returning function results of type `long double`.
- On ARM with the "hard floating-point" variant of EABI, an incompatibility occurs when values of `struct` types are passed as function arguments or results, in the case where these values are composed of 1 to 4 floating-point numbers. The hard floating-point EABI uses 1 to 4 VFD registers to pass these structs as function arguments or return values, while CompCert uses integer registers or memory locations as in the default, "soft floating-point" EABI.

Layout of bit-fields in `struct` types: Several incompatibilities with the ELF ABIs are known for bitfields of type `_Bool`, `char` or `short`, and also for bitfields of type `int` that could share storage with a regular field of type `char` or `short`. If the structure contains only bitfields of type `int` and regular fields of type `int` or bigger, the layout conforms to the ELF ABI.

Chapter 2

Installation instructions

This chapter explains how to install CompCert C.

2.1 Obtaining CompCert C

CompCert C is distributed in source form. It can be freely downloaded from

http://compcert.inria.fr/download.html

The public release above can be used freely for evaluation, research and educational purposes, but commercial uses require purchasing a license from AbsInt (http://www.absint.com/). See the license conditions at http://compcert.inria.fr/doc/LICENSE for more details.

2.2 Prerequisites

The following software components are required to build, install and run CompCert C.

A C compiler: either GNU GCC version 3 or 4, or Windriver Diab C 5
: CompCert C provides its own core compiler, of course, but relies on an external toolchain for preprocessing, assembling and linking. For simplicity, the external preprocessor, assembler and linker are invoked through the `gcc` driver command (for GCC) or `dcc` driver command (for Diab C). It is recommended to use GCC version 4.

  Cross-compilation (e.g. generating PowerPC code from a IA32 host) is possible but requires the installation of a matching GCC or Diab cross compiler and cross libraries.

  For a Debian or Ubuntu GNU/Linux host, install the `gcc` package. For a Microsoft Windows host, install the Cygwin development environment from http://www.cygwin.com/. For a Mac OS host, install the XCode development tools as found on the distribution media or at http://developer.apple.com/.

The Coq proof assistant, version 8.4pl1 to 8.4pl6
: Coq is free software, available from `http://coq.inria.fr/` and also as precompiled packages in several Linux distributions and in MacPorts for MacOS X.

The OCaml functional language, version 4.02 or later
: OCaml is free software, available from `http://caml.inria.fr/`. The OPAM package manager (`http://opam.ocamlpro.com/`) provides a convenient way to install OCaml and its companion tools. OCaml is also available as precompiled packages in most Linux distributions, in MacPorts for MacOS X, and in Cygwin for Windows.

The Menhir parser generator, version 20151110 or later
: Menhir is free software, available from `http://gallium.inria.fr/~fpottier/menhir/`, and prepackaged in OPAM, MacPorts, and several Linux distributions.

Standard C library and header files
: CompCert C does not provide its own implementation of the C standard library, relying on the standard library and header files of the host system.

  For a Debian or Ubuntu GNU/Linux host, install the `libc6-dev` packages. If you are running a 64-bit version of Debian or Ubuntu, also install `libc6-dev-i386`.

  Under MacOS, install the XCode programming environment version 5.0 or later, including the optional command-line tools. With earlier versions of XCode, some standard C include files in `/usr/include/` contain GCC-isms that cause errors when compiling with CompCert. Symptoms include references to undefined types `uint16_t` and `uint32_t`, or a type error when using the `assert` macro. The recommended solution is to upgrade to a more recent XCode.

2.3 Installation

Unpacking Unpack the `.tgz` archive from a terminal window:

```
tar xzf compcert-version-number.tgz
cd compcert-version-number
```

Configuration Run the `configure` script with appropriate options:

`./configure` [*option* ...] *target*

The mandatory *target* argument identifies the target platform. It must be one of the following:

`ppc-linux`	PowerPC, Linux
`ppc-eabi`	PowerPC, EABI, with GNU or Unix tools
`ppc-eabi-diab`	PowerPC, EABI, with Diab tools
`arm-eabi`	ARM, EABI, default calling conventions
`arm-linux`	synonymous for `arm-eabi`
`arm-eabihf`	ARM, EABI, hard floating-point calling conventions
`arm-hardfloat`	synonymous for `arm-eabihf`
`ia32-linux`	IA32 (x86 32 bits), Linux
`ia32-bsd`	IA32 (x86 32 bits), BSD
`ia32-macosx`	IA32 (x86 32 bits), MacOS X
`ia32-cygwin`	IA32 (x86 32 bits), Cygwin environment under Windows

See §1.4.1 for more information on the supported platforms. For ARM targets, the `arm-` prefix can be refined into:

`armv6-`	ARMv6 architecture with VFPv2 coprocessor
`armv7a-`	ARMv7-A architecture with VFPv3-D16 coprocessor
`armv7r-`	ARMv7-R architecture with VFPv3-D16 coprocessor
`armv7m-`	ARMv7-M architecture with VFPv3-D16 coprocessor

The default `arm-` prefix corresponds to `armv7a-`.

For PowerPC targets, the `ppc-` prefix can be refined into:

`ppc64-`	PowerPC 64 bits
`e5500-`	Freescale e5500 core (PowerPC 64 bits with EREF extensions)

The default `ppc-` prefix corresponds to PowerPC 32 bits.

The `configure` script recognizes the following options:

`-bindir` *dir*
: Install the compiler's executable `ccomp` in directory *dir*. The default location is `/usr/local/bin`.

`-libdir` *dir*
: Install the compiler's supporting library and header files in directory *dir*. The default location is `/usr/local/lib/compcert`.

`-prefix` *dir*
: Equivalent to "`-bindir` *dir*`/bin -libdir` *dir*`/lib/compcert`".

`-toolprefix` *pref*
: Prefix the name of the external C compiler driver (`gcc` or `dcc`) with *pref*. This option is particularly useful if a cross-compiler is used. For example:

  - If the `gcc` executable to use is not in the search path, but in the directory `/opt/local/powerpc-linux-gnu`, give the option `-toolprefix /opt/local/powerpc-linux-gnu/` (note the trailing slash).
  - If the `gcc` executable to use is in the search path but is called `powerpc-linux-gnu-gcc`, give the option `-toolprefix powerpc-linux-gnu-` (note the trailing dash).

`-no-runtime-lib`
: Do not compile, install, and use the `libcompcert` library that provides helper functions for 64-bit integer arithmetic. By default, this library is installed and linked with CompCert-generated executables. If it is not, some operations involving 64-bit integers (e.g. division, remainder, conversion to/from floating-point numbers) will not work.

`-no-standard-headers`
: Do not install the CompCert-specific standard header files. By default, the following standard header files are installed and used: `<float.h>`, `<stdarg.h>`, `<stdbool.h>`, `<stddef.h>`, and `<varargs.h>`.

After successful completion, the `configure` script generates a configuration file `Makefile.config` and prints a summary of the configuration. If anything looks wrong, re-run `./configure` with different options, or edit `Makefile.config` by hand.

Building the system From the same directory where `./configure` was executed, issue the command

```
make all
```

or, on a *N*-core machine, to take advantage of parallel compilation:

```
make -j N all
```

This re-checks all the Coq proofs, then extracts Caml code from the Coq specification and combines it with supporting hand-written Caml code to generate the executable for CompCert. This step can take about 30 minutes on a recent machine with a single core, but less if several cores are used.

Installation CompCert is now ready to be installed. This will create the `ccomp` command (documented in chapter 3) in the binary directory selected during configuration, and install supporting `.h` and `.a` files in the library directory if needed. Become superuser if necessary and do

```
make install
```

Chapter 3

Using the CompCert C compiler

This chapter explains how to invoke the CompCert C compiler and describes its command-line interface.

3.1 Overview

The CompCert C compiler is a command-line executable named `ccomp`. Its interface similar to that of many other C compilers. An invocation of `ccomp` is of the form

`ccomp` [*option* ...] *input-file* ...

By default, every input file is processed in sequence to obtain a compiled object file; then, all compiled object files thus obtained, plus those given on the command line, are linked together to produce an executable program. The name of the generated executable is controlled with the `-o` option; it is `a.out` if no option is given. The `-c`, `-S` and `-E` options allow the user to stop this process at an intermediate stage. For example, the `-c` option stops compilation before invoking the linker, leaving the compiled object files (with extension `.o`) as the final result. Likewise, the `-S` option stops compilation before invoking the assembler, leaving assembly files with the `.s` extension as the final result.

CompCert C accepts several kinds of input files:

`.c` C source files
: Arguments ending in `.c` are taken to be source files written in C. Given the file *x*`.c`, the compiler preprocesses the file, then compiles it to assembly language, then invokes the assembler to produce an object file named *x*`.o`.

`.i` or `.p` C source files that should not be preprocessed
: Arguments ending in `.i` or `.p` are taken to be source files written in C and already preprocessed, or not using any preprocessing directive. These files are not run through the preprocessor. Given the file *x*`.i` or *x*`.p`, the compiler compiles it to assembly language, then invokes the assembler to produce an object file named *x*`.o`.

`.s` Assembly source files
: Arguments ending in `.s` are taken to be source files written in assembly language. Given the file

x`.s`, the compiler passes it to the assembler, producing an object file named *x*`.o`.

`.S` Assembly source files that must be preprocessed
: Arguments ending in `.S` are taken to be source files written in assembly language plus C-style macros and preprocessing directives. Given the file *x*`.S`, the compiler passes the file through the C preprocessor, then through the assembler, producing an object file named *x*`.o`.

`.o` Compiled object files
: Arguments ending in `.o` are taken to be object files obtained by a prior run of compilation. They are passed directly to the linker.

`.a` Compiled library files
: Arguments ending in `.a` are taken to be libraries. Like `.o` files, they are passed directly to the linker.

`-l`*lib* Compiled library files
: Arguments starting in `-l` are taken to be system libraries. They are passed directly to the linker.

`.cm` Cminor source files
: Arguments ending in `.cm` are taken to be source files written in Cminor, an intermediate language of the CompCert C compiler. They are subject to compilation to assembly, then assembling. The Cminor language is not documented. Cminor input files are of interest only to the developers of the CompCert compiler and of front-end compilers that reuse CompCert's back-end.

Here are some examples of use. To compile the single-file program `src.c` and create an executable called `exec`, just do

```
ccomp -o exec src.c
```

To compile a two-file program `src1.c` and `src2.c`, do

```
ccomp -c src1.c
ccomp -c src2.c
ccomp -o exec src1.o src2.o
```

To see the generated assembly code for `src1.c`, do

```
ccomp -S src1.c
```

3.2 Options

The `ccomp` command recognizes the following options. All options start with a minus sign (`-`).

3.2.1 Options controlling the output

`-c` Compile or assemble the source files, but do not link. The final output is an object file *x*`.o` for every input file *x*`.c` or *x*`.s` or *x*`.S` or *x*`.cm`. The name of the output can be controlled using the `-o` option.

`-S` Compile the source files all the way to assembly, but do not assemble nor link. The final output is an assembly file *x*`.s` for every input file *x*`.c` or *x*`.cm`. The name of the output can be controlled using the `-o` option.

`-E` Stop after the preprocessing stage; do not compile nor link. The output is preprocessed C source code for every input file *x*`.c`. If no `-o` option is given, the preprocessed code is sent to the standard output. If a `-o` option is given, the preprocessed code is saved to the indicated file.

`-o` *file* Generate the final output in file named *file*. If none of the `-c`, `-S` or `-E` options are given, the final output is the executable program produced during the linking phase. The `-o` *file* option causes this executable to be placed in *file*. Otherwise, it is placed in file `a.out` in the current directory.

If the `-c` option is given along with the `-o` option, the object file generated by the compilation of the source file given on the command line is saved in *file*. If no `-o` option is given, it is generated in the current directory with extension `.o`.

If the `-S` option is given along with the `-o` option, the assembly file generated by the compilation of the source file given on the command line is saved in *file*. If no `-o` option is given, it is generated in the current directory with extension `.s`.

If the `-E` option is given along with the `-o` option, the result of preprocessing the source file given on the command line is saved in *file*. If no `-o` option is given, the preprocessed result is sent to standard output.

When the `-o` option is given in conjunction with one of the `-c`, `-S` or `-E` options, there must be only one source file given on the command line.

`-sdump` In addition to the outputs normally produced by Compcert, generate a *x*`.sdump` file for every *x*`.c` input file. The `.sdump` file contains the abstract syntax tree for the generated assembly code, in JSON format. The `.sdump` files can be used by the `Valex` validation tool distributed by AbsInt.

3.2.2 Preprocessing options

`-I`*dir* Add directory *dir* to the list of directories searched for included `.h` files.

`-D`*name* Define *name* as a macro that expands to "`1`". This is equivalent to adding a line "`#define` *name* `1`" at the beginning of the source file.

`-D`*name*`=`*def*
Define *name* as a macro that expands to *def*. This is equivalent to adding a line "`#define` *name def*" at the beginning of the source file. A parenthesized list of parameters can occur between *name* and the `=` sign, to define a macro with parameters. For example, `-DF(x,y)=x` is equivalent to adding a line "`#define F(x,y) x`" at the beginning of the source file.

`-U`*name* Erase any previous definition of the macro *name*, either built-in or performed via a previous `-D` option. This is equivalent to adding a line "`#undef` *name*" at the beginning of the source file.

`-Wp,`*opt* Pass *opt* as an option to the preprocessor. If *opt* contains commas (`,`), it is split into multiple options at the commas.

The macro `__COMPCERT__` is always predefined, with expansion "`1`".

The preprocessing options above can either be concatenated with their arguments (as shown above) or separated from their arguments by spaces.

3.2.3 Optimization options

`-O` (default mode)
: Optimize the code with the objective of improving execution speed. This is the default.

`-O1` / `-O2` / `-O3`
: Synonymous for `-O` (optimize for speed).

`-Os`
: Optimize the code with the objective of reducing the size of the executable. CompCert's optimizations improve both execution speed and code size, but some of the code generation heuristics in `-O` mode favor speed over compactness. The `-Os` option biases these heuristics in the other direction, favoring compactness over speed.

`-O0`
: Turn most optimizations off. This produces slower code but reduces compilation times. Equivalent to `-fno-const-prop -fno-cse -fno-redundancy -fno-tailcalls`. The only optimizations performed are 1- integer constant propagation within expressions, 2- register allocation, and 3- dead code elimination.

`-fconst-prop` / `-fno-const-prop`
: Turn on/off the constant propagation optimization.

`-fcse` / `-fno-cse`
: Turn on/off the elimination of common subexpressions.

`-fredundancy` / `-fno-redundancy`
: Turn on/off the elimination of redundant computations and useless memory stores.

`-ftailcalls` / `-fno-tailcalls`
: Turn on/off the optimization of function calls in tail position.

`-ffloat-const-prop` *N*
: This option controls whether and how floating-point constants are propagated at compile-time. The constant propagation optimization consists in evaluating, at compile-time, arithmetic and logical operations whose arguments are constants, and replace these operations by the constants just obtained. A constant, here, is either an integer or float literal, the initial value of a `const` variable, or, recursively, the result of an arithmetic or logical operation itself contracted by constant propagation. The `-ffloat-const-prop` controls how floating-point constants are propagated and translated.

  `-ffloat-const-prop 2` (default mode)
  : Full propagation of floating-point constants. Float arithmetic is performed by the compiler in IEEE double precision format, with round-to-nearest mode. This propagation is correct only if the program does not change float rounding mode at run-time, leaving it in the default round-to-nearest mode.

`-ffloat-const-prop 0`
: No propagation of floating-point constants. This option should be given if the program changes the float rounding mode during its execution.

`-ffloat-const-prop 1`
: Propagate floating-point constants, assuming round-to-nearest mode, but only for arguments of integer-valued operations such as float comparisons and float-to-integer conversions. In other words, floating-point constants are propagated, but no new floating-point constants are inserted in the generated assembly code. This option is useful for some processor configurations where floating-point constants are stored in slow memory and therefore loading a floating-point constant from memory can be slower than recomputing it at run-time.

3.2.4 Code generation options

`-falign-functions` *N*
: Force the entry point to any compiled function to be aligned on an *N* byte boundary. The default alignment for function entry points is 16 bytes for the IA32 target and 4 bytes for the ARM and PowerPC targets.

`-falign-branch-targets` *N*
: In the generated assembly code, align the targets of branch instructions to a multiple of *N* bytes. Only branch targets that cannot be reached by fall-through execution are thus aligned.

`-falign-cond-branches` *N*
: This option is specific to the PowerPC target. It causes conditional branch instructions (`bc`) to be aligned to a multiple of *N* bytes in the generated assembly code.

`-fsmall-data` *N*
: This option is specific to the PowerPC EABI target platform. It causes global variables of size less than or equal to *N* bytes and of non-`const` type to be placed in the small data area (SDA) of the generated executable, and to be addressed by 16-bit offsets relative to the SDA register. This is more efficient than the default absolute addressing used to access global variables. If no `-fsmall-data` option is given, *N* is taken to be zero by default, turning off the placement of variables in the small data area.

`-fsmall-const` *N*
: Similar to `-fsmall-data` *N*, but governs the placement of `const` global variables in the small data area.

`-Wa,`*opt* Pass *opt* as an option to the assembler. If *opt* contains commas (`,`), it is split into multiple options at the commas.

`-fno-fpu`
: Prevent the generation of floating-point or SSE2 instructions for assignments between composites (structures or unions) and for the `__builtin_memcpy_aligned` built-in function.

3.2.5 Target processor options

`-mthumb`
: This option applies only to the ARM port of CompCert. It instructs CompCert to generate code using the Thumb2 instruction encoding. This is the default if CompCert was configured for the ARMv7R profile.

`-marm`
: This option applies only to the ARM port of CompCert. It instructs CompCert to generate code using the classic ARM instruction encoding. This is the default if CompCert was configured for a profile other than ARMv7R.

3.2.6 Debugging options

`-g`
: Generate debugging information in DWARF format. Programs compiled and linked with the `-g` option can be debugged using a debugger such as GDB. For the PowerPC/Diab and PowerPC/GNU platforms, full debugging information is produced. For the ARM and IA32 platforms, the information currently generated is restricted to source file names and line numbers, plus call frame information.

3.2.7 Linking options

`-l`*x*
: Link with the system library `-l`*x*. The linker searches a standard list of directories for the file `lib`*x*`.a` and links it.

`-L`*dir*
: Add directory *dir* to the list of directories searched for `-l`*lib* libraries.

`-Wl,`*opt*
: Pass *opt* as an option to the linker. If *opt* contains commas (`,`), it is split into multiple options at the commas.

3.2.8 Language support options

The formally-verified part of the CompCert compiler lacks several features of the C language. Some of these features can be simulated by prior source-to-source transformations, performed during the elaboration phase, before entering the formally-verified part of the compiler. The following language support options control which features are simulated this way. Note that these source-to-source transformations are not formally verified yet and cannot be absolutely trusted. For high-assurance software, it is recommended to deactivate them entirely (option `-fnone`) or to review the C source code after these transformations (option `-dc`).

`-fbitfields`
: Support bit-fields in structure declarations. Consecutive bit-fields are grouped into integer fields of appropriate sizes. Accesses to bit-fields are replaced by bit shifting and masking over the generated integer fields.

`-fno-bitfields` (default)
: Reject bit-fields in structure declarations.

`-flongdouble`
: Accept the `long double` type and treat it as synonymous for the `double` type, that is, double-precision IEEE 754 floats. This implementation of `long double` is correct according to the C standards, but does not conform to the ABIs of the target platforms. In other terms, the code generated by CompCert in `-flongdouble` mode may not interoperate with code generated by an ABI-conformant compiler.

`-fno-longdouble` (default)
: Reject all occurrences of the `long double` type.

`-fpacked-structs`
: Enable the programmer to control the alignment of `struct` types and of their individual fields, via the non-standard `packed` type attribute (§6.2).

`-fno-packed-structs` (default)
: Ignore the `packed` type attribute, and always use the field alignment rules specified by the ABI of the target platform.

`-fstruct-passing`
: Support functions that take parameters and return results of composite types (`struct` or `union` types) by value.

`-fno-struct-passing` (default)
: Reject all functions that take arguments or return results of `struct` or `union` types.

`-funprototyped` (default)
: Support the declaration and invocation of functions declared without function prototypes ("old-style" unprototyped functions).

`-fno-unprototyped`
: Reject all functions that are not declared with a function prototype.

`-fvararg-calls` (default)
: Support defining functions with a variable number of arguments, and calling such functions. A typical example is the `printf` function and its variants from the C standard library.

`-fno-vararg-calls`
: Reject all attempts to define or invoke a variable-argument function.

`-finline-asm`
: Activate support for inline assembly statements (see section 6.5). Indiscriminate use of this statement can ruin all the semantic preservation guarantees of CompCert.

`-fno-inline-asm` (default)
: Reject all uses of `asm` statements.

`-fall` Activate all language support options above.

`-fnone` Turn off all language support options above.

As listed in the description above, the `-fvararg-calls` and `-funprototyped` language support options are turned on by default, and all other are turned off by default.

3.2.9 Tracing options

The following options direct the compiler to save the file being compiled into files at various stages of compilation. The three most useful tracing options are:

`-dparse`
: Save the C file after parsing, elaboration, and source-to-source transformations as described in section "Language support options". If the source file is named *x*`.c`, the intermediate form is saved in file *x*`.parse.c`, in C syntax. This intermediate form is useful to review the effect of the unverified source-to-source transformations.

`-dc` Save the generated CompCert C code, just before entering the verified part of the compiler. If the source file is named *x*`.c`, the intermediate form is saved in file *x*`.compcert.c`, in C syntax. This intermediate form is useful in conjunction with the reference interpreter, because it represents the program exactly as it is interpreted.

`-dasm` Save the generated assembly code, just before calling the assembler. If the source file is named *x*`.c`, the assembly code is saved in file *x*`.s`. Unlike with option `-S`, compilation does not stop here and continues with assembling and linking.

The remaining tracing options are of interest mainly to the CompCert developers. In the description below, we assume that the source file is named *x*`.c`.

`-dclight`
: Save generated Clight intermediate code in file *x*`.light.c`, in C-like syntax.

`-dcminor`
: Save generated Cminor intermediate code in file *x*`.cm`.

`-drtl` Save generated RTL form at successive stages of optimization in files *x*`.rtl.0`, *x*`.rtl.1`, etc.

`-dltl` Save LTL form after register allocation in *x*`.alloc.ltl`

`-dmach` Save Mach form after stack layout in file *x*`.mach`

3.2.10 Miscellaneous options

`-v` Before every invocation of an external command (preprocessor, assembler, linker), print the command and its arguments.

`-timings`
: Measure and display the time spent in various compilation passes.

`-stdlib` *dir*
: Specify the directory *dir* containing the CompCert C specific library and header files. This option

is useful in the rare case where the user needs to override the default location specified at CompCert installation time.

`-conf` *name*
: Specify a configuration file different from the default `compcert.ini` automatically generated when building CompCert from sources.

Chapter 4

Using the CompCert C interpreter

This chapter describes the CompCert C reference interpreter and how to invoke it.

4.1 Overview

The CompCert C reference interpreter executes the given C source file by interpretation, displaying the outcome of the execution (normal termination or aborting on an undefined behavior), as well as the observable effects (e.g. `printf` calls) performed during the execution.

The reference interpreter is faithful to the formal semantics of the CompCert C language: all the behaviors that it displays are possible behaviors according to the formal semantics. In particular, the reference interpreter immediately reports and stops when an undefined behavior of the C source program is encountered. This is not the case for the machine code generated by compiling this program: once the undefined behavior is triggered, the machine code can crash, but it can also continue with any other behavior.

The primary use of the reference interpreter is to check whether a particular run of a C program exhibits behaviors that are undefined in CompCert C. If it does, something is wrong with the program, and the program should be fixed. The reference interpreter can also be useful to familiarize oneself with the CompCert C formal semantics, and validate it experimentally by testing.

The reference interpreter is presented as a special mode, `-interp`, of the `ccomp` command-line executable of the CompCert C compiler. An invocation of the reference interpreter is of the form

`ccomp -interp` [*option* ...] *input-file*`.c`

The input C file is preprocessed, elaborated to the CompCert C subset language, then interpreted and its observable effects displayed.

4.2 Limitations

The following limitations apply to the C source files that can be interpreted.

1. The C source file must contain a complete, standalone program, including in particular a `main` function.
2. The only external functions available to the program are

`printf`	to display formatted text on standard output
`malloc`	to dynamically allocate memory
`free`	to free memory allocated by `malloc`
`__builtin_annot`	to mark execution points
`__builtin_annot_val`	(likewise)
`__builtin_fabs`	floating-point absolute value

3. The `main` function must be declared with one of the two types allowed by the C standards, namely:

```
int main(void) { ... }
int main(int argc, char ** argv) { ... }
```

4. In the second form above, `main` is called with `argc` equal to zero and `argv` equal to the `NULL` pointer. The program does not, therefore, have access to command-line arguments.

4.3 Options

The following options to the `ccomp` command apply specifically to the reference interpreter.

4.3.1 Controlling the output

By default, the reference interpreter prints whatever output is produced by the program via calls to the `printf` function, plus messages to indicate program termination as well as other observable events.

`-quiet` Do not print any trace of the execution. The only output produced is that of the `printf` calls contained in the program.

`-trace` Print a detailed trace of the execution. At each time step, the interpreter displays the expression or statement or function invocation that it is about to execute.

4.3.2 Controlling execution order

Like that of C, the semantics of CompCert C is internally nondeterministic: in general, several evaluation orders are possible for a given expression, and different orders can produce different observable behaviors for the program. By default, the interpreter evaluates C expressions following a fixed, left-to-right evaluation order.

`-random`
Randomize execution order. Instead of evaluating expressions left-to-right, the interpreter picks one evaluation order among all those allowed by the semantics of CompCert C. Interpreting the

same program in `-random` mode several times in a row can show that a program is sensitive to evaluation order.

`-all` Explore in parallel all evaluation orders allowed by the semantics of CompCert C, displaying all possible outcomes of the input program. This exploration can be very costly and is feasible only for short programs.

4.3.3 Options shared with the compiler

In addition, all the options of the CompCert C compiler are recognized (see §3.2). The ones that make sense in interpreter mode are:

- Preprocessing options (§3.2.2): `-I`, `-D`, `-U`
- Language support options (§3.2.8): `-fall`, `-fnone`, and the various `-f`*feature* and `-fno-`*feature* options.
- Tracing options (§3.2.9): `-dparse` and `-dc`. The `-dc` option is particularly useful in conjunction with the interpreter, since it saves in a readable file the exact CompCert C program that the interpreter is running.

4.4 Examples of use

4.4.1 Running a simple program

Consider the file `fact.c` containing the following program:

```
#include <stdio.h>

int fact(int n)
{
 int r = 1;
 int i;
 for (i = 2; i <= n; i++) r *= i;
 return r;
}

int main(void) {
 printf("fact(10) = %d\n", fact(10));
 return 0;
}
```

Running `ccomp -interp fact.c` produces the following output:

```
fact(10) = 3628800
Time 251: observable event:
                     extcall printf(& __stringlit_1, 3628800) -> 0
Time 256: program terminated (exit code = 0)
```

The first line is the output produced by the `printf` statement. The other three lines report the two observable effects performed by the program: first, after 251 execution steps, a call to the external function `printf`; then, after 256 execution steps, successful termination with exit code 0.

To make more sense out of the messages, we can add the `-dc` option to the command line, then look at the generated `fact.compcert.c` file, which contains the CompCert C program as the interpreter sees it:

```
unsigned char const __stringlit_1[15] = "fact(10) = %d\012";

extern int printf(unsigned char *, ...);

int fact(int n)
{
  int r;
  int i;
  r = 1;
  for (i = 2; i <= n; i++) {
    r *= i;
  }
  return r;
}

int main(void)
{
  printf(__stringlit_1, fact(10));
  return 0;
}
```

We see that the string literal appearing as first argument to `printf` was lifted outside the call and bound to a global variable `__stringlit_1`.

Interpreting `fact.c` with the `-trace` option, we obtain a long and detailed trace of the execution, of which we show only a few lines:

```
Time 0: calling main()
--[step_internal_function]-->
Time 1: in function main, statement
  printf(__stringlit_1, fact(10)); return 0;
--[step_seq]-->
Time 2: in function main, statement printf(__stringlit_1, fact(10));
--[step_do_1]-->
Time 3: in function main, expression printf(__stringlit_1, fact(10))
--[red_var_global]-->
Time 4: in function main, expression printf(<loc __stringlit_1>, fact(10))
--[red_rvalof]-->
Time 5: in function main, expression printf(<ptr __stringlit_1>, fact(10))
[...]
Time 254: in function main, statement return 0;
--[step_return_1]-->
Time 255: in function main, expression 0
--[step_return_2]-->
Time 256: returning 0
```

```
Time 256: program terminated (exit code = 0)
```

The labels on the arrows (e.g. `step_do_1`) are the names of the reduction rules being applied. The reduction rules can be found in the CompCert C formal semantics (module `Csem.v` of the CompCert sources).

4.4.2 Exploring undefined behaviors

Consider the file `outofbounds.c` containing the following C code:

```
#include <stdio.h>

int x[2] = { 12, 34 };
int y[1] = { 56 };

int main(void)
{
  int i = 65536 * 65536 + 2;
  printf("i = %d\n", i);
  printf("x[i] = %d\n", x[i]);
  return 0;
}
```

Running it with `ccomp -interp -trace outofbounds.c`, we obtain the following trace, shortened to focus on the interesting parts:

```
[...]
Time 3: in function main, expression i = 65536 * 65536 + 2
--[red_var_local]-->
Time 4: in function main, expression <loc i> = 65536 * 65536 + 2
--[red_binop]-->
Time 5: in function main, expression <loc i> = 0 + 2
--[red_binop]-->
Time 6: in function main, expression <loc i> = 2
[...]
Time 27: in function main, expression printf(<ptr __stringlit_2>, *<ptr x+8>)
--[red_deref]-->
Time 28: in function main, expression printf(<ptr __stringlit_2>, <loc x+8>)
Stuck state: in function main, expression printf(<ptr __stringlit_2>, <loc x+8>)
Stuck subexpression: <loc x+8>
ERROR: Undefined behavior
```

We see that the expression `65536 * 65536 + 2` caused an overflow in signed arithmetic. This is an undefined behavior according to the C standards, but the CompCert C semantics fully defines this behavior as computing the result modulo 2^{32}. Therefore, the expression evaluates to 2, without error.

On the other hand, the array access `x[i]` triggers an undefined behavior, since `i`, which is equal to 2, falls outside the bounds of `x`, which is of size 2. The interpreter gets stuck when trying to dereference the l-value `x + 2` (printed as `<loc x+8>` to denote the location 8 bytes from that of variable `x`).

4.4.3 Exploring evaluation orders

Consider the following C program in file `abc.c`:

```
#include <stdio.h>

int a() { printf("a "); return 1; }
int b() { printf("b "); return 2; }
int c() { printf("c "); return 3; }

int main () {
  printf("%d\n", a() + (b() + c()));
  return 0;
}
```

Interpreting it multiple times with `ccomp -interp -quiet -random abc.c` produces various outputs among the following six possibilities:

```
a b c 6
a c b 6
b a c 6
b c a 6
c a b 6
c b a 6
```

Indeed, according to the C standards and to the CompCert C formal semantics, the calls to functions `a()`, `b()` and `c()` can occur in any order.

On the `abc.c` example, exploring all evaluation orders with the `-all` option results in a messy output. Let us do this exploration on a simpler example (file `nondet.c`):

```
int x = 0;
int f() { return ++x; }
int main() { return f() - x; }
```

Running `ccomp -interp -all nondet.c` shows the two possible outcomes for this program:

```
Time 17: program terminated (exit code = 0)
Time 17: program terminated (exit code = 1)
```

The first outcome corresponds to calling `f` first, setting `x` to 1 and returning 1, then reading `x`, obtaining 1. The second outcome corresponds to the other evaluation order: `x` is read first, producing 0, then `f` is called, returning 1.

If we add the `-trace` option, we can follow the breadth-first exploration of evaluation states. At any given time, up to three different states are reachable.

```
State 0.1:  calling main()
Transition state 0.1 --[step_internal_function]--> state 1.1
State 1.1:  in function main, statement return f() - x;
Transition state 1.1 --[step_return_1]--> state 2.1
State 2.1:  in function main, expression f() - x
Transition state 2.1 --[red_var_global]--> state 3.1
Transition state 2.1 --[red_var_global]--> state 3.2
```

```
State 3.1:  in function main, expression <loc f>() - x
State 3.2:  in function main, expression f() - <loc x>
Transition state 3.1 --[red_rvalof]--> state 4.1
Transition state 3.1 --[red_var_global]--> state 4.2
Transition state 3.2 --[red_var_global]--> state 4.2
Transition state 3.2 --[red_rvalof]--> state 4.3
State 4.1:  in function main, expression <ptr f>() - x
State 4.2:  in function main, expression <loc f>() - <loc x>
State 4.3:  in function main, expression f() - 0
Transition state 4.1 --[red_call]--> state 5.1
Transition state 4.1 --[red_var_global]--> state 5.2
Transition state 4.2 --[red_rvalof]--> state 5.2
Transition state 4.2 --[red_rvalof]--> state 5.3
Transition state 4.3 --[red_var_global]--> state 5.3
State 5.1:  calling f()
State 5.2:  in function main, expression <ptr f>() - <loc x>
State 5.3:  in function main, expression <loc f>() - 0
[...]
```

Chapter 5

The CompCert C language

This chapter describes the dialect of the C programming language that is implemented by the CompCert C compiler and reference interpreter. It follows very closely the ISO C99 standard [4]. A few features of C99 are not supported at all; some other are supported only if the appropriate language support options are selected on the command line. On the other hand, some extensions to C99 are supported, borrowed from the ISO C2011 standard [5].

In this chapter, we describe both the restrictions and the extensions of CompCert C with respect to the C99 standard. We also document how CompCert implements the behaviors specified as implementation-dependent in C99. The description follows the structure of the C99 standard document [4]. In particular, section numbers (e.g. "5.1.2.2") are those of the C99 standard document.

5. Environment

5.1.2.2 Hosted environment.

CompCert C follows the hosted environment model. The function called at program startup is named `main`. According to the formal semantics, it must be defined without parameters: `int main(void) { ... }`. The CompCert C compiler also supports the two-parameter form `int main(int argc, char *argv[])`.

5.2.1.2 Multibyte characters.

Multibyte characters in program sources are not supported.

5.2.4.2 Numerical limits.

Integer types follow the "ILP32LL64" model:

Type	Size	Range of values
`unsigned char`	1 byte	0 to 255
`signed char`	1 byte	−128 to 127
`char`	1 byte	like `signed char` on IA32
		like `unsigned char` on PowerPC and ARM
`unsigned short`	2 bytes	0 to 65535
`signed short` and `short`	2 bytes	−32768 to 32767
`unsigned int`	4 bytes	0 to 4294967295
`signed int` and `int`	4 bytes	−2147483648 to 2147483647
`unsigned long`	4 bytes	0 to 4294967295
`signed long` and `long`	4 bytes	−2147483648 to 2147483647
`unsigned long long`	8 bytes	0 to 18446744073709551615
`signed long long` and `long long`	8 bytes	−9223372036854775808 to 9223372036854775807
`_Bool`	1 byte	0 or 1

Floating-point types follow the IEEE 754-2008 standard [10]:

Type	Representation	Size	Mantissa	Exponent
`float`	IEEE 754 single precision (`binary32`)	4 bytes	23 bits	−126 to 127
`double`	IEEE 754 double precision (`binary64`)	8 bytes	52 bits	−1022 to 1023
`long double`	not supported by default; with `-flongdouble` option, like `double`			

During evaluation of floating-point operations, the floating-point format used is that implied by the type, without excess precision and range. This corresponds to a `FLT_EVAL_METHOD` equal to `0`.

6. Language

6.2 Concepts

6.2.5 Types

CompCert C supports all the types specified in C99, with the following exceptions:

- The `long double` type is not supported by default. If the `-flongdouble` option is set, it is treated as a synonym for `double`.
- Complex types (`double _Complex`, etc) are not supported.
- The result type and argument types of a function type must not be a structure or union type, unless the `-fstruct-passing` option is active (§3.2.8).
- Variable-length arrays are not supported. The size N of an array declarator $T[N]$ must always be a compile-time constant expression.

6.2.6 Representation of types

Signed integers use two's-complement representation.

6.3 Conversions

Conversion of an integer value to a signed integer type is always defined as reducing the integer value

modulo 2^N to the range of values representable by the N-bit signed integer type.

Pointer values can be converted to any pointer type. A pointer value can also be converted to the integer types `int`, `unsigned int`, `long` and `unsigned long` and back to the original pointer type: the result is identical to the original pointer value.

Conversions from `double` to `float` rounds the floating-point number to the nearest floating-point number representable in single precision. Conversions from integer types to floating-point types round to the nearest representable floating-point number.

6.4 Lexical elements

6.4.1 Keywords.

The following tokens are reserved as additional keywords:

```
_Alignas      _Alignof       __attribute__    __attribute
__const       __const__      __inline         __inline__
__restrict    __restrict__   __packed__
asm           __asm          __asm__
```

6.4.2 Identifiers

All characters of an identifier are significant, whether it has external linkage or not. Case is significant even in identifiers with external linkage. The "$" (dollar) character is accepted in identifiers.

6.4.3 Universal character names

Universal character names are supported in character constants and string literals. They are not supported within identifiers.

6.5 Expressions

CompCert C supports all expression operators specified in C99, with the restrictions and extensions described below.

Overflow in arithmetic over signed integer types is defined as taking the mathematically-exact result and reducing it modulo 2^{32} or 2^{64} to the range of representable signed integers. Bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`) over signed integer types are interpreted following two's-complement representation.

Floating-point operations round their results to the nearest representable floating point number, breaking ties by rounding to an even mantissa. If the program changes the rounding mode at run-time, it must be compiled with flag `-ffloat-const-prop 0` (§3.2.3). Otherwise, the compiler will perform compile-time evaluations of floating-point operations assuming round-to-nearest mode.

Floating-point intermediate results are computed in single precision if they are of type `float` (i.e. all arguments have integer or `float` types) and in double precision if they are of type `double` (i.e. one argument has type `double`). This corresponds to `FLT_EVAL_METHOD` equal to `0`.

An integer or floating-point value stored in (part of) an object can be accessed by any lvalue having integer or floating-point type. The effect of such an access is defined taking into account the bit-level representation of the types (two's complement for integers, IEEE 754 for floats) and the endianness of the target platform (big-endian for PowerPC, little-endian for ARM and IA32). In contrast, a pointer

value stored in an object can only be accessed by a lvalue having pointer type or 32-bit integer type (`signed int`, `unsigned int`, `signed long` and `unsigned long`). In other words, while the bit-level, in-memory representation of integers and floats is fully exposed by the CompCert C semantics, the in-memory representation of pointers is kept opaque and cannot be examined at any granularity other than a 32-bit word.

6.5.2 Postfix operators

If a member of a union object is accessed after a value has been stored in a different member of the object, the behavior is as described in the last paragraph above: the operation is well defined as long as it does not entail accessing a stored pointer value with a type other than a pointer type or a 32-bit integer type. For example, the declaration

```
union u { double d; unsigned int i[2]; unsigned char c[8]; };
```

supports accessing any member after any other member has been stored. On the other hand, consider

```
union u { char * ptr; unsigned char c[4]; };
```

If a pointer value is stored in the `ptr` member, accesses to the elements of the `c` member are not defined.

6.5.3 Unary operators

Symmetrically with the `sizeof` operator, CompCert C supports the `_Alignof` operator from C2011, which can also be written `__alignof__` as in GNU C. This operator applies either to a type or to an expression. It returns the natural alignment, in bytes, of this type or this expression's type.

The type `size_t` of the result of `sizeof` and `_Alignof` is taken to be `unsigned long`.

6.5.4 Casts

See the comments on point 6.3 ("Conversions") above concerning pointer casts and casts to signed integer types.

6.5.5 Multiplicative operators

Division and remainder are undefined if the second argument is zero. Signed division and remainder are also undefined if the first argument is the smallest representable signed integer (-2147483648 for type `int`) and the second argument is -1 (the only case where division overflows).

6.5.6 Additive operators

Adding a pointer and an integer, or subtracting an integer from a pointer, are always defined even if the resulting pointer points outside the bounds of the underlying object. The byte offset with respect to the base of the underlying object is treated modulo 2^{32}. Such out-of-bounds pointers cause undefined behavior when dereferenced or compared with other pointers.

6.5.7 Bitwise shift operators

The right shift operator `>>` applied to a negative signed integer is specified as shifting in "1" bits from the left.

6.5.8 Relational operators

Comparison between two non-null pointers is always defined if both pointers fall within the bounds of the underlying objects. Additionally, comparison is also defined if one of the pointers is "one past the end of an object" and the other pointer is identical to the first pointer or falls within the bounds of the same object. Comparison between a pointer "one past" the end of an object and a pointer within a different object is undefined behavior.

6.5.9 Equality operators

Same remark as in 6.5.8 concerning pointer comparisons.

6.6 Constant expressions

No differences with C99.

6.7 Declarations

CompCert C supports all declarations specified in C99, with the restrictions and extensions described below.

6.7.2 Type specifiers

Complex types (the `_Complex` specifier) are not supported.

6.7.2.1 Structure and union specifiers

Bit fields in unions are not supported at all. Bit fields in structures are not supported by default, but are supported through source-to-source program transformation if the `-fbitfields` option is selected (§3.2.8).

Bit fields of "plain" type `int` are treated as signed. In accordance with the ELF Application Binary Interfaces, bit fields within an integer are allocated most significant bits first on the PowerPC platform, and least significant bits first on the ARM and IA32 platforms. Bit fields never straddle an integer boundary.

Bit fields can be of enumeration type, e.g. `enum e x: 2`. Such bit fields are treated as unsigned if this enables all values of the enumeration to be represented exactly in the given number of bits, and as signed otherwise.

The members of a structure are laid out consecutively in declaration order, with enough bytes of padding inserted to guarantee that every member is aligned on its natural alignment. The natural alignment of a member can be modified by the `_Alignas` qualifier. Different layouts can be obtained if the `-fpacked-structs` option is set (§3.2.8) and the `packed` attribute (§6.2) is used.

6.7.2.2 Enums

The values of an enumeration type have type `int`.

6.7.3 Type qualifiers

The `const` and `volatile` qualifiers are honored, with the restriction below on `volatile` composite types. The `restrict` qualifier is accepted but ignored.

Accesses to objects of a volatile-qualified *scalar* type are treated as described in paragraph 6 of section 6.7.3: every assignment and dereferencing is treated as an observable event by the CompCert C formal semantics, and therefore is not subject to optimization by the CompCert compiler. Accesses to objects of a volatile-qualified *composite* type (`struct` or `union` type) are

treated as regular, non-volatile accesses: no observable event is produced, and the access can be optimized away. The CompCert compiler emits a warning in the latter case.

Following ISO C2011, CompCert supports the `_Alignas` construct as a type qualifier. This qualifier comes in two forms: `_Alignas(`*N*`)`, where *N* is a compile-time constant integer expression that evaluates to a power of two; and `_Alignas(`*T*`)`, where *T* is a type. The latter form is equivalent to `_Alignas(_Alignof(`*T*`))`.

The effect of the `_Alignas(`*N*`)` qualifier is to change the alignment of the type being qualified, setting the alignment to *N*. In particular, this affects the layout of `struct` fields. For instance:

```
struct s { char c; int _Alignas(8) i; };
```

The `Alignas(8)` qualifier changes the alignment of field `i` from 4 (the natural alignment of type `int`) to 8. This causes 7 bytes of padding to be inserted between `c` and `i`, instead of the normal 3 bytes. This also increases the size of `struct s` from 8 to 12, and the alignment of `struct s` from 4 to 8.

The alignment *N* given in the `_Alignas(`*N*`)` qualifier should normally be greater than or equal to the natural alignment of the modified type. For target platforms that support unaligned memory accesses (IA32 and PowerPC, but not ARM), *N* can also be smaller than the natural alignment.

Finally, CompCert C provides limited support for GCC-style attributes (`__attribute` keyword) used in type qualifier position. See section 6.2.

6.7.5.2 Array declarators

Variable-length arrays are not supported. The only supported array declarators are those of ISO C90, namely `[]` for an incomplete array type, and `[`*N*`]` where *N* is a compile-time constant expression for a complete array type.

6.7.5.3 Function declarators

The result type of a function must not be a structure or union type, unless the `-fstruct-return` option is active (§3.2.8).

6.7.8 Initialization

Both traditional (ISO C90) and designated initializers are supported, conforming with ISO C99.

6.8 Statements and blocks

All forms of statements specified in C99 are supported in CompCert C, with the exception described below.

6.8.4.2 The switch statement

The `switch` statement in CompCert C is restricted to the "structured" form present in Java and mandated by Misra-C. Namely, the `switch` statement must have the following form:

```
switch (expression) {
    case expr1: ...
    case expr2: ...
    ...
    default: ...
}
```

In other words, the `case` and `default` labels that pertain to a `switch` must occur at the top-level of the block following the `switch`. They cannot occur in nested blocks or under other control structures such as `if`, `while` or `for`. In particular, the infamous Duff's device is not supported.

The `asm` statement (a popular extension for inline assembly) is not supported by default, but is supported if option `-finline-asm` is set. See section 6.5 for a complete description of the syntax and semantics of `asm` statements.

6.9 External definitions

Function definitions should be written in modern, prototype form. The compiler accepts traditional, non-prototype function definitions but converts them to prototype form. In particular, `T f() {...}` is automatically converted to `T f(void) {...}`.

Functions with a variable number of arguments, as denoted by an ellipsis `...` in the prototype, are supported.

The result type of the function must not be a structure or union type, unless the `-fstruct-return` option is active (§3.2.8).

6.10 Preprocessing directives

The CompCert C compiler does not perform preprocessing itself, but delegates this task to an external C preprocessor, such as that of GCC. The external preprocessor is assumed to conform to the C99 standard.

7. Library

The CompCert C compiler does not provide its own implementation of the C standard library. It provides a few standard headers and relies on the standard library of the target system for the others. CompCert has been successfully used in conjunction with the GNU `glibc` standard library. Note, however, the following points:

7.6 Floating-point environment `<fenv.h>`

The CompCert formal semantics and optimization passes assume round-to-nearest behavior in floating-point arithmetic. If the program changes rounding modes during execution using the `fesetround` function, it must be compiled with option `-ffloat-const-prop 0` to turn off certain floating-point optimizations.

7.12 Mathematics `<math.h>`

Many versions of `<math.h>` include `long double` versions of the math functions. These functions cannot be called by CompCert-compiled code by lack of ABI-conformant support for the `long double` type.

7.13 Non-local jumps `<setjmp.h>`

The CompCert C compiler has no special knowledge of the `setjmp` and `longjmp` functions, treating them as ordinary functions that respect control flow. It is therefore not advised to use these two functions in CompCert-compiled code. To prevent misoptimisation, it is crucial that all local variables that are live across an invocation of `setjmp` be declared with `volatile` modifier.

Chapter 6

Language extensions

This chapter describes several extensions to the C99 standard implemented by CompCert: compiler pragmas (§6.1), attributes (§6.2), built-in functions (§6.3), a code annotation mechanism (§6.4), and GCC-style extended inline assembly (§6.5).

6.1 Pragmas

This section describes the pragmas supported by CompCert C. The compiler emits a warning for an unrecognized pragma.

`#pragma reserve_register` *reg-name*
: Ensure that all subsequent function definitions do not use register *reg-name* in their compiled code, and therefore preserve the value of this register. The register must be a callee-save register. The following register names are allowed:

On PowerPC:	`R14`, `R15`, ..., `R31` (general-purpose registers)
	`F14`, `F15`, ..., `F31` (float registers)
On ARM:	`R4`, `R5`, ..., `R11` (integer registers)
	`F8`, `F9`, ..., `F15` (float registers)
On IA32:	`EBX`, `ESI`, `EDI`, `EBP` (32-bit integer registers)

`#pragma section` *ident* `"`*iname*`"` `"`*uname*`"` *addr-mode access-mode*
: Define a new linker section, or modify the characteristics of an existing linker section. The parameters are as follows:

ident The compiler internal name for the section.

iname The linker section name to be used for initialized variables and for function code.

uname The linker section name to be used for uninitialized variables.

addr-mode
: The addressing mode used to access variables located in this section. On PowerPC, setting

addr-mode to `near-data` denotes a small data area, which is addressed by 16-bit offsets relative to the corresponding small data area register. On PowerPC, setting *addr-mode* to `far-data` denotes a relocatable data area, which is addressed by 32-bit offsets relative to the corresponding register. Any other value of *addr-mode* denotes standard absolute addressing.

access-mode
: One or several of `R` to denote a read-only section, `W` to denote a writable section, and `X` to denote an executable section.

Functions and global variables can be explicitly placed into user-defined sections using the `#pragma use_section` directive (see below). Another, more indirect, less recommended way is to modify the characteristics of the default sections in which the compiler automatically place function code, global variables, and auxiliary compiler-generated data. These default sections are as follows:

Internal name	What is put there
`DATA`	global, non-`const` variables of size greater than *N* bytes. *N* defaults to 0 and can be set using the `-fsmall-data` command-line option.
`CONST`	global, `const` variables of size greater than *N* bytes. *N* defaults to 0 and can be set using the `-fsmall-const` command-line option.
`SDATA`	global, non-`const` variables of size less than or equal to *N* bytes. *N* defaults to 0 and can be set using the `-fsmall-data` command-line option.
`SCONST`	global, `const` variables of size less than or equal to *N* bytes. *N* defaults to 0 and can be set using the `-fsmall-const` command-line option.
`STRING`	string literals
`CODE`	machine code for function definitions
`LITERAL`	constants (e.g. floating-point literals) referenced from function code
`JUMPTABLE`	jump tables generated for `switch` statements

A simpler, alternate way to control placement into sections is to use the `section` attribute.

Example: explicit placement into user-defined sections. We first define four new compiler sections.

```
#pragma section MYDATA "mydata_i" "mydata_u" standard RW
#pragma section MYCONST "myconst" "myconst" standard R
#pragma section MYSDA "mysda_i" "mysda_u" near-access RW
#pragma section MYCODE "mycode" "mycode" standard RX
```

We then use the `#pragma use_section` to place variables and functions in these sections, then define the variables and functions in question.

```
#pragma use_section MYDATA a b
int a;                  // uninitialized; goes into linker section "mydata_u"
double b = 3.1415;      // initialized; goes into linker section "mydata_i"
#pragma use_section MYCONST c
const short c = 42;     // goes into linker section "myconst"
```

```
#pragma use_section MYSDA d e
int d;                  // goes into linker section "mysda_u"
double e = 2.718;       // goes into linker section "mysda_i"
#pragma use_section MYCODE f
int f(void) { return a + d; }
                        // goes into linker section "mycode"
                        // accesses d via small data relative addressing
                        // accesses a via absolute addressing
```

Example: implicit placement by modifying the compiler default sections. In this example, we assume that the options `-fsmall-data 8` and `-fsmall-const 0` are given.

```
#pragma section DATA "mydata_i" "mydata_u" standard RW
#pragma section CONST "myconst" "myconst" standard R
#pragma section SDATA "mysda_i" "mysda_u" near-access RW
#pragma section CODE "mycode" "mycode" standard RX
#pragma section LITERAL "mylit" "mylit" standard R

int a;              // small data, uninitialized; goes into "mysda_u"
char b[16];         // big data, uninitialized; goes into "mydata_u"
const int c = 42; // big const data, initialized; goes into "myconst"
double f(void) { return c + 3.14; }
                    // code goes into "mycode"
                    // literal 3.14 goes into "mylit"
```

Caveat: when using non-standard sections, the linker must, in general, be informed of these sections and how to place them in the final executable, typically by providing an appropriate mapping file to the linker.

`#pragma use_section` *ident var* ...

Explicitly place the global variables or functions *var*, ... in the compiler section named *ident*. The `use_section` pragma must occur before any declaration or definition of the variables and functions it mentions. See `#pragma section` above for additional explanations and examples.

6.2 Attributes

Like the GCC compiler, the CompCert C compiler allows the programmer to attach attributes to various parts of C source code.

An attribute qualifier is of the form `__attribute((`*attribute-list*`))` or `__attribute__((`*attribute-list*`))`, where *attribute-list* is a possibly empty, comma-separated lists of attributes. Each attribute is of the following form:

attribute ::= *ident*
| *ident*`(`*attr-arg*`,...,`*attr-arg*`)`
attr-arg ::= *ident*
| `"`*string-literal*`"`
| *const-expr*

Each attribute carries a name and zero, one or several arguments, which can be identifiers, string literals, or C expressions of integer types that are compile-time constants.

For compatibility with other C compilers, the keyword `__packed__` is also recognized as an attribute:

`__packed__()`	is equivalent to	`__attribute((packed))`
`__packed__(`*params*`)`	is equivalent to	`__attribute((packed(`*params*`)))`

In C source files, attribute qualifiers can occur anywhere a standard type qualifier (`const`, `volatile`) can occur, and also just after the `struct` and `union` keywords. For partial compatibility with GCC, the CompCert parser allows attributes to occur in several other places, but silently ignores them.

Warning. Some standard C libraries, when used in conjunction with CompCert, deactivate the `__attribute__` keyword: the standard includes, or CompCert itself, define `__attribute__` as a macro that erases its argument. This is the case for the Glibc standard library under Linux, and the XCode header files under MacOS X. For this reason, please use the `__attribute` keyword in preference to the `__attribute__` keyword.

The following attributes are recognized by CompCert. Unrecognized attributes are silently ignored.

`aligned(`*N*`)` and `__aligned__(`*N*`)`
: Specify the alignment to use for a variable or a `struct` member. The argument *N* is the desired alignment, in bytes. It must be a power of 2. This attribute is equivalent to the qualifier `_Alignas(`*N*`)`.

`packed(`*max-member-alignment*`,` *min-struct-alignment*`,` *byte-swap*`)`
: This attribute is recognizedonly if the `-fpacked-structs` option is active (§3.2.8).

  The `packed` attribute applies to a `struct` declarations and affects the memory layout of the members of this `struct`. Zero, one, two or three integer arguments can be provided.

  If the *max-member-alignment* parameter is provided, the natural alignment of every member (field) of the structure is reduced to at most *max-member-alignment*. In particular, if *max-member-alignment* = 1, members are not aligned, and no padding is ever inserted between members.

  If the *min-struct-alignment* parameter is provided, the natural alignment of the whole structure is increased to at least *min-struct-alignment*.

  If the *byte-swap* parameter is provided and equal to 1, accesses to structure members of integer or pointer types are performed using the opposite endianness than that of the target platform. For PowerPC, accesses are done in little-endian mode instead of the natural big-endian mode; for IA32 and ARM, accesses are done in big-endian mode instead of the natural little-endian mode.

  Examples:

```
struct __attribute__((packed(1))) s1 {     // suppress all padding
```

```
  char c;            // at offset 0
  int i;             // at offset 1
  short s;           // at offset 5
};                   // total size is 7, structure alignment is 1

struct __attribute__((packed(4,16,1))) s2 {
  short s;           // at offset 0; byte-swap at access
  int i;             // at offset 4 (because 4-aligned); byte-swap at access
};                   // total size is 8, structure alignment is 16

struct s3 {          // default layout
  char c;            // at offset 0
  int i;             // at offset 4 (because 4-aligned)
  short s;           // at offset 6
};                   // total size is 8, structure alignment is 4
```

Limitations: For a byte-swapped structure, all members should be of integer or pointer types, or arrays of those types.

Reduced member alignment should not be used on the ARM platform, since unaligned memory accesses on ARM can crash the program or silently produce incorrect results. Only the PowerPC and IA32 platforms support unaligned memory accesses.

`packed`
: This form of the `packed` attribute is equivalent to `packed(1)`. It causes the the structure it modifies to be laid out with no alignment padding between the members. The size of the resulting structure is therefore the sum of the sizes of its members. The alignment of the resulting structure is 1.

`section("`*section-name*`")` and `__section__("`*section-name*`")`
: Specify the linker section *section-name* where to place functions and global variables whose types carry this `section` attribute. The linker section is declared as executable read-only if the attribute applies to a function definition; as read-only if the attribute applies to a `const` variable definition; and as read-write if the attribute applies to a non-`const` variable definition.

  The `#pragma section` and `#pragma use_section` directives can be used to obtain finer control on user-defined sections (§6.1).

6.3 Built-in functions

Built-in functions are functions that are predefined in the initial environment — the environment in which the compilation of a source file starts. In other words, these built-in functions are always available: there is no need to include header files.

Built-in functions give access to interesting capabilities of the target processor that cannot be exploited in standard C code. For example, most processors provide an instruction to quickly compute the absolute value of a floating-point number. In CompCert C, this instruction is made available to the programmer via the `__builtin_fabs` function. It provides a faster alternative to the `fabs` library function from `<math.h>`.

Invocations of built-in functions are automatically inlined by the compiler at point of use. It is a compile-time error to take a pointer to a built-in function.

Some built-in functions are available on all target platforms supported by CompCert. Others are specific to a particular platform.

6.3.1 Common built-in functions

Integer arithmetic:

`unsigned int __builtin_bswap(unsigned int x)`
: Swap the bytes of `x` to change its endianness. If `x` is of the form `0xaabbccdd`, the result is `0xddccbbaa`.

`unsigned int __builtin_bswap32(unsigned int x)`
: A synonym for `__builtin_bswap`.

`unsigned short __builtin_bswap16(unsigned short x)`
: Swap the bytes of `x` to change its endianness. If `x` is of the form `0xaabb`, the result is `0xbbaa`.

Floating-point arithmetic:

`double __builtin_fabs(double x)`
: Return the floating-point absolute value of its argument, or NaN if the argument is NaN. This function is equivalent to the `fabs()` function from the `<math.h>` standard library, but executes faster.

Block copy with known size and alignment:

`void __builtin_memcpy_aligned(void * dst, const void * src, size_t sz, size_t al)`
: Copy `sz` bytes from the memory area at `src` to the memory area at `dst`. The source and destination memory areas must be either disjoint or identical; the behavior is undefined if they overlap. The pointers `src` and `dst` must be aligned on an `al` byte boundary, where `al` is a power of 2. The `sz` and `al` arguments must be compile-time constants. A typical invocation is

```
__builtin_memcpy_aligned(&dst, &src, sizeof(dst), _Alignof(dst));
```

: where `dst` and `src` are two objects of the same complete type. An invocation of `__builtin_memcpy_aligned(dst,src,sz,al)` behaves like `memcpy(dst,src,sz)` as defined in the `<string.h>` standard library. Knowing the size and alignment at compile-time enables the compiler to generate very efficient inline code for the copy.

Memory accesses with reversed endianness:

`unsigned short __builtin_read16_reversed(const unsigned short * ptr)`
: Read a 16-bit integer at address `ptr` and reverse its endianness by swapping the two bytes of the result.

`unsigned int __builtin_read32_reversed(const unsigned int * ptr)`
: Read a 32-bit integer at address `ptr` and reverse its endianness by swapping the four bytes of the result, as `__builtin_bswap` does.

`void __builtin_write16_reversed(unsigned short * ptr, unsigned short x)`
: Reverse the endianness of `x` by swapping its two bytes, then write the 16-bit result at address `ptr`.

`void __builtin_write32_reversed(unsigned int * ptr, unsigned int x)`
Reverse the endianness of `x` by swapping its four bytes, then write the 32-bit result at address `ptr`.

Synchronization:

`void __builtin_membar(void)`
Software memory barrier. Prevent the compiler from moving memory loads and stores across the call to `__builtin_membar`. No processor instructions are generated, hence the hardware can still reorder memory accesses. To generate hardware memory barriers, see the "synchronization" built-in functions specific to each processor.

6.3.2 PowerPC built-in functions

Integer arithmetic:

`int __builtin_mulhw(int x, int y)`
Return the high 32 bits of the full 64-bit product of two signed integers.

`unsigned int __builtin_mulhwu(unsigned int x, unsigned int y)`
Return the high 32 bits of the full 64-bit product of two unsigned integers.

`int __builtin_clz(unsigned int x)`

`int __builtin_clzl(unsigned long x)`
Count the number of consecutive zero bits in `x`, starting with the most significant bit. The result is between 0 and 32 inclusive.

`int __builtin_clzll(unsigned long long x)`
Count the number of consecutive zero bits in `x`, starting with the most significant bit. The result is between 0 and 64 inclusive.

`int __builtin_isel(_Bool cond, int iftrue, int iffalse)`

`unsigned int __builtin_uisel(_Bool cond, unsigned int iftrue, unsigned int iffalse)`
Return `iftrue` if `cond` is true, `iffalse` if `cond` is false. Expands to an `isel` instruction on Freescale EREF processors, and to a branch-free instruction sequence on other PowerPC processors.

Floating-point arithmetic:

`double __builtin_fmadd(double x, double y, double z)`
Fused multiply-add. Compute `x*y + z` without rounding the intermediate product `x*y`.

`double __builtin_fmsub(double x, double y, double z)`
Fused multiply-sub. Compute `x*y - z` without rounding the intermediate product `x*y`.

`double __builtin_fnmadd(double x, double y, double z)`
Fused multiply-add-negate. Compute `-(x*y + z)` without rounding the intermediate product `x*y`.

`double __builtin_fnmsub(double x, double y, double z)`
Fused multiply-sub-negate. Compute `-(x*y - z)` without rounding the intermediate product `x*y`.

`double __builtin_fsqrt(double x)`
Return the square root of `x`, like the `sqrt` function from the `<math.h>` standard library. The corresponding PowerPC instruction is optional and not supported by all processors.

`double __builtin_frsqrte(double x)`
Compute an estimate (with relative accuracy $1/32$) of $1/\sqrt{x}$, the reciprocal of the square root of `x`. The corresponding PowerPC instruction is optional and not supported by all processors.

`float __builtin_fres(float x)`
Compute an estimate (with relative accuracy $1/256$) of the single-precision reciprocal of `x`. The corresponding PowerPC instruction is optional and not supported by all processors.

`double __builtin_fsel(double x, double y, double z)`
Return `y` if `x` is greater or equal to 0.0. Return `z` if `x` is less than 0.0 or is NaN. The corresponding PowerPC instruction is optional and not supported by all processors.

`int __builtin_fcti(double x)`
Round the given floating-point number `x` to an integer and return this integer. The difference with the standard C conversion `(int)x` is that the latter rounds `x` towards zero, while `__builtin_fcti(x)` rounds `x` according to the current rounding mode (by default: to the nearest integer, ties round to even).

Synchronization instructions:

`void __builtin_eieio(void)`
Issue an `eieio` barrier.

`void __builtin_sync(void)`
Issue a `sync` barrier.

`void __builtin_isync(void)`
Issue an `isync` barrier.

`void __builtin_lwsync(void)`
Issue an `lwsync` barrier.

`void __builtin_mbar(int level)`
Issue a `mbar` barrier. The integer argument must be 0 or 1.

`void __builtin_trap(void)`
Abort the program on an unconditional `trap` instruction.

Cache control instructions:

`void __builtin_dcbf(void * addr)`

`void __builtin_dcbi(void * addr)`

```
void __builtin_dcbtls(void * addr, int level)
```

```
void __builtin_dcbz(void * addr)
```

```
void __builtin_icbi(void * addr)
```

```
void __builtin_icbtls(void * addr, int level)
```
Issue the corresponding cache control instruction. `addr` is the address of the cache block affected. `level` must be the constant 0 (L1 cache) or 2 (L2 cache).

```
void __builtin_prefetch(void * addr, int for_write, int level)
```
Issue a `dcbt` instruction if `for_write` is 0 and a `dcbtst` instruction if `for_write` is 1.

Access to special registers:

```
unsigned int __builtin_get_spr(int spr)
```

```
unsigned long long __builtin_get_spr64(int spr)
```

```
void __builtin_set_spr(int spr, unsigned int value)
```

```
void __builtin_set_spr64(int spr, unsigned long long value)
```
The `spr` argument is the number of the special register accessed. It must be a compile-time constant.

Atomic (sequentially-consistent) memory operations:

```
void __builtin_atomic_exchange(int * addr, int * new, int * old)
```
Store the current value of `*addr` in `*old` and set `*addr` to the value `*new`.

```
void __builtin_atomic_load(int * p, int * val)
```
Read the current value of `*p` and store it into `*val`.

```
_Bool __builtin_atomic_compare_exchange(int * p, int * expected, int * desired)
```
Compare the current value of `*p` with `*expected`. If equal, set `*p` to the value `*desired` and return 1. If different, set `*expected` to the current of `*p` and return 0;

```
int __builtin_sync_fetch_and_add(int * p, int delta)
```
Add `delta` to the contents of `*p`. Return the initial value of `*p` before the addition.

6.3.3 IA32 built-in functions

Integer arithmetic:

```
int __builtin_clz(unsigned int x)
```
Count the number of consecutive zero bits in `x`, starting with the most significant bit. The result is undefined if `x` is 0. Otherwise, the result is between 0 and 31 inclusive.

`int __builtin_ctz(unsigned int x)`
Count the number of consecutive zero bits in `x`, starting with the least significant bit. The result is undefined if `x` is 0. Otherwise, the result is between 0 and 31 inclusive, and is the number of the least significant bit that is set in `x`.

Floating-point arithmetic:

`double __builtin_fsqrt(double x)`
Return the square root of `x`, like the `sqrt` function from the `<math.h>` standard library.

`double __builtin_fmax(double x, double y)`
Return the greater of `x` and `y`. If `x` or `y` are NaN, the result is either `x` or `y`, but it is unspecified which.

`double __builtin_fmin(double x, double y)`
Return the smaller of `x` and `y`. If `x` or `y` are NaN, the result is either `x` or `y`, but it is unspecified which.

`double __builtin_fmadd(double x, double y, double z)`
Fused multiply-add. Compute `x*y + z` without rounding the intermediate product `x*y`. Requires a processor that supports the FMA3 extensions.

`double __builtin_fmsub(double x, double y, double z)`
Fused multiply-sub. Compute `x*y - z` without rounding the intermediate product `x*y`. Requires a processor that supports the FMA3 extensions.

`double __builtin_fnmadd(double x, double y, double z)`
Fused multiply-negate-add. Compute `-(x*y) + z` without rounding the intermediate product `x*y`. Requires a processor that supports the FMA3 extensions.

`double __builtin_fnmsub(double x, double y, double z)`
Fused multiply-negate-sub. Compute `-(x*y) - z` without rounding the intermediate product `x*y`. Requires a processor that supports the FMA3 extensions.

6.3.4 ARM built-in functions

Integer arithmetic:

`int __builtin_clz(unsigned int x)`
Count the number of consecutive zero bits in `x`, starting with the most significant bit. The result is between 0 and 32 inclusive.

Floating-point arithmetic:

`double __builtin_fsqrt(double x)`
Return the square root of `x`, like the `sqrt` function from the `<math.h>` standard library.

Synchronization instructions:

`void __builtin_dmb(void)`
Issue a `dmb` (data memory) barrier.

`void __builtin_dsb(void)`
Issue a `dsb` (data synchronization) barrier.

`void __builtin_isb(void)`
Issue an `isb` (instruction synchronization) barrier.

6.4 Program annotations

CompCert C provides a general mechanism to attach free-form annotations (text messages possibly mentioning the values of variables) to C program points, and have these annotations transported throughout compilation, all the way to the generated assembly code. Before describing this annotation mechanism, we motivate it by an example.

Motivating example Several static analysis tools operate not at the level of C source code, but directly on executable machine code. In particular, this is the case for the computation of Worst Case Execution Times (WCET) by static analysis, as performed for instance by the aiT WCET analyzer from AbsInt.

To analyze machine code with sufficient precision, these tools sometimes require the programmers to provide additional information that cannot easily be reconstructed from the machine code alone. Consider for example the following binary search routine, whose WCET we would like to estimate:

```
int bsearch(int tbl[100], int v) {
  int l = 0, u = 99;
  while (l <= u) {
    int m = (l + u) / 2;
    if (tbl[m] < v) l = m + 1;
    else if (tbl[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

To compute a tight upper bound on the WCET, the aiT analyzer must be given two additional pieces of information. First, the `while` loop executes at most $\lceil \log_2(100) \rceil = 7$ iterations. Second, the `m` array index is always between 0 and 99, ensuring that the memory access `tbl[m]` always hits in data memory and never, for instance, in memory-mapped devices. Using aiT, this information must be provided as annotations on the *machine* code, for instance via a separate text file containing

```
loop at "bsearch" + 0x14 max 7
instruction at "bsearch" + 0x28 is entered with r4 = (0,99)
```

Note that these annotations are expressed in terms of machine code addresses (`"bsearch" + 0x14`) and machine registers (`r4`), not in terms of source program points and variables.

Enters CompCert's source annotation mechanism. It is presented as a pseudo built-in function called `__builtin_annot`, taking as arguments a string literal and zero, one or several local variables. Let us use this mechanism to annotate the `bsearch` routine above.

```
int bsearch(int tbl[100], int v) {
  int l = 0, u = 99;
  __builtin_annot("loop max 7");
  while (l <= u) {
    int m = (l + u) / 2;
    __builtin_annot("entered with %1 = (0,99)", m);
    if (tbl[m] < v) l = m + 1;
    else if (tbl[m] > v) u = m - 1;
    else return m;
  }
  return -1;
}
```

We then compile this function using `ccomp -Wa,-a -c bsearch.c` and look at the generated assembly listing:

```
...
  4                     bsearch:
  5 0000 9421FFF0               stwu    r1, -16(r1)
  6 0004 7C0802A6               mflr    r0
  7 0008 90010008               stw     r0, 8(r1)
  8 000c 39200000               addi    r9, 0, 0
  9 0010 39400063               addi    r10, 0, 99
 10                     # annotation: loop max 7
 11                     .L100:
 12 0014 7C095000               cmpw    cr0, r9, r10
 13 0018 41810040               bt      1, .L101
 14 001c 7CE95214               add     r7, r9, r10
 15 0020 7CE50E70               srawi   r5, r7, r1
 16 0024 7CA50194               addze   r5, r5
 17                     # annotation: entered with r5 = (0,99)
 18 0028 54A8103A               rlwinm  r8, r5, 2, 0, 29 # 0xfffffffc
 19 002c 7CC3402E               lwzx    r6, r3, r8
 20 0030 7C062000               cmpw    cr0, r6, r4
 21 0034 4180001C               bt      0, .L102
...
```

We see that CompCert generates no machine code for the two `__builtin_annot` source statements. Instead, it produces assembly comments at the exact program points corresponding to those in the source function. These comments consist of the string literal carried by the annotation, where positional parameters %1, %2, etc, are replaced by the locations (processor registers or stack slots) of the corresponding variable arguments.

From the assembly listing above, an ad-hoc tool (not provided with CompCert) can easily generate a machine-level annotation file usable by the aiT WCET analyzer. (Future directions for CompCert include facilitating the exploitation of annotations, for instance by saving them in special debug sections of the generated executables.)

Generalizing from the example above, we see that `__builtin_annot` statements offer a general and flexible mechanism to mark program points and local variables in C source files, then track them all the way to assembly language. Besides helping with static analysis at the machine code level, this mechanism also

facilitates manual traceability analysis between C and assembly.

Reference description CompCert's annotation mechanism is presented by the following two pseudo built-in functions.

`void __builtin_annot(const char * msg, ...)`

The first argument must be a string literal. It can be followed by arbitrarily many additional arguments, of integer, pointer or floating-point types. In the intended uses described above, the additional arguments are names of local variables, but arbitrary expressions are allowed.

The formal semantics of `__builtin_annot` is that of a *pro forma* "print" statement: the compiler assumes that every time a `__builtin_annot` is executed, the message and the values of the additional arguments are displayed on an hypothetical output device. In other words, an invocation of `__builtin_annot` is treated as an observable event in the program execution. The compiler, therefore, guarantees that `__builtin_annot` statements are never removed, duplicated, nor reordered; instead, they always execute at the times prescribed by the semantics of the source program, and in the same order relative to other observable events such as calls to I/O functions or volatile memory accesses.

As described in the motivational example above, the actual effect of a `__builtin_annot` statement is simply to generate a comment in the assembly code. This comment consists of the message carried by the annotation, where %*n* sequences are replaced by the machine location containing the value of the *n*-th additional argument, or by its value if the *n*-th additional argument is a compile-time constant expression of numerical type.

The location of an argument is either a machine register, an integer or floating-point constant, the name of a global variable, or a stack memory location of the form `mem(`*sp* `+` *offset*`,` *size*`)` where *sp* is the stack pointer register, *offset* a byte offset relative to the value of *sp*, and *size* the size in bytes of the argument. For example, on the PowerPC, register locations are `R3...R31` and `F0...F31`, and stack locations are of the form `mem(R1 +` *offset*`,` *size*`)`.

If all additional arguments are non-volatile C variables or compile-time constant expressions, it is guaranteed that no code (other than the assembly comment) will be generated for `__builtin_annot`. Moreover, the location displayed as a replacement for the %*n* sequence is guaranteed to be the location where the corresponding variable resides.

If one or several additional arguments are complex expressions (neither non-volatile variables nor constant expressions), useless code is generated to compute their values and leave them in temporary registers or stack locations. The location displayed as a replacement for the %*n* sequence is that of the corresponding temporary. This behavior is not particularly useful for static analysis at the machine level, and can generate useless code. It is therefore highly recommended to use only non-volatile variable names or constant expressions as additional parameters to `__builtin_annot`.

`int __builtin_annot_intval(const char * msg, int x)`

In contrast with `__builtin_annot`, which is used as a statement, `__builtin_annot_intval` is intended to be used within expressions, to track the location containing the value of a subexpression and return this value unchanged. A typical use is within array indexing expressions, to express an assertion over the array index:

```
int x = tbl[__builtin_annot_intval("entered with %1=(0,99)", (lo + hi)/ 2)];
```

The formal semantics of `__builtin_annot_intval` is also the *pro forma* effect of displaying the message `msg` and the value of the integer argument `x` on a hypothetical output device. In addition, the value of the second argument `x` is returned unchanged as the result of `__builtin_annot_intval`.

In the compiled code, `__builtin_annot_intval` evaluates its argument `x` to some temporary register, then inserts an assembly comment equal to the text `msg` where occurrences of `%1` are replaced by the name of this register.

6.5 Extended inline assembly

Like in GCC and Clang, inline assembly code using the `asm` statement can be parameterized by C expressions as operands. The actual locations of the operands (registers and memory locations), as determined during compilation, are inserted in the given assembly code fragment.

Warning: indiscriminate use of `asm` statements can ruin all the semantic preservation guarantees of CompCert. For this reason, support for `asm` statements is turned off by default and must be activated through the `-finline-asm` or `-fall` options. For the generated code to behave properly, it is the responsibility of the programmer to list (in the `asm` statement) the registers and memory that are modified by the assembly code, and to avoid modifying memory that is private to the compiled code, such as return addresses stored in the stack.

Examples Here is how to use the PowerPC `mulhw` instruction to compute the high 32 bits of the 64-bit integer product `x * y`:

```
int prod;
asm ("mulhw %0,%1,%2" : "=r"(prod) : "r"(x), "r"(y));
```

The two arguments `x` and `y` are evaluated into registers, which get substituted for `%1` and `%2` (respectively) in the assembly template. Likewise, `%0` is substituted by the register associated with variable `prod`. The three `r` constraint indicates that all three operands are expected in registers. The `=` constraint in `=r` means that the operand is an output.

To designate operands, symbolic names can be used instead of operand numbers. Using named operands, the `mulhw` example above can be written as:

```
asm ("mulhw %[res],%[arg1],%[arg2]" : [res]"=r"(prod) : [arg1]"r"(x), [arg2]"r"(y));
```

Sometimes, inline assembly code has no result value, but modifies memory. An example is the `dcba` instruction of PowerPC, which allocates a cache block without reading its contents from main memory:

```
asm volatile ("dcba 0, %[addr]" : : [addr]"r"(p) : "memory");
```

Note the absence of output operand, the `volatile` modifier indicating that the assembly code has side effects, and the `"memory"` annotation indicating that the assembly code modifies memory in ways that are not predictable by the compiler. CompCert treats all `asm` statements as volatile and clobbering memory, but other compilers need these annotations to produce correct code.

Some instructions operate over memory locations instead of registers. In this case, the `m` constraint should be used instead of `r`. For example, the IA32 instruction `fnstsw` stores the FP control word in a memory location:

```
unsigned short cw;
asm ("fnstsw %0" : "=m" (cw));
```

Note that this `asm` statement has no inputs, hence the second colon can be omitted.

Other instructions demand arguments that are integer constants, instead of registers or memory locations. In this case, the `i` constraint should be used, and the corresponding argument must be a compile-time constant expression. For example, here is how to use the PowerPC `mfspr` and `mtspr` instructions to read and write special registers:

```
#define read_spr(regno,res) \
   asm ("mfspr %0,%1" : "=r"(res) : "i"(regno))
#define write_spr(regno,val) \
   asm volatile ("mtspr %0,%1" : : "i"(regno), "r"(val))
```

We already used the `"memory"` annotation telling the compiler that the assembly code "clobbers" (modifies unpredictably) the memory state. Likewise, if the assembly code modifies registers other than that of the output operand, the names of those registers must be given in the clobber list. For example, here is an implementation of atomic test-and-set using the IA32 locked-exchange instruction:

```
int testandset(int * p)
{ // store 1 in *p and return the previous value of *p
  int res;
  asm volatile
    ("movl $1, %%eax\n\t"
     "xchgl (%[addr]), %%eax\n\t"
     "movl %%eax, %[oldval]"
     : [oldval]"=r"(res) : [addr]"r"(p) : "eax", "memory");
  return res;
```

Note that the assembly template can contain several instructions, using `\n\t` in the template to separate the instructions. Also note the use of `%%` in the assembly template to stand for a single `%` character in the generated assembly code.

Since the template uses `eax` as a temporary register, we must list register `eax` as clobbered. Besides informing the compiler that the previous contents of `eax` are destroyed, this clobber annotation also ensures that none of the `asm` operands (`res` and `p`) are allocated to `eax`.

Register operands of type `long long` or `unsigned long long` (64-bit integers) need special handling, since CompCert allocates them into pairs of 32-bit registers. The assembly template must use `%R` to refer to the most significant half of the register pair, and `%Q` to refer to the least significant half. For example, here is how to use the IA32 `rdtsc` instruction to read the time stamp counter as an unsigned 64-bit integer:

```
unsigned long long rdtsc(void)
{
  unsigned long long res;
  asm("rdtsc\n\t"
      "movl %%eax, %Q0\n\t"
```

```
      "movl %%edx, %R0\n\t"
      : "=r" (res) : : "eax", "edx");
  return res;
}
```

Reference description The syntax of extended inline assembly is as follows:

statement	::=	... | `asm` (`const`|`volatile`)* `(` "*template*" *asm-operands* `)`
asm-operands	::=	`:` *asm-output* `:` *asm-inputs* `:` *asm-clobbers*
	|	`:` *asm-output* `:` *asm-inputs*
	|	`:` *asm-output*
	|	
asm-outputs	::=	*asm-arg*?
asm-inputs	::=	*asm-arg*`,` ... `,` *asm-arg*
asm-clobbers	::=	"*resource*"`,` ... `,` "*resource*"
asm-arg	::=	(`[`*name*`]`)? "*constraint*" `(` *expr* `)`

An `asm` statement carries an assembly template (a string literal) and up to 3 lists of operands separated by colons: a list of zero or one output expressions (results produced by the assembly code); a list of zero, one or several input expressions (arguments used by the assembly code); and a list of zero, one or several resources (e.g. processor registers) that are "clobbered" by the assembly code.

The assembly template is a string literal possibly containing placeholders marked with a `%` (percent) character: either numbered placeholders (`%0`, `%1`, etc) or named placeholders (`%[`*name*`]`). During compilation, the placeholders are replaced by the locations of the corresponding operands, then the resulting text is given to the assembler as is. A `%%` sequence in the template is translated to a single `%` character in the text. Operands are numbered consecutively starting with `%0` for the first output operand (if any) and continuing with the input operands. Instead of numbers, the template can also refer to operands by their optional names, using the `%[`*name*`]` notation.

The assembly outputs and the inputs are comma-separated, possibly empty lists of C expressions preceded by an optional name between brackets and a mandatory constraint (a string literal). CompCert can handle zero or one output; two outputs or more cause a compile-time error. For inputs, the following constraints are supported:

Input constraint	Meaning
`"r"`	Register. The expression is evaluated into a processor register, whose name is inserted in the assembly template.
`"m"`	Memory location. The expression is evaluated into a memory location, whose address is inserted (as a valid processor addressing mode) in the assembly template.
`"i"`	Integer immediate. The expression must be a compile-time constant. Its value is inserted in the assembly template as a decimal literal.

For outputs, the C expressions must be l-values, and the following constraints are supported:

Input constraint	Meaning
`"=r"`	Register. This is either the register allocated to the output expression (if it is a local variable allocated to a register), or a temporary register chosen by CompCert, whose value is assigned to the expression just after the assembly code.
`"=m"`	Memory location. This is the memory location of the output expression. Its address is inserted (as a valid processor addressing mode) in the assembly template.

The fourth, optional component of an `asm` statement is a comma-separated list of resources that are "clobbered" by the assembly code fragment, i.e. set to unpredictable values. The resources, given as string literals, are either processor register names or the special resources `"memory"` and `"cc"`, denoting unpredictable changes to the memory and the processor's condition codes, respectively. CompCert always assumes that inline assembly can modify memory and condition codes in unpredictable ways, even if the `"memory"` and `"cc"` clobbers are not specified. The register names are case-insensitive and depend on the target processor, as follows:

Processor	Register names
ARM	integer registers: `"r0"`, `"r1"`, ..., `"r12"`, `"r14"` VFP registers: `"f0"`, `"f1"`, ..., `"f15"`
IA32	integer registers: `"eax"`, `"ebx"`, `"ecx"`, `"edx"`, `"esi"`, `"edi"`, `"ebp"` XMM registers: `"xmm0"`, `"xmm1"`, ..., `"xmm7"`
PowerPC	integer registers: `"r0"`, `"r3"`, ..., `"r12"`, `"r14"`, ..., `"r31"` FP registers: `"f0"`, `"f1"`, `"f2"`, ..., `"f31"`

Registers not listed above are reserved for use by the compiler and must not be modified by inline assembly code. For example, the stack pointer register (`r13` for ARM, `esp` for IA32, `r1` for PowerPC) must be preserved.

Bibliography

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

[2] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development – Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[3] Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008*, pages 255–264. ACM Press, 2008.

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

[5] ISO. International standard ISO/IEC 9899:2011, Programming languages – C, 2011.

[6] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[7] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[8] John McCarthy and James Painter. Correctness of a compiler for arithmetical expressions. In *Mathematical Aspects of Computer Science*, volume 19 of *Proc. of Symposia in Applied Mathematics*, pages 33–41. American Mathematical Society, 1967.

[9] R[obin] Milner and R[ichard] Weyrauch. Proving compiler correctness in a mechanized logic. In Bernard Meltzer and Donald Michie, editors, *Proc. 7th Annual Machine Intelligence Workshop*, volume 7 of *Machine Intelligence*, pages 51–72. Edinburgh University Press, 1972.

[10] IEEE Computer Society. IEEE standard for floating-point arithmetic, IEEE Std 754-2008, 2008.

[11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 283–294. ACM Press, 2011.