A Unified Coq Framework for Verifying C Programs with Floating-Point Computations

Tahina Ramananandro  Paul Mountcastle  Benoît Meister  Richard Lethin
Reservoir Labs Inc., USA
lastname@reservoir.com

Abstract

We provide concrete evidence that floating-point computations in C programs can be verified in a homogeneous verification setting based on Coq only, by evaluating the practicality of the combination of the formal semantics of CompCert Clight and the Flocq formal specification of IEEE 754 floating-point arithmetic for the verification of properties of floating-point computations in C programs. To this end, we develop a framework to automatically compute real-number expressions of C floating-point computations with rounding error terms along with their correctness proofs. We apply our framework to the complete analysis of an energy-efficient C implementation of a radar image processing algorithm, for which we provide a certified bound on the total noise introduced by floating-point rounding errors and energy-efficient approximations of square root and sine.

Categories and Subject Descriptors G.1.0 [Mathematics of Computing, Numerical Analysis, General]: Computer arithmetic, Error analysis, Interval arithmetic; D.3.1 [Software, Programming Languages, Formal Definitions and Theory]: Semantics; D.2.4 [Software, Software Engineering, Software/Program Verification]: Correctness proofs, Formal methods

Keywords Formal Verification, Coq, Floating-point Computations, C.

1. Introduction

Numerical rounding errors can often have catastrophic effects, and throwing more bits at a problem is no guarantee that these numerical precision problems will be avoided. Conversely, significant performance savings can be achieved by reducing some precision and introducing some approximations, without necessarily introducing dramatic errors in the final results. To what extent can such error bounds be guaranteed on C programs with floating-point computations? How trustworthy can such guarantees be? The goal of our effort described in this paper is to create a context for provable error estimates at lower numerical precision, thereby saving power by possibly avoiding unnecessary computations, and to do so with certifiably lower risk to the mission due to precision failures.

Properties (accuracy, stability, complexity, time, space and energy consumption, etc.) of floating-point computations have been an ongoing concern for industrial companies developing control software heavily relying on machine arithmetic. Since the inception of floating-point computations in computers, their study has led to the entire field of numerical analysis, at the intersection of computer science and mathematics. However, the desire to strengthen the trust in computer implementations of numerical programs has grown to the point that pen-and-paper proofs are no longer sufficient and computerized verification strategies become necessary.

In this paper, we combine both Coq specifications of floating-point arithmetic and C semantics in a unified Coq setting for the purpose of source-level verification of C programs performing floating-point computations, to provide stronger numerical guarantees based on the real-number semantics of floating-point numbers. The main goal of our approach is to show that it is possible to prove numerical properties of practical C programs in a verification setting whose trusted computing base only contains the faithfulness of formal mathematical Coq specifications of C and floating-point numbers, the soundness of Coq’s underlying logic (the Calculus of Inductive Constructions [7]), and the implementation of the Coq proof checker.

Our approach relies on specifications of floating-point arithmetic and C semantics written in a mathematical language and thus meant to be more widely readable and understandable than the actual code of specific implementations of verification tools, especially if such tools are automated and highly optimized, such as Fluctuat [21, 38] which is written in C++. In particular, it is not easy to trust the fact that the results computed by those tools will reflect on the actual behavior of the C program with floating-point computations being verified. In this paper, we provide a verification approach that bridges such gap.

Contributions The contributions of our work, which we describe in this paper, are as follows:

• In Section 3, we clarify the interpretation of the semantics of C floating-point computations by defining a core floating-point calculus and proving its consistency with C implicit type conversions (casts and type promotions) in Coq against the formal semantics of a subset of CompCert C.

• Based on our core floating-point calculus, we develop VCFloat, a verification framework based on an automatic Coq tactic to reason about the real-number values of C floating-point computations, which we describe in Section 4.

• In Section 5, we demonstrate the practicality of VCFloat by applying it to the first complete analysis of a practical C program with floating-point computations: we introduce an energy-efficient C implementation of a radar image processing algorithm with energy-efficient approximations. We have computed
and proved a bound on the total image noise introduced by our C implementation, and we have mechanized the whole proof using Coq, with a Coq theorem statement about the actual behaviour of our C program.

We have carried out our proofs using the Coq proof assistant [16]. Our proofs are available on the Internet at http://github.com/reservoirlabs

2. Related Work

There already exist practical tools to carry on some form of verification of C programs computing floating-point values. Fluctuat [21, 38] is a closed-source commercial automatic static analyzer for the verification of floating-point properties of C programs, based on abstract interpretation [17]. Fluctuat is heavily used in industry [18], and it is implemented in C++. However, trusting the correctness of Fluctuat implies to trust the implementations of the static analysis algorithms and their optimizations, which can be very complex. This is why we rather advocate for the use of general-purpose verification tools in which floating-point arithmetic is formally specified in a readable mathematical specification language.

Mechanized Proofs of Floating-Point Properties with Coq

To provide the highest possible level of trust in IEEE 754 floating-point computations independently of the particular implementation of the verification method or tool, it becomes necessary to specify IEEE 754 using a mathematical specification language in a proof assistant. Floqc [9] is such a comprehensive specification of IEEE 754 in the Coq proof assistant [7, 16], on which our work builds.

The Metalibm project [26] aims to build a certified mathematical library implemented in C with floating-point computations. Metalibm builds on Sollya [15], an open-source tool and environment for the development of “safe floating-point code.” Sollya is targeted to help develop implementations of mathematical elementary transcendental functions such as trigonometry, exponential, etc. and automatically computes approximation and rounding error bounds. Some results produced by Sollya can be verified using the Gappa tool [29], which supports floating-point and fixed-point interval arithmetic and produces a proof certificate that can be checked [10] with Coq against Floqc. Gappa is also used in the CRlibm project [33], a certified mathematical library which comes with a mostly on-paper correctness proof with some parts checked using Gappa, and on which Metalibm is building. The certification effort of Metalibm and CRlibm only focuses on the correctness of the floating-point computations, rather than the verification of their embedding in C code, which we also address in our work.

Mechanized Proofs of Floating-Point Computations in C Programs

Even though floating-point computations can be verified, such verification results must transport to the actual C program in which those computations will be implemented.

There have been very few successful attempts to verify C programs with floating-point computations with respect to a real-number specification. The first fully verified implementation of a numerical (floating-point) C program has been verified by Boldo et al. [11]: a C implementation of a numerical solver for a wave equation. They prove that the function computed by their program is a solution of the wave equation provided by the user within some error bounds. Their verification is based on Frama-C [5], an automatic static analyzer for C programs that generates verification conditions deemed enough to prove the functional correctness of the C code. Such verification conditions are checked using external verification tools such as Coq with Floqc, but also automatic SMT solvers such as Alt-Ergo, CVC3 and Z3. Allowing the user to choose their own combination of tools can make verification very practical (which explains why Frama-C is already used in industry), but it is a major drawback when it comes to assessing the mathematical soundness of such heterogeneous combination of verification tools. Moreover, users must trust the implementation of Frama-C, more precisely the fact that the verification conditions generated by Frama-C are enough to ensure functional correctness. By contrast, our approach advocates the use of a homogeneous combination of verification tools and proof libraries based on Coq only, and trusts as little implementation code as possible, replacing trusted, sometimes heavily optimized implementations of domain-specific verification tools with more trustworthy formal specifications in readable mathematical languages.

Mechanized Proofs on C Programs against a Formal Semantics

To avoid trusting a particular implementation of a C program verifier, it is necessary to formalize the semantics of a suitable subset of C and to build verification tools that are certified against this formal semantics.

CompCert [27, 28] is one of the first realistic efforts to specify a subset of C in Coq for the purpose of formal verification. CompCert specifies several subsets of C, the largest being CompCert C, to build a verified realistic compiler down to x86, PowerPC and ARM assembly. Our work relies on the formal semantics of Clight [8], a subset of C specified by CompCert.

The formal semantics of various subsets of C in CompCert have allowed Appel et al. to develop Verifiable C [3], a subset of C equipped with a powerful program logic in Coq. Certified implementations in Verifiable C include the SHA-256 encryption algorithm [2] and OpenSSL HMAC [6]. However, Verifiable C’s program logic provides no specific support for reasoning with floating-point numbers, so that examples of proofs of C programs with floating-point computations distributed with Verifiable C state no properties about their real-number semantics. The goal of our work is precisely to provide such specific floating-point support to the formal verification of C programs against a formal specification of Clight.

Another formal semantics of C not based on Coq is Ellison and Rosu’s, which formalized a more comprehensive subset of C using the K verification framework [20], from which they derived a program verification tool based on model-checking. However, they have not used it to verify any program with floating-point computations, since their formal semantics does not fully specify IEEE 754 floating-point computations (they are using the implementation provided by K itself instead), which “is fine for interpretation and explicit state model checking, but not for deductive reasoning.”

Combining Floqc with a Formal Semantics of C

Cocc actually solves the problems of trusting verifiers for floating-point computations in C programs, using a combination of Floqc with a formal semantics of a subset of C such as CompCert.

However, in practice, combining Floqc with CompCert was not first meant with source-level program verification in mind. Indeed, such an unified setting was first meant for compiler verification, namely formal verification of semantics-preserving optimizations of floating-point computations [12]. In their work, Boldo et al. focus on the semantics preservation of floating-point computations down to the bit-wise representation of floating-point numbers, thus permitting some of those operations to be implemented using integer operators instead. Their representation preservation covers infinities and also NaN (“not a number”) cases. In other words, their view of floating-point numbers is merely in terms of low-level raw bits rather than their high-level real-number meaning. So, whereas Boldo et al.’s work shows the practicality of the Floqc-CompCert combination on the compiler verification side, we show it on the source program verification side.

Jourdan et al. developed Verasco [24], a verified static analyzer for C programs. Verasco allows the user to annotate their program

16
with assumptions and assertions in such a way that it can be, subse-
quently, automatically proven not to go wrong (no divide by zero,
no mishandling of infinities or NaNs). This is the premise neces-
sary for verified compilers such as CompCert to ensure that com-
pliation preserves the semantics of the C program. Verasco is wholly
proved in Coq and fully supports CompCert C#minor (a subset of
C with side effect-free expressions and a weaker type system), in-
cluding floating-point computations. However, although Verasco’s
floating-point analysis is also based on combining Flocq with a for-
mal semantics of a subset of C, it does not support error analysis
with respect to the corresponding real-number computations. In-
deed, since such property is based on true real numbers, it cannot be
even stated in Verasco’s assertion language. In fact, any such prop-
erty is not useful for Verasco’s particular purpose of showing that a
C#minor program cannot go wrong, so Verasco handles floating-
point computations with a purely floating-point interval analyzer
without the need for a real-number interpretation.

By contrast with these works, our approach allows the formal
verification of implementations of C programs against a formal
semantics of C to prove functional correctness properties about the
real-number values of the floating-point computations performed
by such C programs, such as approximation or rounding error
analysis.

3. Floating-Point Computations in C

Our setting is based on the CompCert Clight language, which
allows us to design a faithful view of C floating-point expressions
and their semantics, which we describe in this section.

3.1 The Source Language: CompCert Clight

We assume that our program is written in the CompCert Clight
subset of C where expressions have no side effects and each func-
tion call is isolated as a standalone statement. In this case, CompCert
Clight expressions are pure and deterministic.

However, it may not be totally obvious to assess the trustworthi-
ness of the formal semantics of CompCert Clight expressions with
respect to the actual semantics specified by ANSI C. In fact, Boldo
et al. [12, §3] describe a more comprehensive floating-point seman-
tics for CompCert C, a nondeterministic subset of ANSI C that is
much larger than Clight and that is actually the top-most source
language of CompCert. Then, the trustworthiness of the CompCert
C semantics of floating-point expressions is transported to Clight
thanks to the fact that CompCert C is compiled to Clight in a
provably semantics-preserving way as part of CompCert’s frontend
[27].

The basic principle of floating-point computations in C, as cor-
correctly specified by CompCert C and Clight, is that every binary op-
eration is performed in the higher of the two precisions of its argu-
ments, regardless of the precision actually expected when reusing
the result in another expression. Consider for instance the following
C code:

```c
float x = ... ; float y = ... ; double z = x + y;
```

Then, the sum `x+y` is first performed in single-precision (due to `x`
and `y` being single-precision floating-point arguments) before being
cast to `double` when stored to `z`. To enforce its computation in
double-precision, the user would have to explicitly cast either of
the two arguments to `double`, and then the other argument would
be implicitly cast to `double`.

The formal semantics of Clight defines expression evaluation
rules as a big-step semantics. CompCert Clight’s expression evalua-
tion rules are described in extenso in Blazy et al.’s paper [8, §2.2,
§3.2] as well as in the Coq development of CompCert [27].

3.2 A Core Floating-Point Calculus for C

In this section, we describe our view of floating-point computa-
tions, and we prove that it is consistent with the semantics of Com-
pCert Clight floating-point expressions.

Following the IEEE 754 Standard [1] as specified in Flocq [9],
a finite floating-point number of precision `prec ∈ ℕ` and exponent
range \((emin, emax) ∈ ℤ^2\) is a number that can be represented in
one of the two following ways:

- either \((-1)^s × 2^{(prec-1)} × (2^{prec-1} + m) × 2^e\) with the sign
  bit \(s ∈ \{0, 1\}\), the significand \(m ∈ ℕ \cap [0, 2^{prec-1}]\), and the
  exponent \(e ∈ ℤ \cap \{emin, emax\}\). In this case, the floating-point
  number is said to be normal (or normalized).

- or \((-1)^s × 2^{(prec-1)} × m × 2^{emin}\), with the sign bit \(s ∈ \{0, 1\}\),
  the significand \(m ∈ ℕ \cap [0, 2^{prec-1}]\), and the exponent equal
to \(emin\). In this case, the floating-point number is said to be
denormal (or denormalized).

These two cases can be merged into a unique case \((-1)^s × m′ × 2^e\)
with the sign bit \(s ∈ \{0, 1\}\), the mantissa \(m′ ∈ ℕ \cap [0, 2^{prec}]\),
and the exponent \(e ∈ ℤ \cap \{emin − prec, emax − prec\}\), with the
boundary between normal and denormal numbers being at
\(m′ = 2^{prec-1}\) and \(e = emin\).

Our VCFLOAT framework supports typed floating-point computa-
tions. We describe the type of floating-point numbers of precision
`prec` and exponent range \((emin, emax)\) as the pair of integers
\((prec, emax)\) with \(emin = 3 − emax\). We assume that
\(2 ≤ prec < emax\). For IEEE 754 floating-point numbers, the
double-precision type is represented by \((53, 1024)\) whereas the
single-precision type is represented by \((24, 128)\). Our implemen-
tation is actually generic in the type for future support of IEEE
extended double precision (C’s `long double`) floating-point num-
bers, which can be easily supported by Flocq\(^1\), but currently not by
CompCert.

VCFLOAT automatically transforms every Clight floating-point
expression into a term \(t\) of the grammar defined in Figure 1 (where
we assume \(V\) is an infinite set of variables, and \(F\) represents the
type of IEEE 754 floating-point literals as specified in Flocq). \(Q/\)
represents the rounded square root. A floating-point term \(t\) is valid
if, and only if, for all typed constants \((f, \tau)\) appearing in \(t\), \(f\) is a
floating-point number of type \(\tau\). From now on, we only consider
valid floating-point terms.

Every floating-point term \(t\) has a type \(\langle t\rangle\) defined in Figure 2.
The type of a typed constant or variable is explicitly given. The type
of a (rounded or exact) unary operation is the type of its operand,
except for a cast to some type \(\tau\) which has type \(\tau\). For (rounded or
exact) binary operations between two operands of respective types
\(\tau_1, \tau_2\), the type of the binary expression is the least upper-bound

\(^1\)Flocq is even more generic, since it is also generic in the radix itself, to
accomodate radix-10 floating-point numbers specified in IEEE 754:2008 and
used in some financial contexts.

---

Flocq is even more generic, since it is also generic in the radix itself, to
accomodate radix-10 floating-point numbers specified in IEEE 754:2008 and
used in some financial contexts.
that, for any typed variable
and implement practical automation mechanisms to sidestep all
proved consistent with the semantics of C expressions, we design
On top of our core calculus of floating-point terms, which we
for infinities and NaNs.
of floating-point semantics, which means that the result also holds
in the semantics of Clight expressions.
Such casts introduced in the semantics of the floating-point terms of
least upper-bound type prior to computing the operation, following
choosing the floating-point number with the even significand.

Figure 3: Real-number semantics of VCFloat floating-point terms

\[
\begin{align*}
(\langle f, \tau \rangle)_\rho &= f \\
(\langle t_1 \oplus t_2 \rangle)_\rho &= \text{rem}_{(t_1)\cup(t_2)}(\langle t_1 \rangle_\rho \oplus (t_2)_\rho) \\
(\sqrt{t})_\rho &= \sqrt{[t]_\rho} \\
(-t)_\rho &= -[t]_\rho \\
\end{align*}
\]

Figure 2: Type of floating-point terms

\[
\begin{align*}
\tau_1 \sqcup \tau_2 &\text{ defined as:} \\
&\{\text{max}(\text{prec}_1, \text{prec}_2), \text{max}(\text{emax}_1, \text{emax}_2)\} \\
\end{align*}
\]

Given an environment \( \rho : V \rightarrow \mathbb{F} \), the semantics \([t]_\rho\) of a floating-
point term \( t \) is a floating-point number of type \( \tau \) (if we assume
that, for any typed variable \( \langle v, \tau \rangle \) appearing in \( t \), \( \rho(v) \) is a floating-
point number of type \( \tau \), as defined in Figure 3.

If \( x \) is a real-number and \( \tau \) is a floating-point type, then we
use CompCert’s choice of the rounding operator \( \text{rem}(x) \), namely the
floating-point number of type \( \tau \) nearest to \( x \), with ties broken to
choosing the floating-point number with the even significant.

As the general principle of the real-number semantics of IEEE
floating-point computations, as specified by Flocc, rounded operations
are first computed in real numbers then rounded to the desti-
nation type. For binary operators, both operands are first cast to the
least upper-bound type prior to computing the operation, following
the general principle of C floating-point computations. Such casts are
actually innocuous for finite floating-point numbers (their real-
number values are not changed) but we include them to preserve the
C semantics of floating-point computations for all floats (including
infinities and NaNs) as specified by Flocc [9].

In fact, we define the semantics of our floating-point terms di-
rectly using the floating-point operators defined in Flocc; their real-
number semantics explained in Figure 3 is actually based on theo-
rems (already proven in Flocc) valid only if no overflow occurs.

Then, we have proved the following theorem:

**Theorem 1.** Every CompCert Clight floating-point expression
has the same semantics as its transformed floating-point term of
VCFloat.

This theorem is important in particular to show that the type
casts introduced in the semantics of the floating-point terms of
VCFloat are consistent with the implicit casts and type promotions
in the semantics of Clight expressions.

In our Coq development, we have proved this theorem in terms of
floating-point semantics, which means that the result also holds
for infinities and NaNs.

4. **VCFloat: From Floating-Point to Real-Number Expressions**

On top of our core calculus of floating-point terms, which we
proved consistent with the semantics of C expressions, we design
and implement practical automation mechanisms to sidestep all
floating-point-specific reasoning, which we describe in this section.

Given a floating-point term \( t \), we first compute all constant
subexpressions of \( t \) using Flocc and replace the corresponding
subterms with their constant results.
set \( W \subseteq V \times \mathbb{R} \) of pairs of variables and real-number intervals, so that the following correctness theorem holds:

**Theorem 3.** Let \( u \) be an annotated term and \( \rho \) be an environment such that, for every typed variable \((v, \tau)\) appearing in \( u \), \( \rho(v) \) is a finite floating-point value of type \( \tau \).

Assume \( R(u, \rho) = (x, W, P) \). Then, if the validity condition \( P \) holds, then the real-number \( x \) is equal to the real-number value of the floating-point semantics \( [u]_\rho \). In other words, we have:

\[
\forall W, P \Rightarrow (\exists W, [u]_\rho = x)
\]

where, if \( W = \{(x_1, I_1), (x_2, I_2), \ldots\} \), then we write \( \exists W, P \) for \( \exists x_1 \in I_1, \exists x_2 \in I_2, \ldots \), and similarly for \( \forall \).

The generated conditions and rounding expressions are summarized in the table in Figures 6 and 7. The validity conditions enforce the following principles:

- operations must not overflow in the target type.\(^3\)
- for rounded operators where the real-number result before rounding is \( r \), the rounded term is of one of the following three forms, as formalized in Floção [9]:
  - \( r \times (1 + \delta) \) if annotated by Norm (i.e. the result is expected to be a normal number)
  - \( r + \epsilon \) if annotated by Deno (i.e. the result is expected to be a denormal number)
  - \( (r \times (1 + \delta)) + \epsilon \) if annotated by Unkn (i.e. we don’t know)
- Sterbenz’s condition must hold for Sterbenz’s exact subtraction
- multiplying or dividing with a power of two must not gradually underflow (i.e. flush to a denormal number)
- cast to a greater type introduces no rounding error

The validity condition \( P \) is a sufficient condition for the soundness of the rounding error terms, in particular in the case where those can be optimized away. The validity condition for an expression accumulates the validity conditions of all of its subexpressions.

We write \( W_1 \sqsubseteq W_2 \) to denote disjoint union.\(^4\)

**Our Tactic** We have implemented a Ltac tactic which, from a non-annotated floating-point term \( t \) to be evaluated in a given environment \( \rho \), automatically generates an annotated term \( u \), its real-number semantics \( x \) and a Coq proof term \( \pi \) such that \( \langle u \rangle = t \) and \( R(u, \rho) = (x, W, P) \) and \( \pi \) is a proof of \( \forall W, P \). Our tactic produces the proof term \( \pi \) by automatically checking the validity conditions defined in Figures 6 and 7 on the fly.

For each annotation of one operation, its subexpressions are first recursively annotated and their corresponding real-number expressions computed. Then, the validity condition for one possible annotation for the considered operation can be checked using the real-number expressions computed from the already annotated subexpressions.

For one operation, once its subexpressions have already been annotated:

- For a subtraction, Sterbenz’s condition is checked first
- For a product or a quotient, constants are first checked to be equal to a power of 2

\( \text{Indeed, if any operation overflows, it is flushed to a floating-point infinity, which is not representable in Coq real numbers and cannot be reasoned upon. Our VCFLOAT framework only focuses on finite floating-point numbers, which can be associated to a meaningful real-number value.} \)

\( \text{In our Coq formalism, we implemented R with an additional argument and an additional result to record the domain of variables that are already used, to ensure that such unions are always disjoint} \)

---

**Figure 6:** Real-number semantics and validity conditions for floating-point annotated terms: rounded cases

| R(R(u, \rho)) = (f, \delta, True) | R(R(u, \rho)) = (f, \delta, True) |
| R(R(\neg u, \rho)) = (\neg x, W, P) | R(R(\neg u, \rho)) = (\neg x, W, P) |
| where | where |
| R(R(\neg u, \rho)) = (\neg x, W, P) | R(R(\neg u, \rho)) = (\neg x, W, P) |
| and | and |
| R(\neg u, \rho) = (\neg x, W, P) | R(\neg u, \rho) = (\neg x, W, P) |
| and | and |
| \( [u]_{\rho, \text{prec}, \text{enorm}} = \langle x, W, P \rangle \) | \( [u]_{\rho, \text{prec}, \text{enorm}} = \langle x, W, P \rangle \) |
| and | and |
| \( \text{prec} \leq \text{enorm} \) | \( \text{prec} \leq \text{enorm} \) |
| and | and |
| \( \text{enorm} \leq \text{enorm} \) | \( \text{enorm} \leq \text{enorm} \) |
| R(\text{round}(x, W, P \wedge | R(\text{round}(x, W, P \wedge |
| (\text{No overflow}) = \text{round}(x, W, P \wedge | (\text{No overflow}) = \text{round}(x, W, P \wedge |
| where | where |
| R(\text{round}(x, W, P \wedge | R(\text{round}(x, W, P \wedge |
| and | and |
| R(\text{round}(x, W, P \wedge | R(\text{round}(x, W, P \wedge |
| and | and |
| \( n \) | \( n \) |
| R(\text{round}(x, W, P \wedge | R(\text{round}(x, W, P \wedge |
| (\text{No gradual underflow}) = \text{round}(x, W, P \wedge | (\text{No gradual underflow}) = \text{round}(x, W, P \wedge |
| where | where |
| R(\text{round}(x, W, P \wedge | R(\text{round}(x, W, P \wedge |
| and | and |
| R(\text{round}(x, W, P \wedge | R(\text{round}(x, W, P \wedge |
| and | and |
| \( n \leq 0 \) | \( n \leq 0 \) |

**Figure 7:** Real-number semantics and validity conditions for floating-point annotated terms: exact cases

| R(u_{\text{Sterbenz}}(x, W, P \wedge | R(u_{\text{Sterbenz}}(x, W, P \wedge |
| (\text{Sterbenz’s condition}) = (r_1 - r_2, W_1 \sqsubseteq W_2, P \wedge | (\text{Sterbenz’s condition}) = (r_1 - r_2, W_1 \sqsubseteq W_2, P \wedge |
| where | where |
| R(u_{\text{Sterbenz}}(x, W, P \wedge | R(u_{\text{Sterbenz}}(x, W, P \wedge |
| and | and |
| R(u_{\text{Sterbenz}}(x, W, P \wedge | R(u_{\text{Sterbenz}}(x, W, P \wedge |
| and | and |
| \( r_1 / r_2 \leq r_1 \leq r_2 \times 2 \) | \( r_1 / r_2 \leq r_1 \leq r_2 \times 2 \) |
• Otherwise, Norm, Deno, Unkn are tried in this order, stopping at the first success. If all fail, then it means that overflow cannot be ruled out, and so the overall tactic fails: no other annotations are tried for subexpressions.

Once an annotation is successfully found for one operation, it is no longer changed, so the total number of checks is linear in the number of operations (tree nodes) in the expression.

Our tactic automatically checks the corresponding validity conditions using Coq-Interval [30, 31], a Coq proof and tactic library for automatic interval arithmetic. The implementation of VCFloat with the connection to CompCert Clight totals less than 8,000 lines of Coq code (3,000 lines of specification and tactics and 5,000 lines of proof).

Examples
Our tactic allows to automatically sort out all floating-point issues (namely the absence of overflow and the shape of rounding error terms). Consider the following C expressions:

- \( 2.0f \times \text{(float)} \ x - 3.0 \), where \( x \) is a double-precision floating-point number known to have a value between 1 and 2. Then, VCFloat first automatically converts this expression to the non-annotated core expression \( (2 \times [x]) \in [2, 128] \) \( \otimes [x] \in [2, 128] \). Then, \( [x] \in [2, 128] \) is automatically annotated with Norm since \( x \) is large enough to be represented by a normal floating-point number. Then, the product \( 2 \times [x] \in [2, 128] \) is automatically annotated to become \( 2^1 \times [x] \in [2, 128] \). Then, the subtraction \( 2^1 \times [x] \in [2, 128] \) is automatically annotated to become \( 2^1 \times [x] \in [2, 128] \). Finally, VCFloat automatically computes the real-number expression for this fully annotated expression as \( \frac{2 \times (x \times (1 + \delta)) - 3}{\delta} \) where \( \delta \) is a free variable representing some unknown real number in \(-2^{-24}, 2^{-24}\).

- \( \text{DBL}_\text{MAX} \times (x + 0.5) \), where \( x \) is a double-precision floating-point number known to have a value greater than 0.5. Then, VCFloat first automatically converts this expression to the non-annotated core expression \( (2 \times [x]) \in [2, 128] \) \( \otimes [x] \in [2, 128] \). Then, \( x \otimes 5 \in [2, 128] \) is automatically annotated with Norm since \( x + 0.5 \) is large enough to be represented by a normal floating-point number. However, the product \( (2 \times [x]) \in [2, 128] \) cannot be annotated since overflow cannot be ruled out. So, the tactic immediately and unrecoverably fails, without even trying Unkn for the subexpression \( x \otimes 5 \in [2, 128] \).

Thanks to VCFloat, the user can directly reason on the real-number value of a Clight floating-point computation, as we illustrate with our complete program analysis example, which we describe in the next section.

5. Application: Certified Energy-Efficient Radar Image Processing

We apply our VCFloat framework to the completeverification of an energy-efficient C implementation of a radar image processing algorithm, namely Synthetic Aperture Radar (SAR) [35] image backprojection [19]. Our high-level goal is to estimate certified error bounds introduced by floating-point computations, and to evaluate their variations when introducing approximations and reducing precision for some floating-point computations. Indeed, since image processing on embedded radar systems involves heavy numerical computations onboard energy-constrained platforms, we hereby want to show that practical energy-efficient optimizations in floating-point computations can be achieved with provably bounded noise, thus providing some strong formal guarantee on the quality of the synthesized radar image.

In classical radar imaging, the image resolution is limited by the aperture of the physical antenna. To relax this constraint, Synthetic Aperture Radar (SAR) allows simulating much larger apertures by embedding classical radar sensors onto a platform onboard a plane flying over the target zone to be imaged. Then, the radar periodically sends pulse signals down to the ground target and measures the amplitude and phase of each returned pulse signal, all along the flight path of the plane. The data collected from these multiple platform locations about the same target zone is then reprocessed by software to synthesize an image. Backprojection is such a software image reconstruction algorithm for SAR; the general algorithm for backprojection is described in Figure 8 (which also defines the notations of constants in this section.)

SAR image backprojection actually depends on a binSample function, which interpolates a sample from the discrete measured sensor data. In our case here, we choose linear interpolation (see Figure 9).

The goal of our verification project is to compute upper bounds on the signal-noise ratio (SNR) of the synthetic image computed by SAR backprojection with respect to a gold-standard image \( \text{image}_g \): $\text{SNR} := \frac{\|\text{image}_g\|^2}{\|\text{image} - \text{image}_g\|^2}$, where \( |M| \) is the 2-norm of matrix \( M \), defined by $|M| := \sum_{x, y} |M[x, y]|^2$. A human-readable value of SNR is expressed in dB (i.e., $10 \log \text{SNR}$).

To estimate the noise actually introduced by floating-point computation roundings and approximations, we assume that the gold-standard image is computed with the real number algorithm of SAR.
backprojection with linear interpolation\textsuperscript{5}, and the actual image is computed with our implementation in floating-point number with approximations for square root and sine.

Since we have no information on the sensor data or platform position, we have no information on the “signal” (i.e., the numerator) part of the SNR, so we are interested in an upper bound on the denominator. It is straightforward to see that, if for all pixels, $|\Re(\text{image}_u[y][x]) - \text{image}_u[y][x]| \leq \varepsilon$ and $|\Im(\text{image}_u[y][x]) - \text{image}_u[y][x]| \leq \varepsilon$, then:

$$\|\text{image} - \text{image}_u\|^2 \leq 2 \times \text{BP}_\text{NPIX} \times \text{BP}_\text{NPIX} \times \varepsilon^2$$

since $|z|^2 = |\Re(z)|^2 + |\Im(z)|^2$. So it is enough to compute an absolute error bound $\varepsilon$ on the computation of the real or the imaginary part of all pulse contributions for one pixel image.

In this paper, we assume that the input data are exact, and we are only interested in the implementation error. Indeed, propagation of input data errors can be analyzed directly on the real algorithm, independently of any implementation. In particular for absolute error, when SAR backprojection is used in a larger context where input data is computed by another algorithm introducing some error, we simply add both the absolute propagation error and the absolute implementation error for SAR backprojection.

Overview of our Implementation and Proofs We implement SAR backprojection with linear interpolation in the CompCert Clight subset of C. In Figure 8, \textit{bin} is computed in double-precision floating-point numbers through an approximate, adaptive computation for the norm, which we study in detail in Section 5.1. Then, we perform linear interpolation (Section 5.2) in single-precision floating-point numbers. We compute the complex exponential using approximate sine and cosine, which we study in detail in Section 5.3. The contribution of one pulse is obtained by their product, which introduces several further rounding errors, which our VCFloat framework handles automatically. Finally, we compute the final sum of all pulse contributions for one pixel using naive summation, which we study in detail in Section 5.4. We summarize our overall bound on the noise introduced by our C floating-point implementation in Section 5.5. We finally comment on the energy savings of our code in Section 5.6.

Our implementation totals more than 120 lines of C code. We apply VCFloat to compute and prove error bounds using Coq 8.5beta2 and the trunk version of Coq-Interval\textsuperscript{31}.

Beyond numerical bounds, we assume nothing more on the input data — in particular, no statistical argument. The reason is that we want to construct proofs of worst-case error bounds. Also, Coq and most other proof assistants provide poor support for statistical or probabilistic reasoning, if any at all. Moreover, it is known\textsuperscript{32} that floating-point rounding errors are not random (i.e., floating-point computations over uniformly distributed floating-point values do not yield uniformly distributed floating-point errors), which makes statistical reasoning about rounding errors and their propagation even harder, even on paper.

5.1 Approximate Norm To save energy while preserving the quality of the synthesized radar image, we introduce a tradeoff between the amount of computations performed by the program and the resulting precision. To this end, we compute the norm using a Taylor approximation for square root.

In this subsection, we are studying the total absolute error bound introduced by our implementation of the square root for the norm. This error contains three sources of error:

\begin{itemize}
  \item Propagation of the computation errors introduced in the computation of the squared norm
  \item Method error introduced by the Taylor approximation
  \item Rounding errors introduced by the actual floating-point computations
\end{itemize}

\textbf{Error Propagation.\textsuperscript{}} Coq and its standard library allow us to easily provide a formal proof of $\sqrt{x^2} - \sqrt{y^2} = (x' - x)/(\sqrt{x^2} + \sqrt{y^2})$. Using this rewriting, since we already know a bound on $x' - x$ (which is the error to propagate) and individual bounds on $x$ and $x'$, we avoid correlation problems and thus we can directly use Coq-Interval\textsuperscript{31} on the rewritten expression to derive a bound on the propagation of the argument error in the square root.

\textbf{Method Error.\textsuperscript{}} Instead of always computing the square root using the standard square root function specified by IEEE 754 and implemented by the standard mathematical C library, we approximate the square root with a second-order Taylor polynomial $S$ around some $x_0$ defined as $S = \sqrt{x_0} + \frac{x - x_0}{2 \sqrt{x_0} - \frac{1}{2} \frac{(x - x_0)^2}{x_0^{3/2}}}$ where $\sqrt{x_0}$ was previously computed and memorized.

Based on the univariate Taylor theorem with mean value formalized in CoqApprox\textsuperscript{14}, we know that $\sqrt{x} - S = \frac{(x - x_0)^3}{16 \times (x_0 - \sqrt{x_0})^5}$ for some $\xi \in [x_0, x] \cup [x, x_0]$, which allows us to prove the following method error bound:

\textbf{Lemma 4.\textsuperscript{}} On the half-line of positive numbers, consider the disc of some radius $\tau_{\text{S2}} > 0$ centered on some point $x_0 > \tau_{\text{S2}}$. For any $x > 0$ within this disc (i.e. $|x - x_0| \leq \tau_{\text{S2}}$), we have $|\sqrt{x} - S| \leq \frac{(\tau_{\text{S2}})^3}{16 \times (x_0 - \tau_{\text{S2}})^5} x$.

\textbf{Rounding Errors and C Implementation.\textsuperscript{}} We now compute and certify an absolute rounding error bound in the evaluation of our C implementation of $S$ for $|x - x_0| \leq \tau_{\text{S2}}$, assuming that $\sqrt{x_0}$ is not computed exactly but accurately rounded in double-precision floating-point numbers like other arithmetic operations in the polynomial evaluation.

To minimize the number of computations and thus both energy consumption and the number of potential sources of rounding errors, we compute $S$ as follows:

$$n_0 = \frac{1}{\sqrt{x_0}}$$

$$u = \sqrt{x_0} \times (2 \times n)$$

$$S = \sqrt{x_0} + u \times (1 - n)$$

Our VCFloat framework applied to the implementation of this algorithm automatically highlights the following rounding errors:

\begin{itemize}
  \item VCFloat determines that $4x_0$ is computed exactly (since 4 is a power of 2). So the only rounding error introduced in the computation of $n_0$ is the one introduced for $1/(4x_0)$.
  \item From the bounds on $x$ and $x_0$, VCFloat determines that the hypotheses of Sterbenz’s theorem are satisfied, so $(x - x_0)$ introduces no rounding error. Thus, the computation of $n$ only introduces one new rounding error, in addition to the one in $n_0$.
  \item Since multiplying by 2 never introduces rounding error, the computation of $u$ introduces only one further rounding error.
  \item Finally, three further rounding errors are introduced by the computation of $S$.
\end{itemize}

We implement approximate square root in an adaptive fashion: the value of $x_0$ is not determined upfront but may change during the computation. The code of our Clight implementation is shown in Figure 10. We choose to dynamically adapt the computation of $x_0$; we assume that at any point, the correctly rounded value $s_0$ of $\sqrt{x_0}$ is available, as well as the computed value of $n_0(x_0)$, and we use them with our Taylor approximation for all $x$ until $|x - x_0| \geq 2 \tau_{\text{S2}}$. In that case, we replace $x_0$ with $x$ and compute...
5.2 Linear Interpolation

The next step in the SAR backprojection algorithm is to obtain the measured signal corresponding to the current pixel and pulse. However, the signal is measured on a discrete, regularly-spaced set of antennas, represented in software as an integer-indexed array. To obtain the signal, we would need to index the array using the range bin value corresponding to each pixel and pulse. Since this value is not necessarily an integer, we need to interpolate the signal from the next higher and lower integer values. In this work, we use a linear interpolation method.

The computation of the range bin introduces some error. Such error may introduce a non-continuous error in the computation of $\theta$ and for any pulse $p$, we have the following “smoothness” condition on the data:

$$|\Re(\text{data}[p][i + 1]) - \Re(\text{data}[p][i])| \leq \frac{2^{\text{N,\text{REAL}}} - 1}{2^{\text{N,\text{REAL}}} - 1}$$

5.3 Approximate Sine

In this section, we discuss the derivation of polynomial approximations to $\sin \theta$ and $\cos \theta$ that minimize the maximum error over the entire domain $\mathbb{R}$. Because of this minimax property, these polynomials converge uniformly with small error in relatively few terms. This makes them good candidates for establishing tight theoretical error bounds on polynomial approximations.

**Argument Reduction**

Accurate implementations of argument reduction for trigonometric functions have been proved in Gappa [29] in the context of the development of the CRlibm [33] correctly rounded mathematical library. However, such implementations can be energy-costly. Since we are ready to trade some accuracy for energy-efficiency, as long as the accuracy loss can be provably bounded, we can focus on certifying the error of a naive implementation of argument reduction.

Our C implementation for argument reduction is presented in Figure 11. Our VCFloat framework addresses most of its floating-point computations automatically except calls to ISO C99 copysign which is not part of CompCert’s supported built-in floating-point operations. The purpose of using copysign is to manipulate the sign bit of a floating-point number while avoiding tests and cache misses in the C implementation. In particular, for our purposes, we have manually proved that the sign bit of a product is still meaningful if the product overflows. So, through our proof using VCFloat and Coq-Interval [31], we found that our naive argument reduction introduces absolute error at most $3567 \cdot 2^{-33}$ for initial argument of magnitude less than $2^{30}$, due to both rounding errors and approximations of $\pi$.

**Polynomial Approximation**

We now concentrate on finding the polynomial of order $N$ that minimizes the maximum deviation
Figure 11: Argument reduction for sine/cosine

from sin θ for arguments in the restricted domain [0, π/2], while satisfying boundary conditions on the value and the first derivative at the two endpoints. This ensures an approximation that is everywhere smooth and continuous, can never go outside the valid range (−1, 1), and (with the additional error introduced by argument reduction) is accurate for any argument in ℝ as the worst case error in the domain [0, π/2]. This error is to be minimized by choice of fit coefficients.

We will denote our polynomial approximation of order N by

\[ s(x|N,c) \]

where x denotes an argument in the restricted domain, N denotes the order of the polynomial, and c denotes the vector \([c(0), c(1), c(2), \ldots, c(N)]\) of polynomial coefficients, so that we have

\[ s(x|N,c) = \sum_{n=0}^{N} c(n) x^n \]

leading to the following identity for its derivative:

\[ s'(x|N,c) = \sum_{n=1}^{N} c(n) n x^{n-1} \]

Setting the value to zero and the slope to one at \(x = 0\), and the value to 1 and the slope to zero at \(x = \pi/2\) provides four linear equations of constraint in the \(N + 1\) coefficients \(c(n)\).

The four equations of constraint in \(N\) unknown polynomial coefficients can be written as \(Ac = b\), where \(c\) is a column vector containing the polynomial coefficients, \(b\) is a \(4 \times N\) matrix, and \(A\) is the \(4 \times N\) matrix

\[
A = \begin{bmatrix}
1 & 0 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & 0 & \cdots & 0 \\
1 & \frac{\pi}{2} & \left(\frac{\pi}{2}\right)^2 & \left(\frac{\pi}{2}\right)^3 & \cdots & \left(\frac{\pi}{2}\right)^{N-1} \\
0 & 1 & 2 & \left(\frac{\pi}{2}\right)^2 & 3 & \left(\frac{\pi}{2}\right)^3 & \cdots & (N-1) & \left(\frac{\pi}{2}\right)^{N-2}
\end{bmatrix}
\]

The constraints ensure that \(c(0) = 0\) and \(c(1) = 1\), so that the number of nontrivial coefficients is really \(N - 2\). The order 3 polynomial (\(N = 4\)) is of special interest because the polynomial is completely determined by the equations of constraint, since \(A\) is square.

In order to obtain polynomial fits of higher order, it is necessary to perform an optimization, since the system of linear equations is underdetermined if \(N > 4\). For purposes of establishing precision bounds, we minimize the maximum deviation of the fit polynomial from the true sin function subject to the four linear equations of

\[
\min \{ \max |s(x) - s(x|N,c)| \} \text{ subject to } Ac = b.
\]

This problem qualifies as a convex optimization problem, since the functional to be minimized is convex in the unknowns \(c\) and the equations of constraint are linear in the components of \(c\) [13]. Optimizations of this kind can be solved efficiently using modern convex solvers. For purposes of this work, we used the MATLAB routine SeDuMi [37] within the convex optimization framework CVX [22] to perform the needed optimizations.

The obtained polynomial for \(N = 7\), with absolute bound error 1.10186E−06, and coefficients rounded to single-precision, is:

\[
P_6(x) = ((((( −16392343/17179869184 \times x + 21895786/2147483648 \times x + 15145514/8589934592 \times x + 12264200/134217728 \times x + 17533087/137438953472 \times x + 1 \times x \times x)
\]

\[
Proof of C Implementation We efficiently derived polynomial approximations of sine through convex optimization. However, to derive such approximations, we used MATLAB-based tools relying on floating-point computations. This means that all computations performed in MATLAB, and in particular the coefficients of the polynomial approximations and their computed error bounds, bear some rounding errors. Moreover, although MATLAB claims to follow the IEEE 754 standard for floating-point computations, its rounding mode is not clearly specified and could very well conflict with the rounding mode used by the client C code. Those two sources of inaccuracies thus weaken the confidence in the error bounds computed using the method that we described in the previous subsection. So, they have to be further backed by a formal method, which we describe here. Moreover, the error bound that we obtained there is only the approximation method error, and ignores the rounding errors introduced by implementing the evaluation of the polynomial approximation with floating-point numbers, so we have to formally study them as well.

We integrated these polynomial approximations into our verification by computing and proving the correctness of the absolute error bounds for sine computed using the 6-order approximation \(P_6\) of sine obtained in Equation 2, and evaluated in single-precision floating-point. These absolute error bounds take into account both method error (replacing sine with \(P_6\)) and rounding errors (evaluating \(P_6\) in single-precision floating-point).

For our proof, we considered the argument range \([0, 8/5]\) which is a large superset of \([0, \pi/2]\). Actually, this argument range is a superset of \([0, 3373262642.2^{-31}]\) which is the range of the reduced argument that we computed before.

Coq-Interval [31] supports interval arithmetic with a small set of analytic transcendental functions, including sine. So, we can use Coq-Interval to compute and prove a bound of \(|P_6(x) - \sin(x)|\). We find that the absolute method error introduced is at most 2388 · 2−31 ≈ 1.12 · 10−10, similar to the bound empirically found in Equation 2.

Finally, our VCFloat framework allows us to find that our C implementation in single precision introduces absolute rounding error at most 1235 · 2−31. This means that the bound on the error introduced by the core computation is of similar amount of magnitude to the error bound for argument reduction.
to significantly improve the rounding error by a running compensation. Rounding errors for a wide range of floating-point summation algorithms are studied by Higham [23], but this survey is based on a naive rounding model which neglects gradual underflow. In the following, we describe our formal proof of a worst-case error bound for naive summation with both rounding errors in the presence of gradual underflow and propagation of summand errors.

Let \( q \in \mathbb{R}^n \) be a sequence of ideal real values, and \( Q_n+1 = \sum_{i=0}^n q_i \) be their ideal sum (\( Q_0 = 0 \)). Let \( \tilde{q} \) be the sequence of approximate values actually computed for each term of \( q \). Then, the approximate sum \( \tilde{Q} \) actually computed with further rounding errors \( \delta, \epsilon \) introduced at each step has the following shape: \( \tilde{Q}_0 = 0 \) and \( \tilde{Q}_{n+1} = (Q_n + \tilde{q}_n)(1 + \delta_n) + \epsilon_n \).

We want to find an absolute error bound for the computation of \( \tilde{Q} \), i.e., bound \( |Q_n + q_n| \).

Assume that \( q \leq Bq; \tilde{q} - q \leq B; |\delta| \leq B\delta, \) and \( |\epsilon| \leq B\epsilon \). Then, we can prove that \( |Q_n + q_n - \tilde{Q}_{n+1}| \leq |Q_n - q_n|M + nL + K \) where \( M = 1 + B; L = B + B; \) and \( K = L + B, M + B \).

Thus, we have: \( |\tilde{Q}_n - Q_n| \leq D_n \), where \( D \) is the recursive real sequence defined as \( D_0 = 0 \) and \( D_{n+1} = D_n + M + nL + K \), for which we prove that \( D_n = \frac{1}{1 - M} \left( K \frac{1}{1 - L} + \frac{n + 1}{1 - M} \right) \). Assuming \( M \neq 1 \) and \( 0^\delta = 1 \). Although we found this formula through a pen-and-paper proof based on formal derivation of polynomials, our actual Coq proof does not need such an argument and works out directly by induction on \( n \) using basic real field algebra.

It is interesting to know that as long as the bounds of the terms of the sum are uniform, the error bound on the sum does not depend on the order in which the terms are summed. However, our result assumes that the sum is computed linearly: it cannot apply to cases where partial sums are first computed and then summed up. In particular, our result does not apply to divide-and-conquer or other parallel summing strategies. For our implementation of SAR backprojection, this is not the case: whereas the contributions of any two different pixels can be computed in parallel since they are independent of each other, we have chosen to sequentially compute and sum the contributions of each pulse for one pixel.

### 5.5 Summary and Interpretation of Error Bounds

We first performed our proof in the case where all steps of the sum are computed in single precision. Given the low quality of the obtained bound, we decided to tackle the case where all steps of the sum are computed in double precision, but the final result is cast back to single precision. We further extended our proofs by replacing all or part of our approximations with standard mathematical functions assumed to be correctly rounded, and by studying the impact of the precision choice for the computation of the linear interpolation. The table in Figure 12 summarizes the absolute implementation error bound \( \epsilon \) on the real or imaginary part of all pulse contributions for one angle, and the upper bound on the denominator of the SNR \( \sum_{y, z} (\text{image}[y][z] - \text{image}[y][z])^2 \), computed as

\[
D = 2 \times \text{BP_NPIX}_X \times \text{BP_NPIX}_Y \times \epsilon^2.
\]

The bounds on the input data are taken from the PERFECT suite [4] for three image sizes (512 × 512, 1024 × 1024 and 2048 × 2048 pixels).

Our results show that if the steps of the sum are computed in single precision, then the accumulation of rounding errors introduced by the summation actually overwhelms all other implementation errors, and goes well beyond acceptable losses. However, if the steps of the sum are computed in double precision, the worst case for large images with approximations enabled introduces losses up to 44 dB, which is nearly acceptable in practice (if the PERFECT data suite arbitrarily sets the ideal SNR to 140 dB and SNR ranges around 100 dB and beyond are considered acceptable).

The implementation error grows with the size of the image and the number of pulses. This may seem surprising since images of bigger size with more pulses are supposed to reduce the error in practice, but error growth is actually due to the accumulation of implementation and rounding errors in the final sum. Thus in practice, the input data seems to have statistical properties prone to lower or cancel the error. We are not relying on any such statistical assumptions in our verification work, which is focused on a certified proof for worst-case error bounds. From the radar image processing point of view, it means that we do not assume anything on the behavior of the adversary with respect to terrain camouflage or signal scrambling. A more thorough analysis taking into account such statistical arguments might reveal potential weaknesses in the implementation, which an adversary might exploit to worsen the quality of the synthesized image.

### 5.6 Performance Measurements

To clearly assess the gain introduced by approximations, we measured the energy and power performance improvements introduced by our C implementation of SAR backprojection.

We performed performance tests on an Intel SandyBridge machine with 4 overclocked processors with 8 physical cores (i.e., 16 logical cores) each. We measure time and energy consumption for both the naive (accurate square root and sine) implementation and our approximate implementation, each both sequentially and in a parallel setting using OpenMP where we only use one processor and 8 cores of this processor. Unsounded floating-point optimizations such as associativity reorderings have been disabled. The results are summarized in Figure 13. We take our raw input data from the PERFECT data suite [4], which arbitrarily sets the ideal SNR to 140 dB. We show that our approximations cut energy consumption by nearly one half. Our results show that energy gains are mainly due to the speedup obtained by our approximate algorithms. This is fairly understandable since on SandyBridge machines, most floating-point operations consume the same amount of power.

We also performed time measurements on an Intel Haswell platform, shown in Figure 14. However, hardware floating-point counters are disabled on Haswell platforms. So, assuming that floating-point operations consume similar amounts of power (similar to SandyBridge machines), we can deduce energy consumption trends from time. For parallel executions, we show that our approximations cut time consumption by 9% to 18%.

### 6. Limitations and Future Work

Our VCFoot framework is specialized in real numbers with rounding error terms, so it is targeted to the verification of numerical C programs. However, it may not be totally suitable to the verification of implementations of elementary functions such as CRlibm [33], which may require reasoning about the actual significands and exponents of floating-point numbers.

The main limitation of VCFoot that we encountered during our proof of SAR backprojection is that we have not investigated the interplay between floating-point numbers and integers. In particular, linear interpolation needs to cast a floating-point number to an integer to index an array, which we currently address using a manual proof, after using VCFoot for the purely floating-point part. Conversely, integers would deserve to be more automatically handled by VCFoot as well, so we plan to take advantage of the fact that casting an integer of magnitude less than \( 2^{24} \) (resp. \( 2^{25} \)) to a single-precision (resp. double-precision) floating-point number does not modify its real-number value.

Also, on the practicality side, although we have drastically reduced the size of floating-point-related proof scripts, our total overall proof of our C implementation of SAR backprojection is still
more than 12,000 lines long mainly due to C language constructs. We have not focused on such constructs, since they can be mostly addressed by existing Coq program logics for C such as Verifiable C [3], which our proofs are not using. Thus, to further shorten our proofs, we plan to integrate VCFloat into Verifiable C, combined with linear interpolation.

7. Conclusion

To the best of our knowledge, our certified C implementation of SAR backprojection is the first example of a realistic C program with floating-point computations proven correct using a unified verification setting based on Coq only, against a specification involving error estimates in real-number values. By virtue of its small trusted computing base containing only the faithfulness of the formal specifications of C and floating-point arithmetic, the soundness of Coq’s underlying logic, and the correctness of the Coq proof checker, our work shows that it is possible to gain an unprecedented level of confidence in verified source C programs with floating-point computations. We claim that our work, once combined with Verifiable C [3], will broaden the verifiable features of C towards more complete programs. Overall, our work validates the approach of specifying extensive formal semantics for C including all of its “dark corners” such as floating-point computations for the purpose of source-level verification, thus confirming the validity of the following motto:

“always look on the dark side of C”

Acknowledgments

We thank Xavier Leroy and Jacques-Henri Jourdan for our insightful discussions, and our anonymous reviewers for their many valuable comments and questions. This work is sponsored in part by DARPA MTO as part of the Power Efficiency Revolution for Embedded Computing Technologies (PERFECT) program (issued by DARPA/CMO under Contract No: HR0011-12-C-0123). The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of the DARPA or the U.S. Government. Distribution Statement “A” (Approved for Public Release, Distribution Unlimited.) Technologies described in section 5 are patent-pending.

References
