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

Tahina Ramananandro, Paul Mountcastle, Benoît Meister, Richard Lethin
The High-Level Problem

Goal: energy-efficient implementations of radar algorithms

- Naïve implementations consume time and energy
- Ideas: compute in lower-precision floating-point and/or with further approximations
- Approximations and their floating-point implementations introduce some error in the result
- How to compute some implementation error bound?
- How can we trust this error bound?
Our Achievements

VCFloat: a Coq library for handling floating-point computations in the verification of C programs

- Automatically compute real-number expressions with rounding error terms and their correctness proofs

Use case: SAR backprojection with linear interpolation

- Introduce approximations for square root and sine
- Tune between single- and double-precision floating-points
- Compute error bounds wrt. “ideal” mathematical real-number algorithm
- **Formal proof** of correctness using the Coq proof assistant
- Energy measurements: ~10–20% saved on Intel Haswell
This Presentation

- Certified error bounds for energy-efficient radar image processing
- Our Coq framework: VCFloat
- Demo
- Conclusions
Certified Error Bounds for

RADAR IMAGE PROCESSING
Synthetic Aperture Radar (SAR) Backprojection

for all pixels $x$ do
  for all pulses $p$ do
    $R = \text{distance between platform and pixel } x \text{ for pulse } p$
    $s = \text{sample interpolated from pulse } p \text{ in neighborhood of range } R$
    Apply phase correction to sample $s$ based on range $R$
    Accumulate sample $s$ into pixel $x$
  end for
end for

Real-number algorithm

Figure from Park et al. *Efficient Backprojection-Based Synthetic Aperture Radar Computation with Many-Core Processors*, SC 2012
SAR Backprojection

Real-number algorithm

Figure from Park et al. Efficient Backprojection-Based Synthetic Aperture Radar Computation with Many-Core Processors, SC 2012
SAR Backprojection

Figure from Park et al. Efficient Backprojection-Based Synthetic Aperture Radar Computation with Many-Core Processors, SC 2012

Real-number algorithm

```
for y := 0 to BP_NPIX_Y - 1 do
  py := (y + 1 - BP_NPIX_Y) \times dxy
  for x := 0 to BP_NPIX_X - 1 do
    px := (x + 1 - BP_NPIX_X) \times dxy
    image[y][x] := 0 \in \mathbb{C}
  end for
  for p := 0 to N_PULSES - 1 do
    r := ||platpos[p] - (px, py, z[p][y][x])||
    bin := (r - r_0) / dr
    sample := binSample(N_RANGE_UPSAMPLED, data[p], bin)
    matchedFilter := \exp(2i \times ku \times r)
    image[y][x] := image[y][x] + sample \times matchedFilter
  end for
end for
return image
```
SAR Backprojection

Implementation

Floating-point computations

Square root

Sine

Figure from Park et al. Efficient Backprojection-Based Synthetic Aperture Radar Computation with Many-Core Processors, SC 2012
void backprojection
(int const BP_NPIX_X, int const BP_NPIX_Y,
int const N_PULSES, …,
float const **data_r, float const **data_i,
double** image_r, double** image_i, …) {
for (int y = 0; y < BP_NPIX_Y, ++y) {
  ...
  for (int x = 0; x < BP_NPIX_X, ++x) {
    ...
      for (int p = 0; p < N_PULSES, ++p) {
        ...
          double ... = ... sqrt(...) ...
          ...
          double ... = ... sin(...)
          ...
      }
  ...
}
void backprojection
(int const BP_NPIX_X, int const BP_NPIX_Y,
 int const N_PULSES, …,
 float const **data_r, float const **data_i,
double** image_r, double** image_i, …) {
 for (int y = 0; y < BP_NPIX_Y, ++y) {
   …
   for (int x = 0; x < BP_NPIX_X, ++x) {
     …
     for (int p = 0; p < N_PULSES, ++p) {
       …
       double … = … approx_sqrt(…) ;
       …
       double … = … approx_sin(…) ;
       …
     }
   }
 }
}
void backprojection(int const BP_NPIX_X, int const BP_NPIX_Y, 
int const N_PULSES, ...,
float const **data_r, float const **data_i,
float** image_r, float** image_i, ...) 
for (int y = 0; y < BP_NPIX_Y, ++y) 
  ... for (int x = 0; x < BP_NPIX_X, ++x) { 
    ... for (int p = 0; p < N_PULSES, ++p) { 
      ... double ... = ... approx_sqrt(...); ...;
      float ... = ... approx_sin(...); ...;
    ...}
  }
}

Figure from Park et al. Efficient Backprojection-Based Synthetic Aperture Radar Computation with Many-Core Processors, SC 2012
Image Error Analysis

Maximize Signal-Noise Ratio:

$$SNR := \frac{\|image_0\|^2}{\|image - image_0\|^2}$$

Find an upper bound on the denominator

• Absolute error bound is enough

Error sources:

• Method errors introduced by approximation
• Rounding errors introduced by floating-point computations
Final Correctness Statement (slightly simplified)

\[\forall P \ (\text{HYPS: SARHypotheses } P) \ m\]
\[\hspace{1cm} (\text{Hm: holds } m \ (P \ + + \ Pperm_int \ \text{bir oir (BP_NPIX_X \times BP_NPIX_Y)} \ + + \ Pperm_int \ \text{bii oii (BP_NPIX_X \times BP_NPIX_Y)}),\]

\[\exists m', \ \text{star Clight.step2}\]
\[\hspace{1cm} (\text{Callstate fn_sar_backprojection }...) \ (\text{Returnstate Vundef Kstop } m') \land\]

\[\exists \text{ image}_r \ \text{image}_i,\]
\[\hspace{1cm} \text{holds } m \ (P \ + + \ Parray_int \ \text{image}_r \ \text{bir oir (BP_NPIX_X \times BP_NPIX_Y)} \ + + \ Parray_int \ \text{image}_i \ \text{bii oii (BP_NPIX_X \times BP_NPIX_Y)}) \land\]
\[\forall y, (y < \text{BP_NPIX_Y})_\text{onat} \to \forall x, (x < \text{BP_NPIX_X})_\text{onat} \to\]
\[\hspace{1cm} \text{let } ir := \text{image}_r (y \times \text{BP_NPIX_X} + x) \ \text{in}\]
\[\hspace{1cm} \text{let } ii := \text{image}_i (y \times \text{BP_NPIX_X} + x) \ \text{in}\]
\[\hspace{1cm} \text{is_finite } \_ \_ \ ir = \text{true} \land \text{is_finite } \_ \_ \ ii = \text{true} \land\]
\[\hspace{1cm} \text{let } (tr, ti) := \text{SARBackProj.sar_backprojection } y \ x \ \text{in}\]
\[\hspace{1cm} \text{Rabs } (\text{B2R } \_ \_ \ ir - tr) \leq \text{pixel_bound} \land\]
\[\hspace{1cm} \text{Rabs } (\text{B2R } \_ \_ \ ii - ti) \leq \text{pixel_bound}.\]
Final Correctness Statement (slightly simplified)

∀ P ` (HYPS: SARHypotheses P) m
   (Hm: holds m (P ++
       Pperm_int bir oir (BP_NPIX_X × BP_NPIX_Y) ++
       Pperm_int bii oii (BP_NPIX_X × BP_NPIX_Y)),

∃ m', star Clight.step2
   (Callstate fn_sar_backprojection ...) (Returnstate Vundef Kstop m') ∧
∃ image_r image_i,
   holds m (P ++
       Parray_int image_r bir oir (BP_NPIX_X × BP_NPIX_Y) ++
       Parray_int image_i bii oii (BP_NPIX_X × BP_NPIX_Y)) ∧
∀ y, (y < BP_NPIX_Y)%nat → ∀ x, (x < BP_NPIX_X)%nat →
   let ir := image_r (y × BP_NPIX_X + x)) in
   let ii := image_i (y × BP_NPIX_X + x)) in
   is_finite _ _ ir = true ∧ is_finite _ _ ii = true ∧
   let (tr, ti) := SARBackProj.sar_backprojection y x in
   Rabs (B2R _ _ ir - tr) ≤ pixel_bound ∧
   Rabs (B2R _ _ ii - ti) ≤ pixel_bound.

Hypotheses on input data
Memory contents and permissions

Conclusions
Final Correctness Statement (slightly simplified)

\( \forall P \ (\text{HYPS: SARHypotheses } P) \ m \)

\( (Hm: \text{holds } m \ (P ++ \ Pperm\_int \ bir \ oir \ (BP\_NPIX\_X \times BP\_NPIX\_Y) ++ \ Pperm\_int \ bii \ oii \ (BP\_NPIX\_X \times BP\_NPIX\_Y)), \)

\( \exists m', \text{star Clight.step2} \)

\( (\text{Callstate fn_sar_backprojection } ...) \ (\text{Returnstate Vundef Kstop } m') \land \)

\( \exists \text{image_r image_i}, \)

\( \text{holds } m \ (P ++ \ Parray\_int \ image\_r \ bir \ oir \ (BP\_NPIX\_X \times BP\_NPIX\_Y) ++ \ Parray\_int \ image\_i \ bii \ oii \ (BP\_NPIX\_X \times BP\_NPIX\_Y)) \land \)

\( \forall y, (y < BP\_NPIX\_Y)\%\text{nat} \rightarrow \forall x, (x < BP\_NPIX\_X)\%\text{nat} \rightarrow \)

\( \text{let } ir := \text{image_r} \ (y \times BP\_NPIX\_X + x) \text{ in} \)

\( \text{let } ii := \text{image_i} \ (y \times BP\_NPIX\_X + x) \text{ in} \)

\( \text{is_finite } \_ \_ \ ir = \text{true} \land \text{is_finite } \_ \_ \ ii = \text{true} \land \)

\( \text{let } (tr, ti) := \text{SARBackProj.sar_backprojection } y \times x \text{ in} \)

\( \text{Rabs } (\text{B2R } \_ \_ \ ir - tr) \leq \text{pixel_bound} \land \)

\( \text{Rabs } (\text{B2R } \_ \_ \ ii - ti) \leq \text{pixel_bound}. \)
Polynomial Approximations of Sine

Built-in hardware sine is costly in energy and time

- Replace core sine with a polynomial approximation
  - Use convex optimization
  - Compute coefficients with unverified numerical tools
  - Do not trust the results, use Coq to prove an error bound

- Naïve argument reduction is enough for SAR
  - Errors due to approximation of $\pi$ and roundings
  - Lower than implementation error for core computation
Adaptive Approximate Square Root

- Replace square root with 2-degree Taylor polynomial
  - Taylor-Lagrange inequality bounds method error
- Valid only in a convergence disc
  - Outside, use accurate hardware square root
  - Adaptive algorithm: Re-center the disc as needed
Precision Results

- Input data bounds from DARPA PERFECT suite
- Error grows with image size
- No statistical reasoning about data

<table>
<thead>
<tr>
<th>Norm</th>
<th>Interpol.</th>
<th>Sine</th>
<th>Final sum</th>
<th>Small dB</th>
<th>Med. dB</th>
<th>Large dB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double</td>
<td>Double</td>
<td>Double</td>
<td>Doub/Sing</td>
<td>-2</td>
<td>4</td>
<td>10</td>
</tr>
<tr>
<td>Double</td>
<td>Single</td>
<td>Doub/Sing</td>
<td>Doub/Sing</td>
<td>3</td>
<td>15</td>
<td>27</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Double</td>
<td>Double</td>
<td>Doub/Sing</td>
<td>12</td>
<td>24</td>
<td>37</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Single</td>
<td>Doub/Sing</td>
<td>Doub/Sing</td>
<td>14</td>
<td>26</td>
<td>39</td>
</tr>
<tr>
<td>Double</td>
<td>Single</td>
<td>Approx.</td>
<td>Doub/Sing</td>
<td>17</td>
<td>29</td>
<td>40</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Single</td>
<td>Approx.</td>
<td>Doub/Sing</td>
<td>20</td>
<td>32</td>
<td>44</td>
</tr>
<tr>
<td>Double</td>
<td>Single</td>
<td>Doub/Sing</td>
<td>Single</td>
<td>34</td>
<td>52</td>
<td>70</td>
</tr>
<tr>
<td>Adaptive</td>
<td>Single</td>
<td>Approx.</td>
<td>Single</td>
<td>35</td>
<td>53</td>
<td>70</td>
</tr>
</tbody>
</table>
Performance Measurements for Optimized C Code

- Intel SandyBridge: direct energy measurements
- Intel Haswell: energy model unknown, time instead

Energy (J) on SandyBridge, large

Time % on Haswell, parallel

Time (s) on SandyBridge, large
SAR proof: facts and figures

C code size: 150 lines

Proof size:
- Previous all-manual proof: 26k lines, no connection with C
- Thanks to VCFLOAT: reduced to 12k lines
  - 5k lines of spec (loop invariants), 7k lines of proof
  - ~2k lines of proof for real-number reasoning
  - Remaining part due to C language constructs, could be further reduced when integrating with Verifiable C (Appel et al. 2014)

Proof building/checking time:
- 1 hour (4-core Intel Core i7, 2.10 GHz, 4 Gb RAM)
- Mostly due to interval computations
Formal Verification of Floating-Point Computations in C Programs

OUR COQ FRAMEWORK: VCFLOAT
Final Correctness Statement (slightly simplified)

∀ P `(HYPS: SARHypotheses P)` m
(Hm: holds m (P ++
  Pperm_int bir oir (BP_NPIX_X × BP_NPIX_Y) ++
  Pperm_int bii oii (BP_NPIX_X × BP_NPIX_Y)),

∃ m', star Clight.step2
  (Callstate fn_sar_backprojection ...) (Returnstate Vundef Kstop m') ∧
∃ image_r image_i,
  holds m (P ++
    Parray_int image_r bir oir (BP_NPIX_X × BP_NPIX_Y) ++
    Parray_int image_i bii oii (BP_NPIX_X × BP_NPIX_Y)) ∧
∀ y, (y < BP_NPIX_Y)%nat → ∀ x, (x < BP_NPIX_X)%nat →
  let ir := image_r (y × BP_NPIX_X + x)) in
  let ii := image_i (y × BP_NPIX_X + x)) in
  is_finite _ _ ir = true ∧ is_finite _ _ ii = true ∧
  let (tr, ti) := SARBackProj.sar_backprojection y x in
  Rabs (B2R _ _ ir - tr) ≤ pixel_bound ∧
  Rabs (B2R _ _ ii - ti) ≤ pixel_bound.

Hypotheses

C code runs

Memory contents

FP does not overflow

Total implementation error bound
(approximation + rounding)
computed at proof-building time
Final Correctness Statement (slightly simplified)

∀ P (HYPS: SARHypotheses P) m
  (Hm: holds m (P ++
    Pperm_int bir oir (BP_NPIX_X × BP_NPIX_Y) ++
    Pperm_int bii oii (BP_NPIX_X × BP_NPIX_Y))
  )

∃ m', star Clight.step2
  (Callstate fn_sar_backprojection ...) (Returnstate Vundef Kstop m')

C code runs

CompCert Clight

∃ image_r image_i,
  holds m (P ++
    Parray_int image_r bir oir (BP_NPIX_X × BP_NPIX_Y) ++
    Parray_int image_i bii oii (BP_NPIX_X × BP_NPIX_Y)) ∧

∀ y, (y < BP_NPIX_Y)%nat → ∀ x, (x < BP_NPIX_X)%nat →
  let ir := image_r (y × BP_NPIX_X + x) in
  let ii := image_i (y × BP_NPIX_X + x) in
  is_finite __ ir = true ∧ is_finite __ ii = true

Flocq

B2R

FP does not overflow

Total implementation error bound (approximation + rounding)
computed at proof-building time

Memory contents
Our Design Choices: Which Formal Methods?

Verification using the Coq proof assistant

Correctness fully embedded in Coq using existing libraries:

  - Formal semantics of a deterministic sequential subset of C
- Flocq (Boldo & Melquiond, ARITH 2009)
  - Formalization of floating-point numbers
- Coq standard library
  - Formalization of real numbers
Our Design Choices: Which Formal Methods?

We use Coq + CompCert Clight + Flocq.

Advantages for trust:

- Unified verification framework
  - OK to combine proof libraries

- Formalization in the Gallina mathematical language of Coq
  - Can be trusted more easily than practical implementations (e.g. Fluctuat, Frama-C/Why3, etc.)

- Coq is the only setting where C, floating-point and real numbers are trustworthily mixed together
Our Approach and our Trusted Computing Base

We use Coq + CompCert Clight + Flocq.

• What do we need to trust?
  – Coq's underlying logic is sound
  – Implementation of Coq is sound wrt. Coq's logic
  – Coq standard library real numbers are consistent and faithful
  – Clight is faithful wrt. the corresponding subset of ISO C99
  – Flocq is faithful wrt. IEEE 754-2008 floating-point numbers

• Formalizations in the Gallina mathematical language

• Can be assessed more easily than practical implementations of verification tools
Verification of C Floating-Point Expressions

2.0f * (float) x - 3.0;

C floating-point expression

Real-number semantics
Floating-Point Numbers

IEEE 754–2008 modelled by Flocq (Boldo et al. 2009)

• A binary operation $a \odot b$ is not computed exactly
• Rounded from its ideal value
  – Example rounding mode: rounding to nearest
• What is the shape of the rounding error?
Floating-Point Numbers

IEEE 754–2008 modelled by Flocq (Boldo et al. 2009)

\[ \pm m \cdot 2^e \quad 0 \leq m < 2^{\text{prec}}, \quad e_{\text{min}} \leq e \leq e_{\text{max}} \]
Floating-Point Numbers

IEEE 754-2008 modelled by Flocq (Boldo et al. 2009)

\[ \pm m \cdot 2^e \mid 0 \leq m < 2^3 = 8; -3 \leq e \leq -1 \]

Example: prec = 3, emin = -3, emax = -1
Floating-Point Numbers

IEEE 754–2008 modelled by Flocq (Boldo et al. 2009)

$$\pm m \cdot 2^e \mid 0 \leq m < 2^3 = 8; -3 \leq e \leq -1$$

Example: prec = 3, emin = -3, emax = -1
Floating-Point Numbers

IEEE 754–2008 modelled by Flocq (Boldo et al. 2009)

\[ \pm m \cdot 2^e \mid 0 \leq m < 2^3 = 8; -3 \leq e \leq -1 \]

Example: prec = 3, emin = -3,emax = -1
Floating-Point Numbers

IEEE 754–2008 modelled by Flocq (Boldo et al. 2009)

\[ \pm m \cdot 2^e \mid 0 \leq m < 2^3 = 8; -3 \leq e \leq -1 \]

Example: prec = 3, emin = -3, emax = -1
Floating-Point Numbers

IEEE 754-2008 modelled by Flocq (Boldo et al. 2009)

\[ \pm m \cdot 2^e \mid 0 \leq m < 2^3 = 8; -3 \leq e \leq -1 \]

Example: prec = 3, emin = -3, emax = -1

Denormal numbers
- \( e = e_{\text{min}} \)
- \( m < 2^{\text{prec}-1} \)

Normal numbers
- \( 2^{\text{prec}-1} \leq m \)
Floating-Point Numbers

IEEE 754–2008 modelled by Flocq (Boldo et al. 2009)

\[ \pm m \cdot 2^e \quad 0 \leq m < 2^{\text{prec}}, \ e_{\text{min}} \leq e \leq e_{\text{max}} \]

Denormal numbers
\[ e = e_{\text{min}} \]
\[ m < 2^{\text{prec}-1} \]

Normal numbers
\[ 2^{\text{prec}-1} \leq m \]
Floating-Point Numbers and Rounding Errors

IEEE 754–2008 modelled by Flocq (Boldo et al. 2009)

- A binary operation $a \, T \, b$ is not computed exactly
- Rounded from its ideal value
  - Rounding mode: rounding to nearest, ties to even mantissa

Denormal:
$$(a \, T \, b) + c$$
$|c| \leq 2^{emin-1}$

Normal:
$$(a \, T \, b) \, (1 + d)$$
$|d| \leq 2^{-prec}$

General case: $$(a \, T \, b) \, (1 + d) + c$$
with $c \times d = 0$$
Optimized Rounding Errors

• Normal numbers: \((a + b)(1 + d)\), if \(|a + b|\) large enough and no overflow
• Denormal numbers: \((a + b) + e\), if \(|a + b|\) small enough
• Sterbenz subtraction: \((a - b)\) if \(a/2 \leq b \leq 2a\)
• Multiply by power of 2 is always exact (unless overflow)
• Divide by power of 2 is exact if no gradual underflow
Flocq: correctness of floating-point arithmetic

Theorem Bplus_correct :
forall plus_nan m x y,
is_finite x = true ->
is_finite y = true ->
if Rlt_bool (Rabs (round radix2 fexp (round_mode m) (B2R x + B2R y))) (bpow radix2 emax) then
  B2R (Bplus plus_nan m x y) = round radix2 fexp (round_mode m) (B2R x + B2R y) /
  is_finite (Bplus plus_nan m x y) = true /
  Bsign (Bplus plus_nan m x y) =
    match Rcompare (B2R x + B2R y) 0 with
    | Eq => match m with mode_DN => orb (Bsign x) (Bsign y)
    | _ => andb (Bsign x) (Bsign y) end
    | Lt => true
    | Gt => false
  end
else
  (B2FF (Bplus plus_nan m x y) = binary_overflow m (Bsign x) /
    Bsign x = Bsign y).

Theorem relative_error_ex :
forall x,
(bpow emin <= Rabs x)%R ->
exists eps,
  (Rabs eps < bpow (-p + 1))%R /
  round beta fexp rnd x = (x * (1 + eps))%R.

No overflow

Normal numbers
Flocq: correctness of floating-point arithmetic

Theorem Bplus_correct :
forall plus_nan m x y,
is_finite x = true ->
is_finite y = true ->
if Rlt_bool (Rabs (round radix2 fexp (round_mode m) (B2R x + B2R y))) (bpow radix2 emax) then
  B2R (Bplus plus_nan m x y) = round radix2 fexp (round_mode m) (B2R x + B2R y) /
  is_finite (Bplus plus_nan m x y) = true /
  Bsign (Bplus plus_nan m x y) = match Rcompare (B2R x + B2R y) 0 with
  | Eq => match m with mode_DN => orb (Bsign x) (Bsign y)
  | _ => andb (Bsign x) (Bsign y) end
  | Lt => true
  | Gt => false
end
else
  (B2FF (Bplus plus_nan m x y) = binary_overflow m (Bsign x) /
    Bsign x = Bsign y).

Theorem relative_error_ex :
forall x,
(bpow emin <= Rabs x)%R ->
exists eps,
(Rabs eps < bpow (-p + 1))%R /
round beta fexp rnd x = (x * (1 + eps))%R.

No overflow

Better tackle them automatically

Normal numbers
Rounding Error Terms

Optimized cases

- Normal numbers: \((a \pm b)(1 + d)\), if \(|a \pm b|\) large enough and no overflow
- Denormal numbers: \((a \pm b) + e\), if \(|a \pm b|\) small enough
- Sterbenz subtraction: \((a - b)\) if \(a/2 \leq b \leq 2a\)
- Multiply by power of 2 is always exact (unless overflow)
- Divide by power of 2 is exact if no gradual underflow

Our VCFloat approach:

- Automatically generate validity conditions
- Automatically check them on the fly
- Add annotations for optimized rounding
Verification of Rounding Error Terms

Use Coq-Interval (Melquiond 2015) to automatically check validity conditions

- Automatic certified interval arithmetic
- Reduce correlation issues:
  - Bisection (branch-and-bound)
  - Automatic differentiation
  - Taylor models
- Used for all rounding errors
- All computations within Coq: consumes most proof checking time and memory in overall proof
- Stress test
Our Verification Framework: VCFloat

C code to be verified

CompCert Clightgen

Clight AST → Floating-point Core Calculus

Floating-point Core Calculus

Automatic annotations for rounding optimizations

Correctness Proofs

Real-number Expressions

Correctness Proofs

http://github.com/reservoirlabs/vcfloat
Released under GNU GPL v3
Verification of Rounding Error Terms

\[ 2.0f \times (\text{float}) x - 3.0; \]

C floating-point expression
Verification of Rounding Error Terms

\[ 2.0f \times (\text{float}) x - 3.0; \]

\[ \left( 2^{24,128} \otimes \left[ x \right]^{24,128} \right) \oplus 3^{53,1024} \]
Verification of Rounding Error Terms

2.0f * (float) x - 3.0;

Assume x in [1, 2]

\(2^{(24,128)} \otimes \left[x\right]^{(24,128)} \oplus 3^{(53,1024)}\)

C floating-point expression

Core floating-point expression
Verification of Rounding Error Terms

\[ 2.0f \times (\text{float}) x - 3.0; \]

\[ \left( 2(24,128) \otimes [x](24,128) \right) \oplus 3(53,1024) \]

Assume \( x \) in \([1, 2]\)

\[ \left( 2(24,128) \otimes [x]^{\text{Norm}}(24,128) \right) \oplus 3(53,1024) \]

Because \( 2^{-125} \leq |x| < 2^{128} \)

C floating-point expression

Core floating-point expression

Annotated floating-point expression
Verification of Rounding Error Terms

\[ 2.0f \times (\text{float}) x - 3.0; \]

\[ (2^{24,128} \times [x]^{24,128}) \oplus 3^{53,1024} \]

Assume \( x \) in \([1, 2]\)

\[ (2^1 \times [x]^{\text{Norm}}^{24,128}) \oplus 3^{53,1024} \]

Because \(2^{-125} \leq |x| < 2^{128}\)

And for all \( d \) in \([-2^{-24}, 2^{-24}]\), \(|2 \times (x \times (1 + d))| < 2^{128}\)
Verification of Rounding Error Terms

2.0f * (float) x - 3.0;

C floating-point expression

\[ (2^{(24,128)} \otimes [x]^{(24,128)}) \oplus 3^{(53,1024)} \]

Core floating-point expression

Assume x in [1, 2]

\[ (2^1 \otimes [x]^{Norm}\text{ Sterbenz}) \oplus 3^{(53,1024)} \]

Annotated floating-point expression

Because \(2^{-125} \leq |x| < 2^{128}\)
And for all d in \([-2^{-24}, 2^{-24}]\), \(|2 * (x * (1 + d))| < 2^{128}\)
And for all d in \([-2^{-24}, 2^{-24}]\), \(3/2 \leq 2 * (x * (1 + d)) \leq 3*2\)
Verification of Rounding Error Terms

\[ 2.0f \times (\text{float}) \, x - 3.0; \]

Annotated floating-point expression

\[ (2^{24,128}) \otimes [x]^{24,128} \oplus 3^{53,1024} \]

Core floating-point expression

Assume \( x \) in \([1, 2]\)

\[ (2^1) \otimes [x]^{24,128} \otimes 3^{53,1024} \]

Annotated floating-point expression

Because \( 2^{-125} \leq |x| < 2^{128} \)
And for all \( d \) in \([-2^{-24}, 2^{-24}]\), \(|2 \times (x \times (1 + d))| < 2^{128} \)
And for all \( d \) in \([-2^{-24}, 2^{-24}]\), \(3/2 \leq 2 \times (x \times (1 + d)) \leq 3 \times 2\)

\[ 2 \times (x \times (1 + \delta)) - 3 \]

Real-number expression with error terms

For some \( \delta \) in \([-2^{-24}, 2^{-24}]\)
Formal Verification of Error Bounds

DEMO
CONCLUSIONS

Formal Verification of Error Bounds
Conclusion and Future Work

C programs with floating-point computations can now be fully verified within Coq with a TCB smaller than ever:

- Implementation of Coq, faithfulness of Clight and Flocq

Energy-efficient approximate implementations can be allowed in critical applications, once their error bounds are certified correct

Floating-point steps are now automated, but overall more practical improvements are still desirable:

- Integer handling
- C control flow: Integrate into Verifiable C
Thank you!

Coq library and proofs available online:

- [http://github.com/reservoirlabs/vcfloat](http://github.com/reservoirlabs/vcfloat)
- Mostly released under GNU GPL v3
- Currently based on Coq 8.5beta2, will be adapted to 8.5rc1
- Stress test for Flocq, Coq-Interval, and computations within Coq

For more information:

- ramananandro@reservoir.com