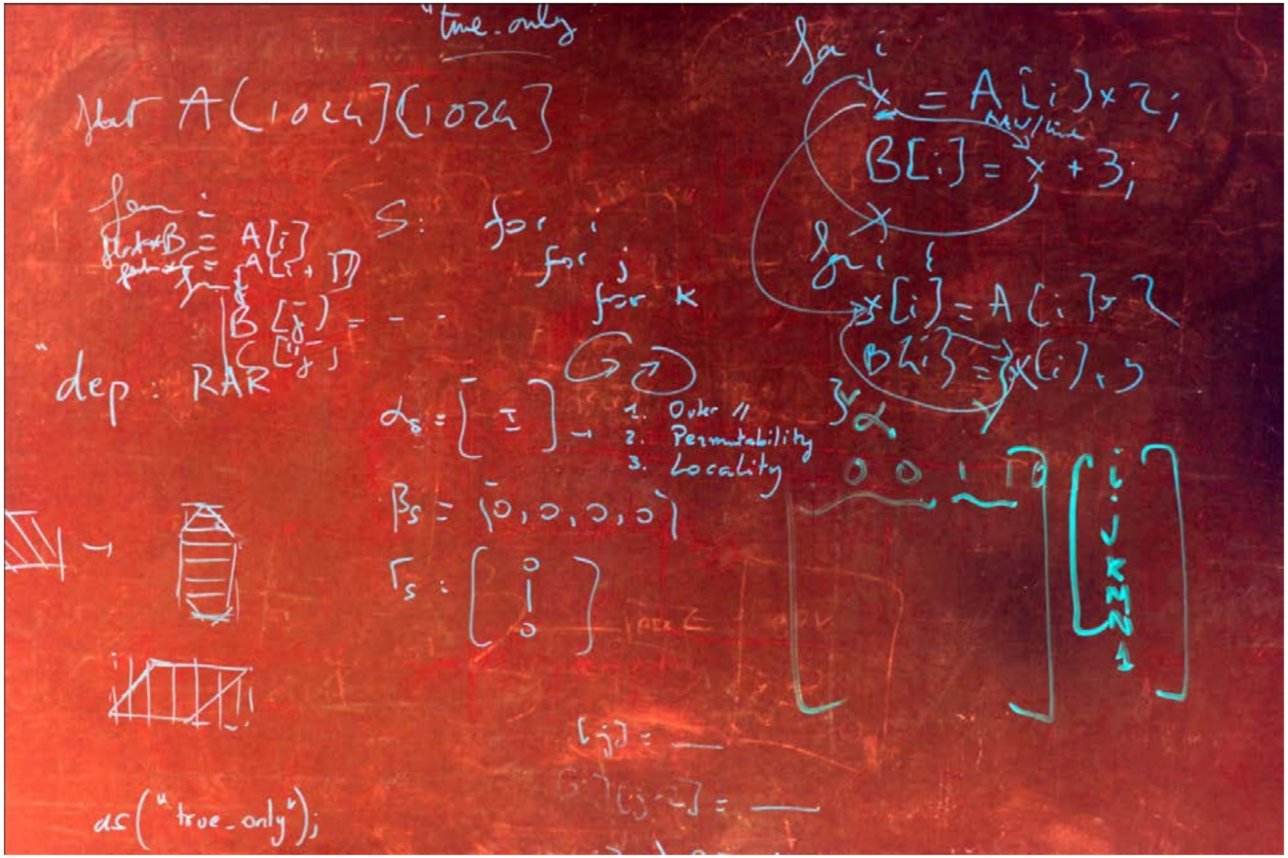
A Unified Coq Framework for Verifying C Programs with Floating-Point Computations

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

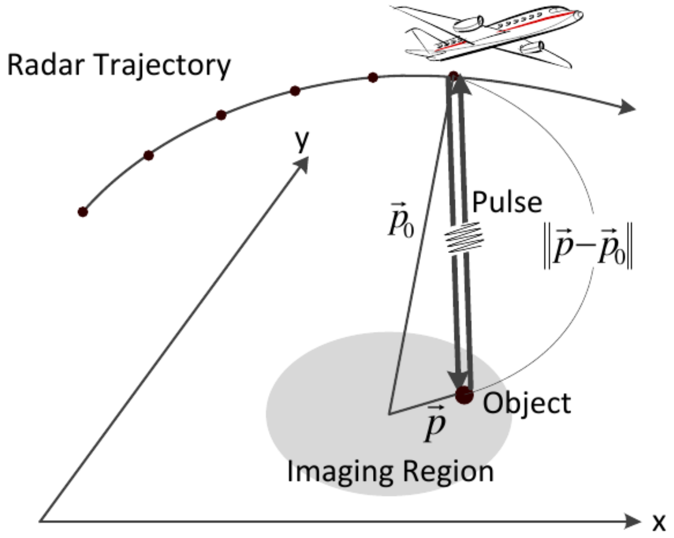
This Presentation

- 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

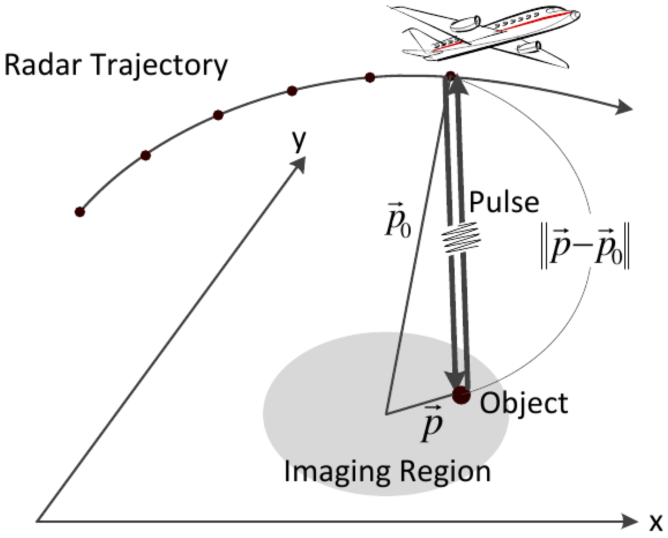


```
for all pixels $x$ do
  for all pulses $p$ do
    $R$ = distance between platform and pixel $x$ for pulse $p$
    $s$ = sample interpolated from pulse $p$ in neighborhood of range $R$
    Apply phase correction to sample $s$ based on range $R$
    Accumulate sample $s$ into pixel $x$
  end for
end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

Real-number algorithm

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

SAR Backprojection

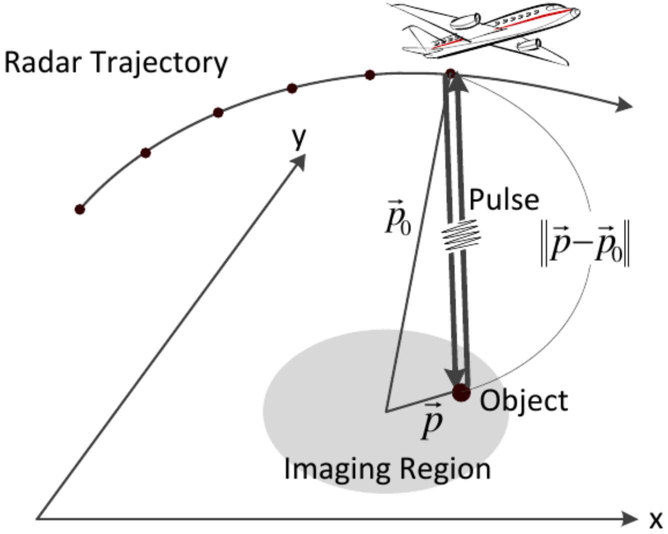


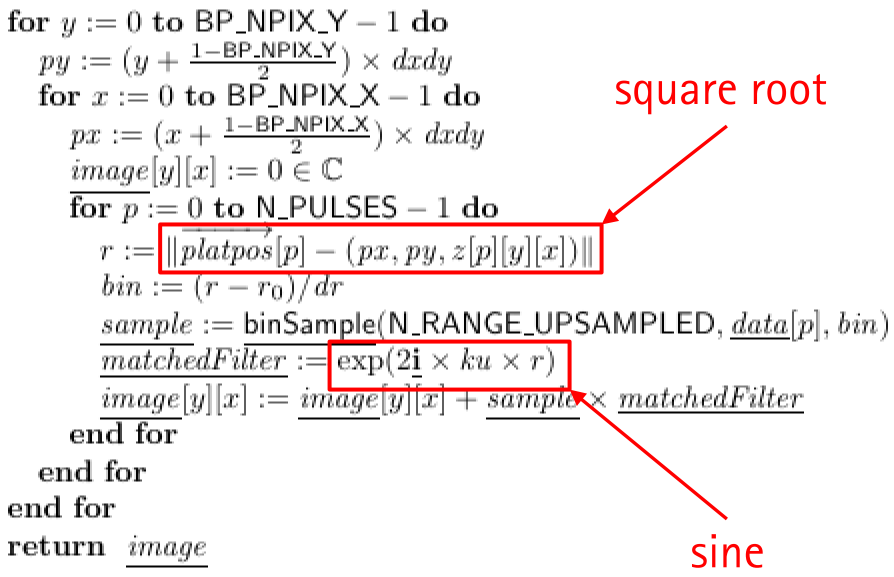
```
for $y := 0$ to BP_NPIX_Y $- 1$ do
  $py := (y + \frac{1-\text{BP\_NPIX\_Y}}{2}) \times dxdy$
  for $x := 0$ to BP_NPIX_X $- 1$ do
    $px := (x + \frac{1-\text{BP\_NPIX\_X}}{2}) \times dxdy$
    $\underline{image}[y][x] := 0 \in \mathbb{C}$
    for $p := 0$ to N_PULSES $- 1$ do
      $r := \|\overrightarrow{platpos}[p] - (px, py, z[p][y][x])\|$
      $bin := (r - r_0)/dr$
      $\underline{sample} := \underline{\text{binSample}}(\text{N\_RANGE\_UPSAMPLED}, \underline{data}[p], bin)$
      $\underline{matchedFilter} := \exp(2\underline{\mathbf{i}} \times ku \times r)$
      $\underline{image}[y][x] := \underline{image}[y][x] + \underline{sample} \times \underline{matchedFilter}$
    end for
  end for
end for
return $\underline{image}$
```

Real-number algorithm

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

SAR Backprojection





```
for $y := 0$ to BP_NPIX_Y $- 1$ do
  $py := (y + \frac{1-\text{BP\_NPIX\_Y}}{2}) \times dxdy$
  for $x := 0$ to BP_NPIX_X $- 1$ do
    $px := (x + \frac{1-\text{BP\_NPIX\_X}}{2}) \times dxdy$
    $\underline{image}[y][x] := 0 \in \mathbb{C}$
    for $p := 0$ to N_PULSES $- 1$ do
      $r := \boxed{\|\overrightarrow{platpos}[p] - (px, py, z[p][y][x])\|}$
      $bin := (r - r_0)/dr$
      $\underline{sample} :=$ binSample(N_RANGE_UPSAMPLED, $\underline{data}[p], bin$)
      $\underline{matchedFilter} := \boxed{\exp(2\underline{\mathbf{i}} \times ku \times r)}$
      $\underline{image}[y][x] := \underline{image}[y][x] + \underline{sample} \times \underline{matchedFilter}$
    end for
  end for
end for
return $\underline{image}$
```

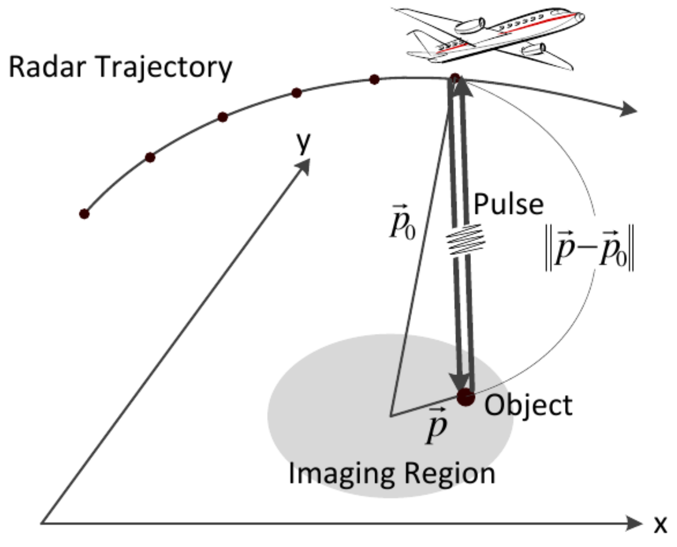
square root

sine

Real-number algorithm

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

SAR Backprojection



```
for $y := 0$ to BP_NPIX_Y $- 1$ do
    $py := (y + \frac{1-\text{BP\_NPIX\_Y}}{2}) \times dxdy$
    for $x := 0$ to BP_NPIX_X $- 1$ do
        $px := (x + \frac{1-\text{BP\_NPIX\_X}}{2}) \times dxdy$
        $image[y][x] := 0 \in \mathbb{C}$
        for $p := 0$ to N_PULSES $- 1$ do
            $r := \boxed{\|\overrightarrow{platpos}[p] - (px, py, z[p][y][x])\|}$
            $bin := (r - r_0)/dr$
            $sample :=$ binSample(N_RANGE_UPSAMPLED, $data[p], bin$)
            $matchedFilter := \boxed{\exp(2\mathbf{i} \times ku \times r)}$
            $image[y][x] := image[y][x] + sample \times matchedFilter$
        end for
    end for
end for
return $image$
```

square root

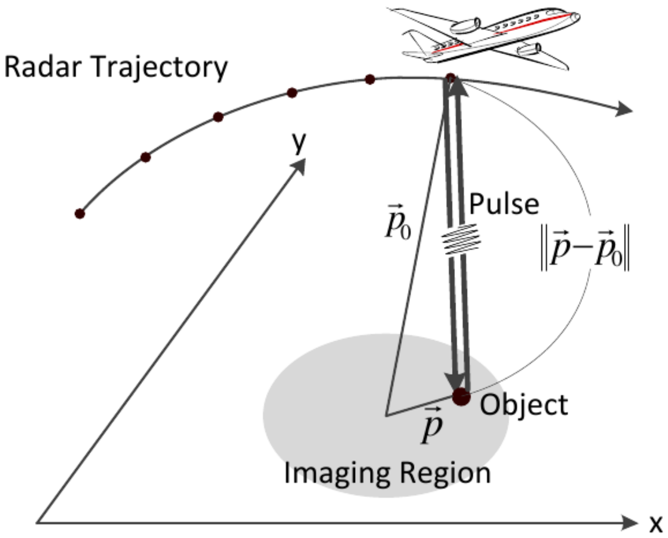
sine

floating-point computations

Implementation

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

SAR Backprojection



square root

```
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(…) … ;
    …
   }
  }
 }
}
```

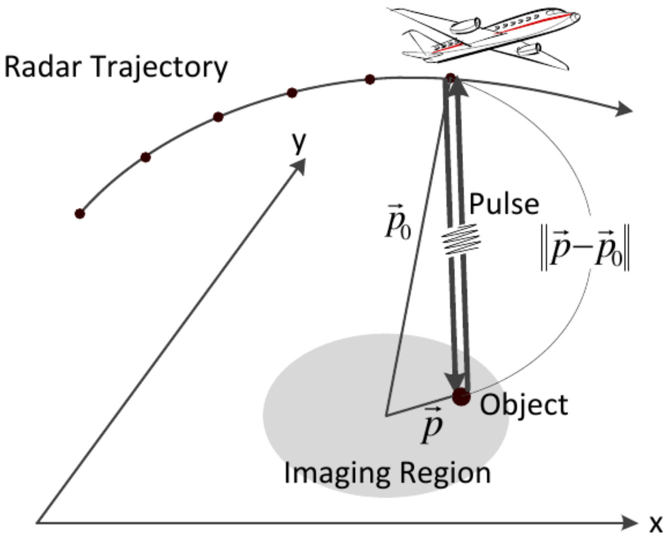
sine

floating-point computations

C Implementation

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

SAR Backprojection



```
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(…) … ;
    …
   }
  }
 }
}
```

approximate square root

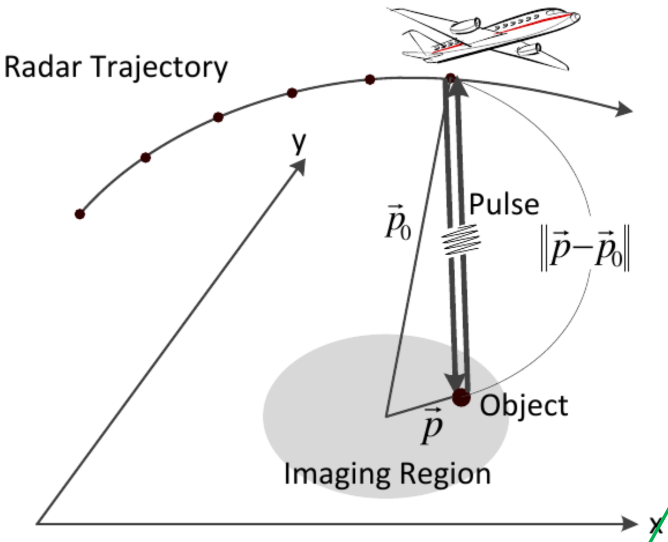
approximate sine

floating-point computations

C Implementation

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

SAR Backprojection



```
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(…) … ;
    …
   }
  }
 }
}
```

approximate square root

approximate sine

Precision tuning

floating-point computations

C Implementation
(~ 150 lines)

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:

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

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)

Hypotheses

```
∀ 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)),
```

Conclusions

```
∃ 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.
```

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)),
```

Hypotheses on input data

Memory contents and permissions

```
∃ 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.
```

Conclusions

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)),
```

Hypotheses

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

C code runs

```
∃ 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)) ∧
```

Memory contents

```
       ∀ 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 ∧
```

FP does not overflow

```
         let (tr, ti) := SARBackProj.sar_backprojection y x    in
         Rabs (B2R _ _ ir - tr) ≤ pixel_bound ∧
         Rabs (B2R _ _ ii - ti) ≤ pixel_bound.
```

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

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 π 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

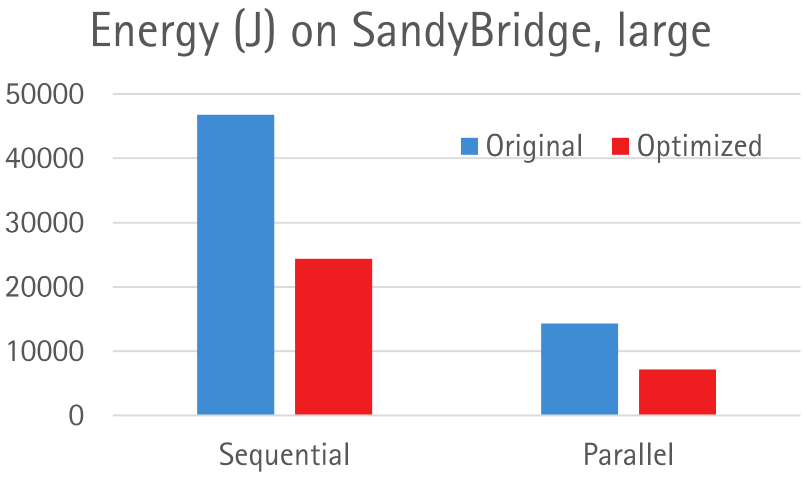
Precision Results

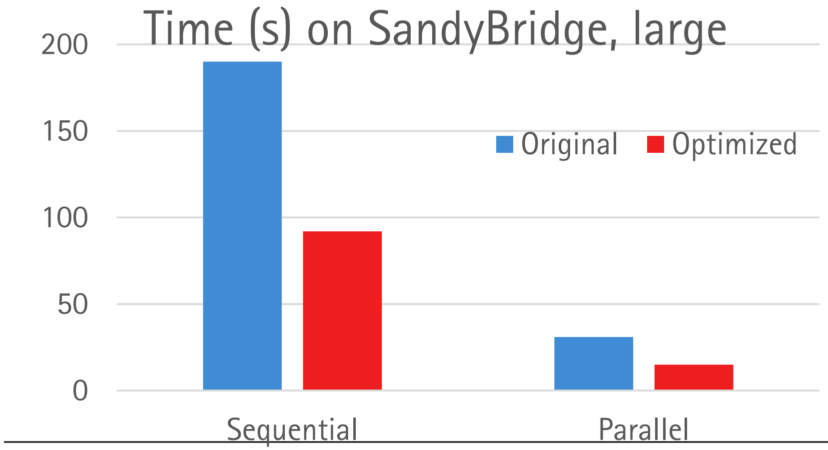
- Input data bounds from DARPA PERFECT suite
- Error grows with image size
- No statistical reasoning about data

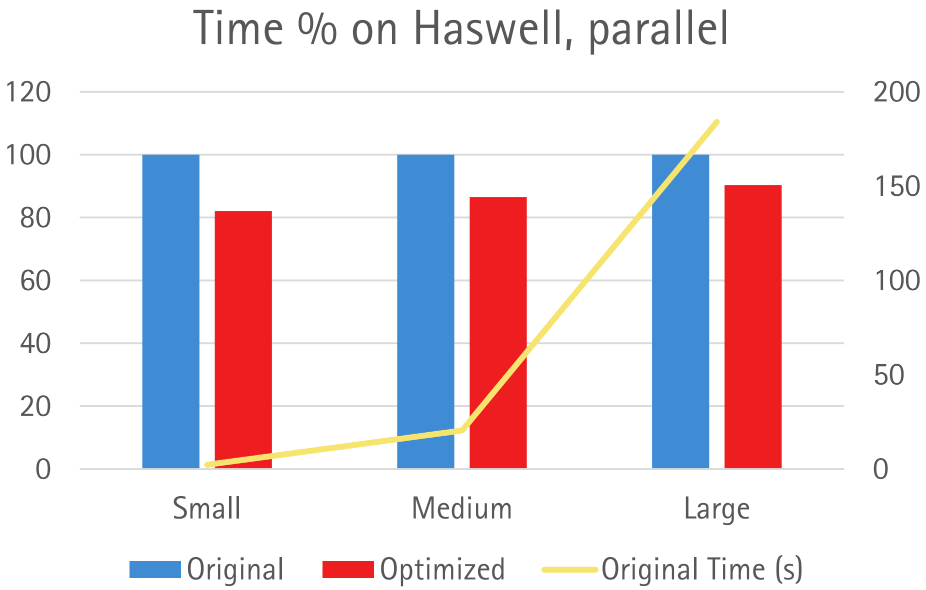
Norm	Interpol.	Sine	Final sum	Small dB	Med. dB	Large dB
Double	Double	Double	Doub/Sing	-2	4	10
Double	Single	Doub/Sing	Doub/Sing	3	15	27
Adaptive	Double	Double	Doub/Sing	12	24	37
Adaptive	Single	Doub/Sing	Doub/Sing	14	26	39
Double	Single	Approx.	Doub/Sing	17	29	40
Adaptive	**Single**	**Approx.**	**Doub/Sing**	**20**	**32**	**44**
Double	Single	Doub/Sing	Single	34	52	70
Adaptive	Single	Approx.	Single	35	53	70

Performance Measurements for Optimized C Code

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


Energy (J) on SandyBridge, large


Time (s) on SandyBridge, large


Time % on Haswell, parallel

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)),
```

Hypotheses

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

C code runs

```
∃ 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)) ∧
```

Memory contents

```
          ∀ 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 ∧
```

FP does not overflow

```
             let (tr, ti) := SARBackProj.sar_backprojection y x    in
             Rabs (B2R _ _ ir - tr) ≤ pixel_bound ∧
             Rabs (B2R _ _ ii - ti) ≤ pixel_bound.
```

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

Final Correctness Statement (slightly simplified)

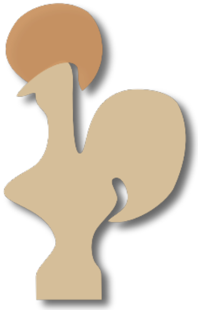
```
∀ P `(HYPS: SARHypotheses P)  m
     (Hm: holds m (P ++
            Pperm_int bir oir (BP_NPIX_X × BP_NPIX_Y) ++
            Pperm_int bii oii (BP_NPIX_X × BP_NPIX_Y)),
∃ m' , star Clight.step2
         (Callstate fn_sar_backprojection ...) (Returnstate Vundef Kstop m') ∧
∃ image_r image_i,
          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

CompCert Clight

Memory contents



FP does not overflow

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

Flocq

Our Design Choices: Which Formal Methods?

Verification using the Coq proof assistant

Correctness fully embedded in Coq using existing libraries:

- CompCert Clight (Blazy & Leroy, J. Autom. Reason. 2009)
  - 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

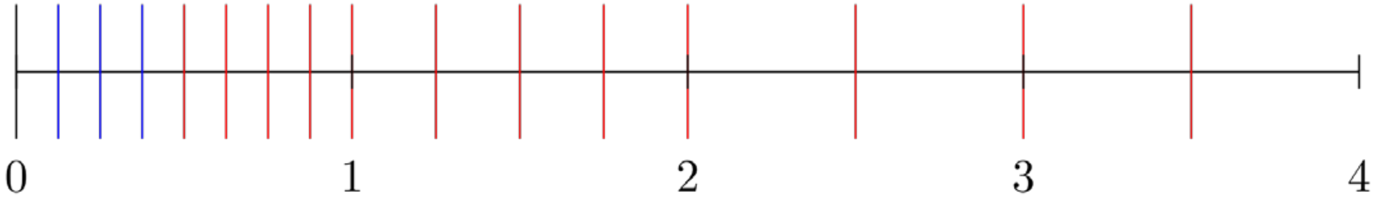
?

Real-number semantics

Floating-Point Numbers

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

- A binary operation a T 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)

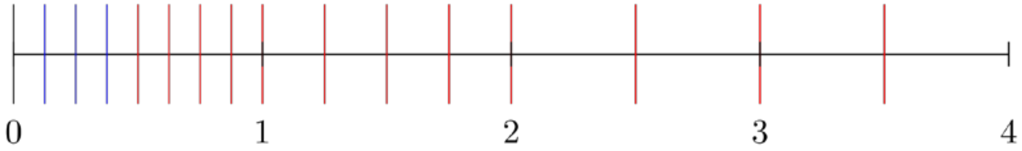
$$\pm m \cdot 2^{e} \qquad 0 \leq m < 2^{prec}, \;\; e_{min} \leq e \leq e_{max}$$

Floating-Point Numbers

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

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

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

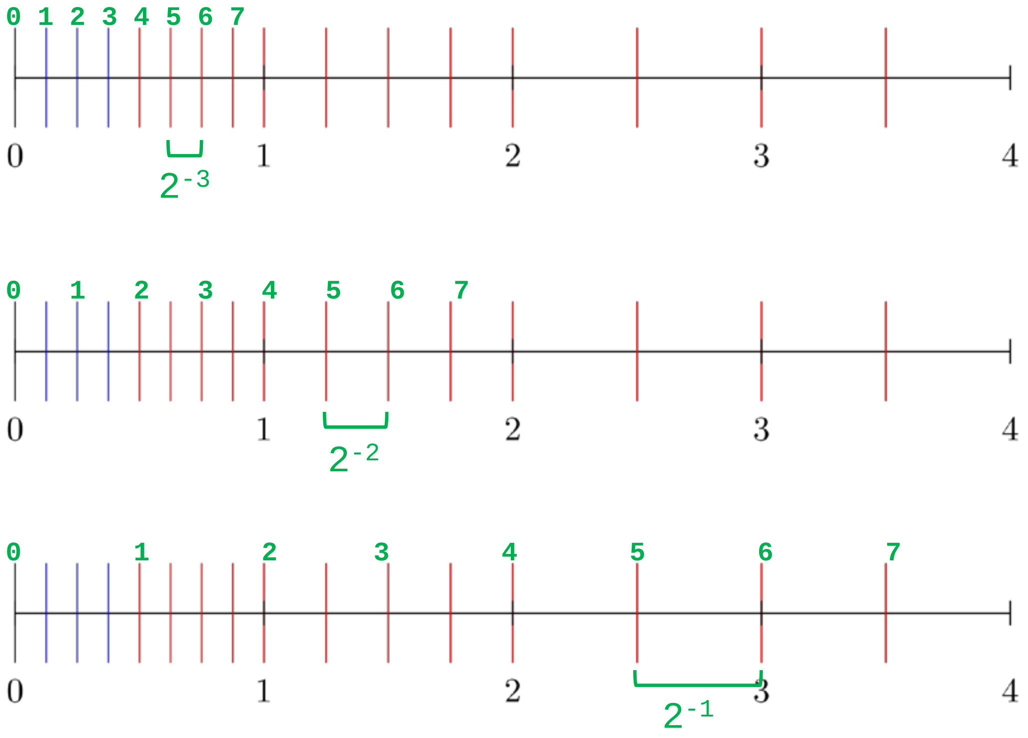


Floating-Point Numbers

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

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

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

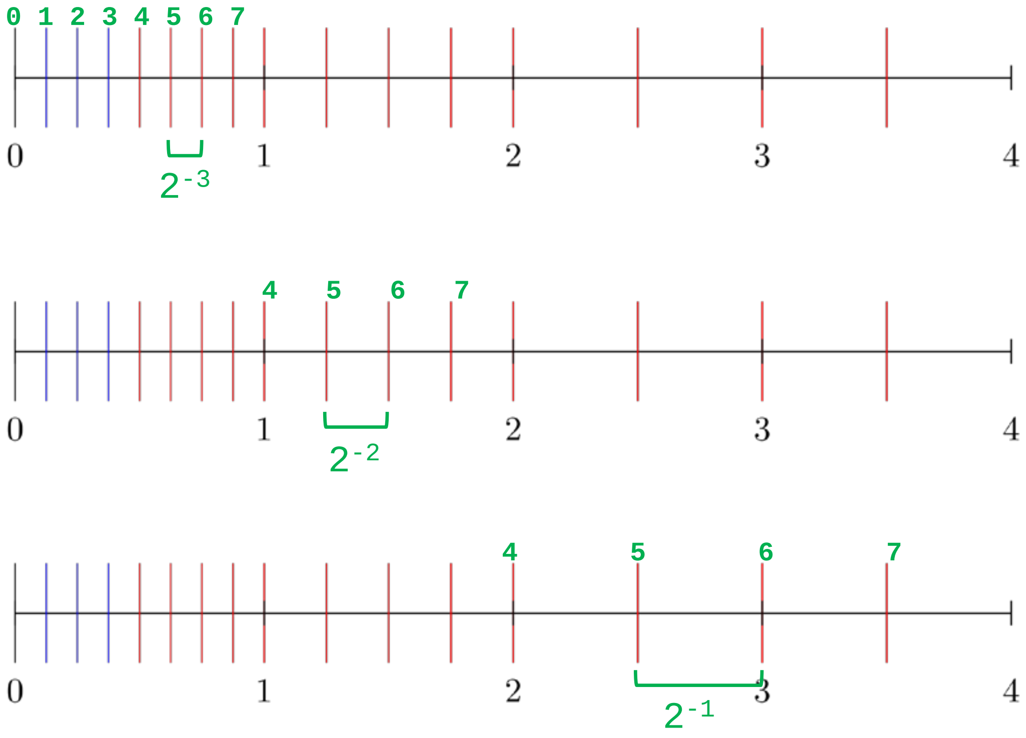


Floating-Point Numbers

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

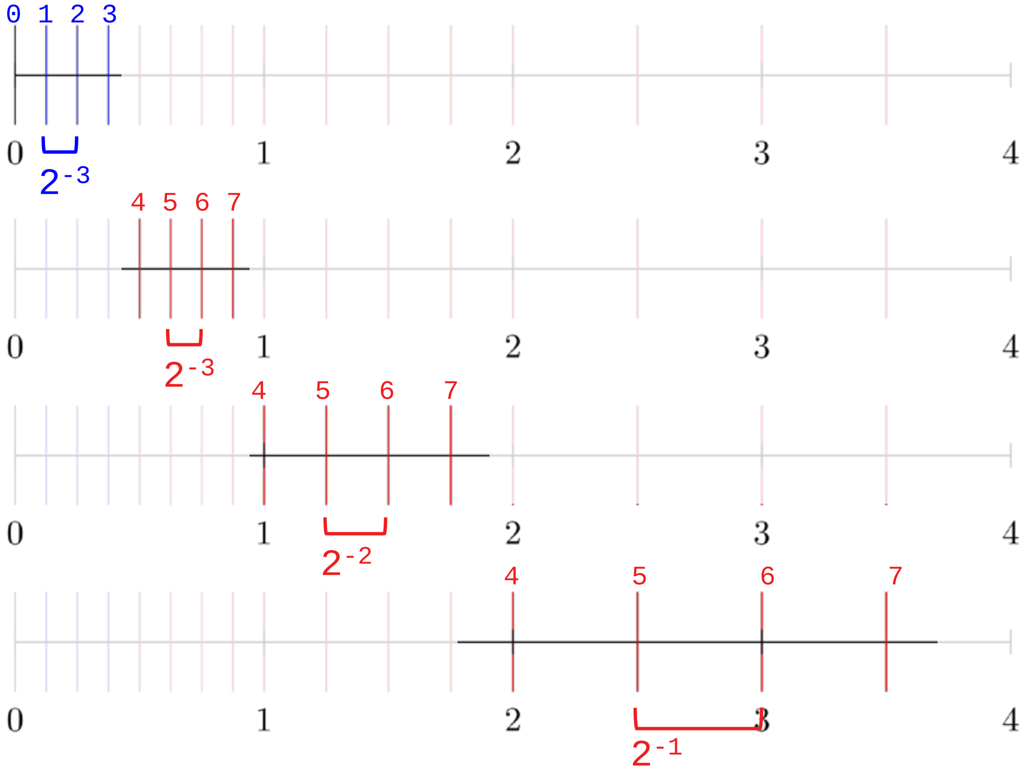


Floating-Point Numbers

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

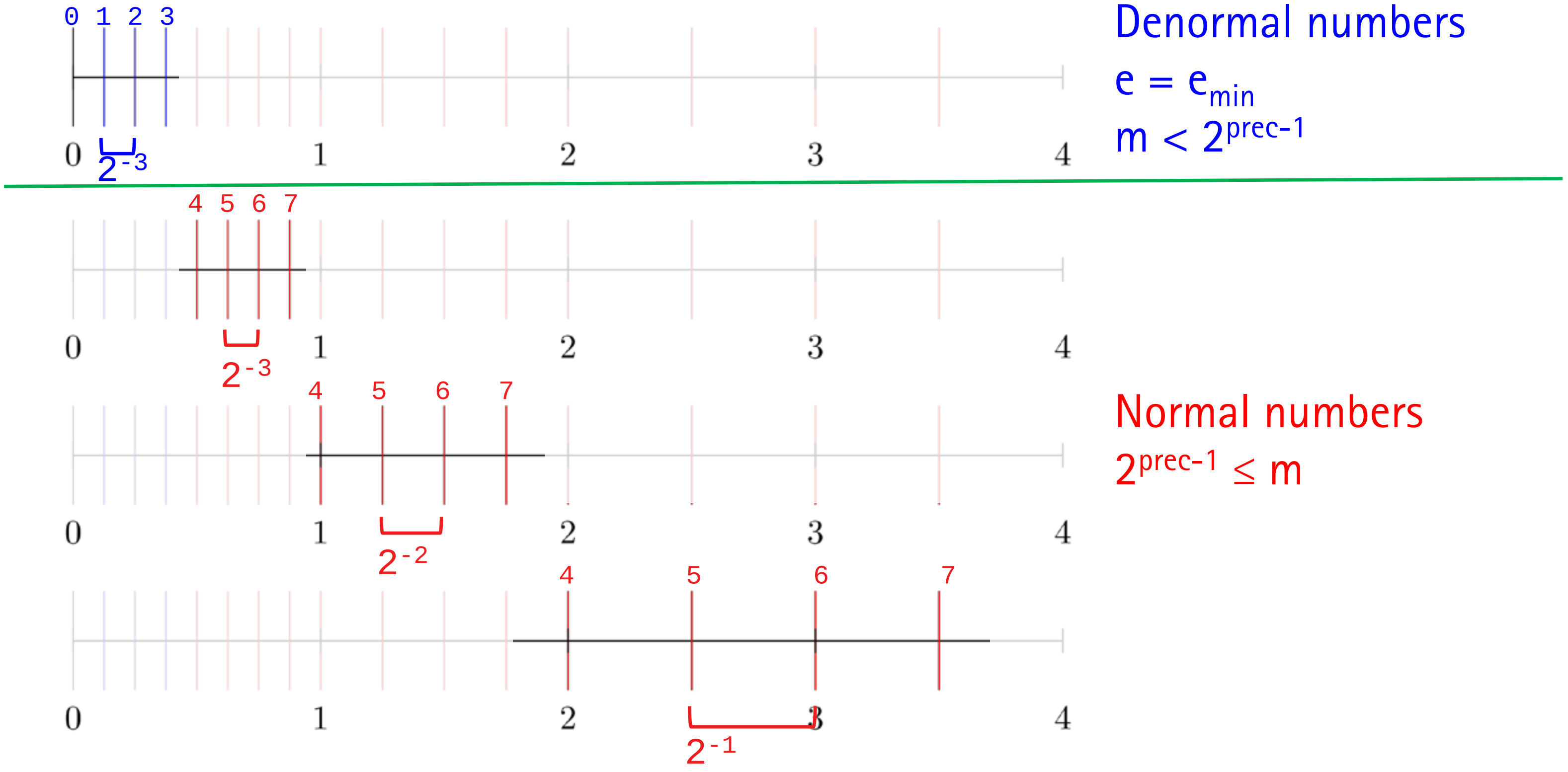


Floating-Point Numbers

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

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

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



Denormal numbers

$e = e_{min}$

$m < 2^{prec-1}$

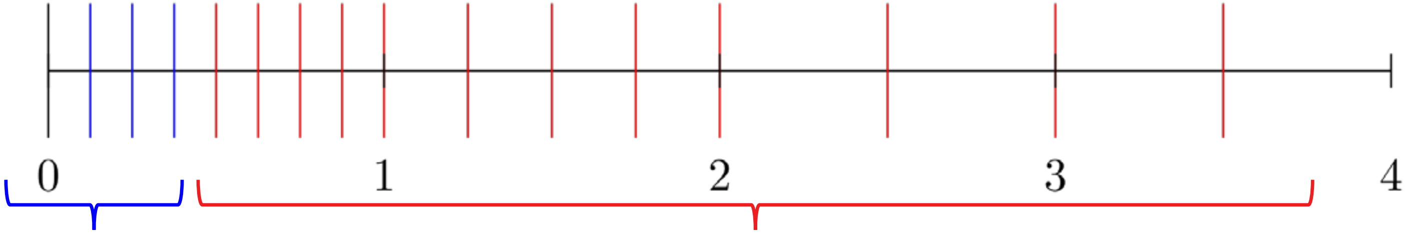
Normal numbers

$2^{prec-1} \leq m$

Floating-Point Numbers

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

$\pm m \cdot 2^{e}$ $\quad\quad 0 \leq m < 2^{prec}, \; e_{min} \leq e \leq e_{max}$



Denormal numbers

$e = e_{min}$

$m < 2^{prec-1}$

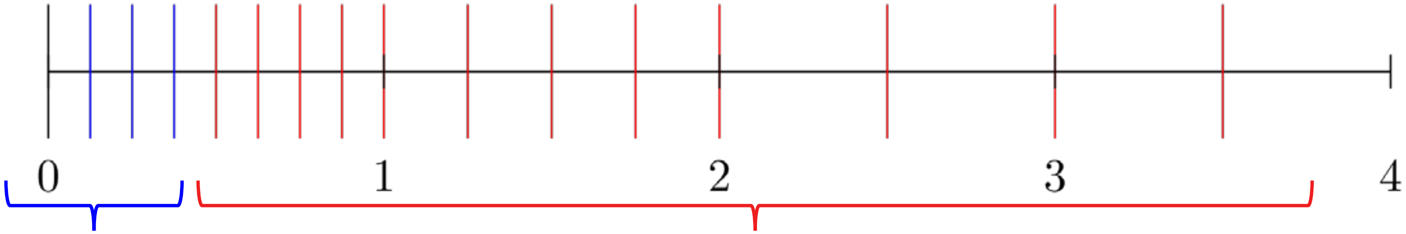
Normal numbers

$2^{prec-1} \leq m$

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*d = 0

Optimized Rounding Errors

- Normal numbers: (a T b) (1 + d), if |a T b| large enough and no overflow
- Denormal numbers: (a T b) + e, if |a T b| small enough
- Sterbenz subtraction: (a – b) if a/2 <= b <= 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).
```

No overflow

```
  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.
```

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_fini
    Bsign
      matc
        | Eq

        | Lt
        | Gt
      end
  else
    (B2FF
```

No overflow

Better tackle them automatically

```
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.
```

Normal numbers

Rounding Error Terms

Optimized cases

- Normal numbers: (a T b) (1 + d), if |a T b| large enough and no overflow
- Denormal numbers: (a T b) + e, if |a T b| small enough
- Sterbenz subtraction: (a – b) if a/2 <= b <= 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:

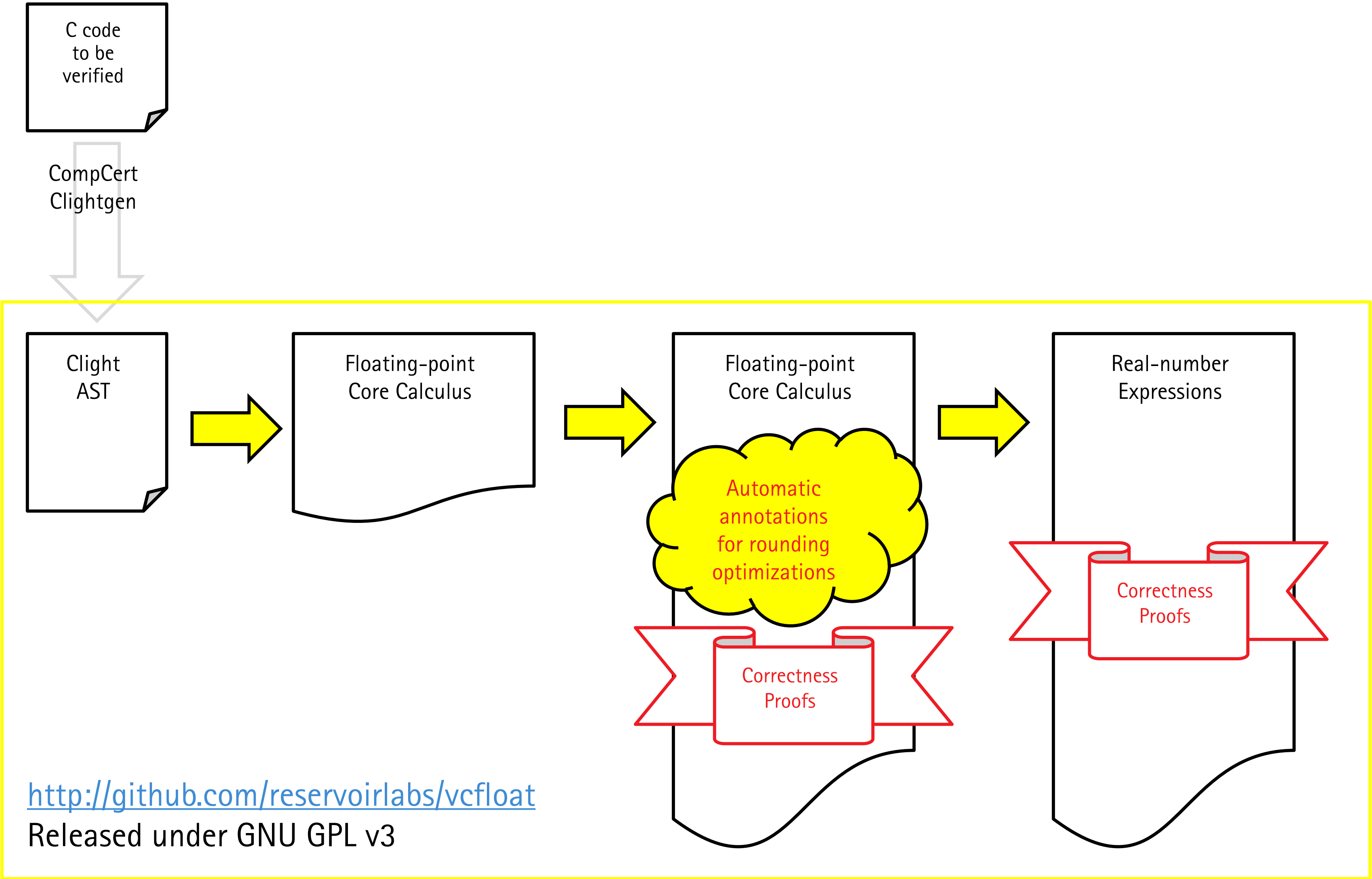
- Automatically generate validity conditions
- Automatically check them on the fly
- Add annotations for optimized rounding

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



http://github.com/reservoirlabs/vcfloat

Released under GNU GPL v3

Verification of Rounding Error Terms

```
2.0f * (float) x - 3.0;
```

C floating-point expression

Verification of Rounding Error Terms

`2.0f * (float) x - 3.0;`

C floating-point expression

$$(2_{(24,128)} \otimes [x]_{(24,128)}) \ominus 3_{(53,1024)}$$

Core floating-point expression

Verification of Rounding Error Terms

```
2.0f * (float) x - 3.0;
```

C floating-point expression

$$(2_{(24,128)} \otimes [x]_{(24,128)}) \ominus 3_{(53,1024)}$$

Core floating-point expression

Assume x in [1, 2]

Verification of Rounding Error Terms

```
2.0f * (float) x - 3.0;
```

C floating-point expression

$$(2_{(24,128)} \otimes [x]_{(24,128)}) \ominus 3_{(53,1024)}$$

Core floating-point expression

Assume x in [1, 2]

$$(2_{(24,128)} \otimes [x]^{\text{Norm}}_{(24,128)}) \ominus 3_{(53,1024)}$$

Annotated floating-point expression

Because 2ˆ-125 ≤ |x| < 2ˆ128

Verification of Rounding Error Terms

```
2.0f * (float) x - 3.0;
```

C floating-point expression

$$\left(2_{(24,128)} \otimes [x]_{(24,128)}\right) \ominus 3_{(53,1024)}$$

Core floating-point expression

Assume x in [1, 2]

$$\left(2^1 \cancel{\otimes} [x]^{\text{Norm}}_{(24,128)}\right) \ominus 3_{(53,1024)}$$

Because 2^-125 ≤ |x| < 2^128

And for all d in [-2^-24, 2^-24], |2 * (x * (1 + d))| < 2^128

Annotated floating-point expression

Verification of Rounding Error Terms

```
2.0f * (float) x - 3.0;
```

C floating-point expression

$$(2_{(24,128)} \otimes [x]_{(24,128)}) \ominus 3_{(53,1024)}$$

Core floating-point expression

Assume x in [1, 2]

$$\left(2^1 \cancel{\otimes} [x]^{\text{Norm}}_{(24,128)}\right) \overset{\text{Sterbenz}}{-} 3_{(53,1024)}$$

Annotated floating-point expression

Because 2^-125 ≤ |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 ≤ 2 * (x * (1 + d)) ≤ 3*2

Verification of Rounding Error Terms

```
2.0f * (float) x - 3.0;
```

C floating-point expression

$$\left(2_{(24,128)} \otimes [x]_{(24,128)}\right) \ominus 3_{(53,1024)}$$

Core floating-point expression

Assume x in [1, 2]

$$\Big(\;2^1 \;\; \xcancel{\otimes}\; [x]^{\text{Norm}}_{(24,128)}\Big) \overset{\text{Sterbenz}}{\cancel{\ominus}} 3_{(53,1024)}$$

Annotated floating-point expression

Because 2^-125 ≤ |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 ≤ 2 * (x * (1 + d)) ≤ 3*2

$$2 \times (x \times (1 + \delta)) - 3$$

For some δ in $[-2^{-24}, 2^{-24}]$

Real-number expression with error terms

Formal Verification of Error Bounds

DEMO

Formal Verification of Error Bounds

CONCLUSIONS

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
- 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