Hardware-independent proofs of numerical programs [⋆]

Sylvie Boldo[1,2] and Thi Minh Tuyen Nguyen[1,2]

[1] INRIA Saclay – Île-de-France, F-91893 Orsay cedex, France
[2] LRI, Univ. Paris-Sud, F-91405 Orsay cedex, France

Abstract. On recent architectures, a numerical program may give different answers depending on the execution hardware and the compilation. Our goal is to formally prove properties about numerical programs that are true for multiple architectures and compilers. We propose an approach that states the rounding error of each floating-point computation whatever the environment. This approach is implemented in the Frama-C platform for static analysis of C code. A case study from avionics is given and is entirely and automatically proved.

1 Introduction

Floating-point computations often appear in current critical systems from domains such as physics, aerospace system, energy, etc. For such systems, hardwares and softwares play an important role.

All current microprocessor architectures support IEEE-754 floating-point arithmetic [1]. However, there exists some architecture-dependent issues. For example, the x87 floating-point unit uses the 80-bit internal floating-point registers on the Intel platform. The fused multiply-add (FMA) instruction, supported by the PowerPC and the Intel Itanium architectures, computes $xy \pm z$ with a single rounding. These issues can introduce subtle inconsistencies between program executions. This means that the floating-point computations of a program running on different architectures may be different [2].

Static analysis is an approach for checking a program without running it. Deductive verification techniques which perform static analysis of code, rely on the ability of theorem provers to check validity of formulas in first-order logic or even more expressive logics. They usually come with expressive specification languages such as JML [3, 4] for Java, ACSL [5] for C, Spec# [6] for C#, etc. to specify the requirements.

Floating-point arithmetic has been formalized firstly by Barret [7]. Since then, many works have been done to formally prove hardware components or algorithms [8–13]. There exists less works on specifying and proving behavioral properties of floating-point programs in deductive verification systems. A work on floating-point in JML for Java is presented in 2006 by Leavens [14]. Another

[⋆] This work was supported by the Hisseo project, funded by Digiteo (`http://hisseo.saclay.inria.fr/`).

proposal has been made in 2007 by Boldo and Filliâtre [15]. Ayad extended this to increase genericity and handle exceptional behaviors [16].

However, these works follow only strict IEEE-754 standard, with neither FMA, nor extended registers. Correctly defining the semantics of the common implementations of floating-point is tricky, because semantics may change according to arguments of compilers and processors. As a result, formal verification of such program is a challenge. The purpose of this paper is to present an approach to prove numerical programs whatever the compiler and the processor. We nevertheless assume that the compiler does preserve the order of operations of the C language. Our approach is implemented in the Frama-C platform[3] associated with Why [17] for static analysis of C code.

This paper is organized as follows. Section 2 presents some basic knowledge needed about floating-point arithmetic, including the x87 unit and the FMA. Section 3 presents a bound on the rounding error of a computation in all possible cases (extended registers, FMA). A case study from avionics is presented in Section 4. This example illustrates our approach and shows the difference of the results between the usual (but maybe incorrect) model and our approach.

2 Floating-point Arithmetic

2.1 The IEEE-754 floating-point standard

The IEEE-754 standard [1] for floating-point arithmetic was developed to define formats and behaviors for floating-point numbers and computations. Five basic formats are defined in this standard: three binary formats, with encodings in length of 32, 64 and 128 bits and two decimal formats, with encodings in length of 64 and 128 bits. We will only consider binary formats here, as they concentrate all the problems. Our ideas could be re-used in decimal formats.

A floating-point number x in a format (p, e_{min}, e_{max}), where e_{min} and e_{max} are the minimal and maximal unbiased exponents and p is the precision, is represented by the triplet (s, m, e) so that

$$x = (-1)^s \times 2^e \times m \tag{1}$$

where

- $s \in \{0, 1\}$ is the sign of x
- e is any integer $e_{min} \leq e \leq e_{max}$
- m $(0 \leq m < 2)$ is the significand of the representation. It has one bit before the radix point and at most $p - 1$ bits after.

We only discuss in this paper the binary format with encoding in 64 bits (usually *double* type in C or Java language), that satisfies (1) with $(53, -1022, 1023)$.

In our definition, some floating-point numbers may have several representations (s, m, e). In the IEEE-754 standard, the representation is unique as it moreover requires assumptions on the significand.

[3] `http://frama-c.cea.fr/`

More precisely, a number with magnitude greater than or equal to $2^{e_{min}}$ is called normal and is required to have a significand greater than or equal to 1. Its significand then has the form: $1.m_1m_2m_3...m_{p-1} \quad (m_i \in \{0,1\})$.

Smaller numbers are called subnormal and are required to have a significand smaller than 1. Its significand has the form: $0.m_1m_2m_3...m_{p-1} \quad (m_i \in \{0,1\})$.

We call *normal range* the set of real numbers which round to a normal floating-point number and *subnormal range* the set of numbers which round to a subnormal number.

When approximating a real number x by its rounding $\circ(x)$, a rounding error happens. We have two types of rounding error: relative error and absolute error. Note that in normal range, the absolute error varies in a wide range, we thus cannot determine an acceptable bound of absolute error. In this case, the relative error is a good choice. The value to bound $\epsilon(x)$ is

$$\epsilon(x) = \left| \frac{x - \circ(x)}{x} \right|. \tag{2}$$

We here consider only round-to-nearest mode, that includes both the default rounding mode (round-to-nearest, ties to even) and the new round-to-nearest, ties away from zero, of the revision of the IEEE-754 standard. In radix 2 and round-to-nearest mode, this relative error is known [18] to be bounded by

$$\epsilon(x) \leq 2^{-p}. \tag{3}$$

In subnormal range, the value of the relative error becomes large (until 0.5). In that case, we prefer a bound based on the absolute error:

$$|x - \circ(x)| \leq 2^{e_{min}-p}. \tag{4}$$

We note that if the arithmetic operation being performed is addition or subtraction, a subnormal result implies an exact result [18], so that $|x - \circ(x)| = 0$. The formula (4) also covers this equation. In order to simplify the cases and to have a unique formula for all basic operations, we ignore this special case and use formula (4) as general formula of rounding error in subnormal range.

2.2 Floating-point computations depend on the architecture

With the same program containing floating-point computations, the result may be different depending to the compiler and the processor. We present in this section some architecture-dependent issues resulting in such problems.

A first cause is the fact that some processors (IBM PowerPC or Intel/HP Itanium) have a *fused multiply-add* (FMA) instruction. Specified in the latest revision of the IEEE-754 standard, the FMA operation computes $(x \times y) \pm z$ as if with unbounded range and precision, and rounds only once to the destination format. This operation can speed up and improve the accuracy of dot product, matrix multiplication and polynomial evaluation, but few processors now support it. But how should $a \times b + c \times d$ be computed? When a FMA is available, the compiler may choose either $\circ(a \times b + \circ(c \times d))$, or $\circ(\circ(a \times b) + c \times d)$, or $\circ(\circ(a \times b) + \circ(c \times d))$. And all those computations may give different results.

Another well-known cause of discrepancy happens in the IA32 architecture (Intel 386, 486, Pentium etc.) [2]. The IA32 processors feature a floating-point unit called "x87". This unit has 80-bit registers in "double extended" format (64-bit significand and 15-bit exponent), often associated to the *long double* C type. When using the x87 mode, the intermediate calculations are computed and stored in the x87 registers (80 bits). The final result is rounded to the destination format. This means that the result may be more accurate as the precision of the intermediate computations (64) is greater than the IEEE-754 double precision (53). It may also be different as the exponent range is also larger than in the usual double format.

Extended registers may also lead to double rounding, where floating-point results are rounded twice. For instance, the operations are computed in the *long double* type of x87 floating-point registers, then rounded to IEEE double precision type for storage in memory. Double rounding may yield different result from direct rounding to the destination type.

An example is given in Figure 1: we assume x is near the midpoint c of two consecutive floating-point numbers a and b in the destination format. Using round-to-nearest, with single rounding, x is rounded to b. However, with double rounding, it may firstly be rounded towards the middle c and then be rounded to a (if a is even). The results obtained in the two cases are different.

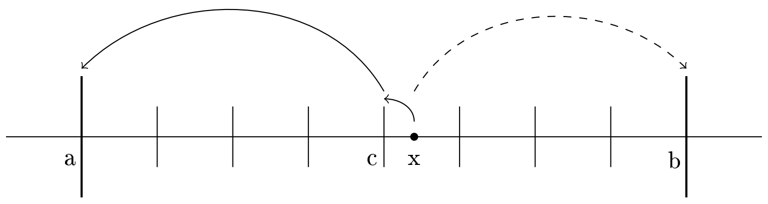


Fig. 1. Bad case for double rounding

This is illustrated by the program of Figure 2. In this example, $y = 2^{-53} + 2^{-64}$ and x are exactly representable in double precision. The values 1 and $1 + 2^{-52}$ are two consecutive floating-point numbers. With strict IEEE-754 double precision computations for `double` type, the result obtained is $z = 1 + 2^{-52}$. Otherwise, on IA32, if the computations on `double` is performed in the `long double` type inside x87 unit, then converted to double precision, $z = 1.0$.

Another example which gives inconsistencies in result between x87 and SSE is presented in Figure 3. This example will be presented and reused in Section 4. In this example, we have a function `int sign(double x)` which returns a value which is either -1 if $x < 0$, or 1 if $x \geq 0$. The function `int eps_line(double sx, double sy, double vx, double vy)` then makes a direction decision de-

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;   // y = 2^-53 + 2^-64
  double z = x + y;
  printf("z=%a\n",z);
}
```

Fig. 2. A simple program giving different answers depending on the architecture.

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}

int eps_line(double sx, double sy,double vx, double vy) {
  int s1,s2;

  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}

int main(){
  double sx = -0x1.0000000000001p0;     // sx = -1 - 2^-52
  double vx = -1.0;

  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1;    // vy = 1 - 2^-53

  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```

Fig. 3. A more complex program giving different answers depending on the architecture.

pending on the sign of a few floating-point computations. We execute this program on SSE unit and obtain that `Result = 1`. When it is performed on IA32 inside x87 unit, the result is `Result = -1`.

3 Hardware-independent bounds for floating-point computations using rounding errors

As the result of floating-point computations may depend on the compiler and the architecture, static analysis is the perfect tool, as it will verify the program without running it, therefore without enforcing the architecture or the compiler. But we need a very generic approach that allows us to give both correct and interesting properties on a floating-point computation without knowing which computation will be in fact executed. The chosen approach is to consider only the rounding error. This will be insufficient in some cases, but we believe this can give useful and sufficient results in most cases.

3.1 Rounding error in 64-bit rounding, 80-bit rounding and double rounding

We know that the choice between 64-bit, 80-bit and double rounding is the main reason that causes the discrepancies of result. We prove a rounding error bound that is valid whatever the hardware, and the chosen rounding.

We denote by $\circ_{64}$ the round-to-nearest in the `double` 64-bit type. We denote by $\circ_{80}$ the round-to-nearest to the extended 80-bit registers.

Theorem 1. *For a real number x, let $\square(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$. Then,*

$$\textit{If } |x| \geq 2^{-1022} \textit{ then } \left(\left|\frac{x-\square(x)}{x}\right| \leq 2050 \times 2^{-64} \textit{ and } |\square(x)| \geq 2^{-1022}\right) \tag{5}$$

$$\textit{else if } |x| \leq 2^{-1022} \textit{ then } \left(|x-\square(x)| \leq 2049 \times 2^{-1086} \textit{ and } |\square(x)| \leq 2^{-1022}\right) \tag{6}$$

Case 1, $\square(x) = \circ_{64}(x)$: Rounding error in 64-bit rounding

The smallest positive number in normal range is 2^{-1022} so the value 2^{-1022} is the frontier of normal/subnormal cases. The comparison of the natural rounding errors and the Theorem 1 is illustrated in Figure 4.

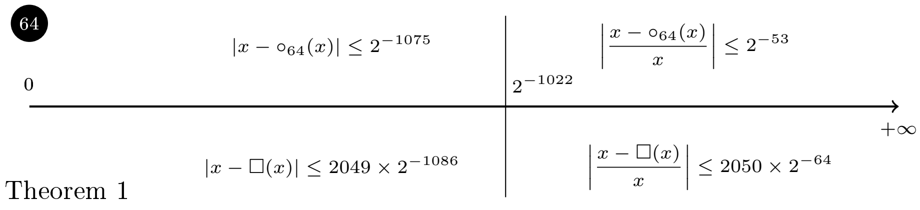

64
$|x - \circ_{64}(x)| \leq 2^{-1075}$
$\left|\frac{x - \circ_{64}(x)}{x}\right| \leq 2^{-53}$
0
2^{-1022}
$+\infty$
$|x - \square(x)| \leq 2049 \times 2^{-1086}$
$\left|\frac{x - \square(x)}{x}\right| \leq 2050 \times 2^{-64}$
Theorem 1

Fig. 4. Rounding error in 64-bit rounding vs. Theorem 1

Normal range. Based on the formula (3), if we assume $|x| \geq 2^{-1022}$, the rounding error is

$$\epsilon(x) = \left|\frac{x - \circ_{64}(x)}{x}\right| \leq 2^{-53} \tag{7}$$

We see that $2^{-53} = 2048 \times 2^{-64}$ is less than 2050×2^{-64}. Moreover, we know that round-to-nearest mode is monotone. Thus, if $|x| \geq 2^{-1022}$ then we have $|\square(x)| \geq 2^{-1022}$. The first case of Theorem 1 is held.

Subnormal range. Let $\eta_{64} = 2^{-1074}$ be the smallest positive subnormal number. For subnormal numbers, from formula (4), the rounding error is represented by the absolute error as follows, if $|x| \leq 2^{-1022}$:

$$|x - \circ_{64}(x)| \leq \frac{1}{2}\eta_{64} \tag{8}$$

The bound $\frac{1}{2}\eta_{64} = 2048 \times 2^{-1086}$ is less than 2049×2^{-1086}. In addition, as we have discussed in normal range, round-to-nearest mode is monotone. Therefore, if $|x| \leq 2^{-1022}$ then we have $|\square(x)| \leq 2^{-1022}$. The second case of Theorem 1 is also held. The correctness of Theorem 1 is proved in 64-bit rounding.

Case 2, $\square(x) = \circ_{80}$: Rounding error in 80-bit rounding

Remind that the 80-bit registers used in x87 have 64-bit significand and 15-bit exponent. Thus, the smallest positive number in normal range is then 2^{-16382} and the smallest positive subnormal number is $\eta_{80} = 2^{-16445}$.

As we have discussed in the 64-bit rounding, 2^{-1022} is also a floating-point number in 80-bit rounding and the round-to-nearest mode is monotone. Then, if $|x| \leq 2^{-1022}$, then $|\circ_{80}(x)| \leq 2^{-1022}$. And if $|x| \geq 2^{-1022}$, then $|\circ_{80}(x)| \geq 2^{-1022}$.

The bounds for rounding errors are calculated as illustrated in Figure 5.

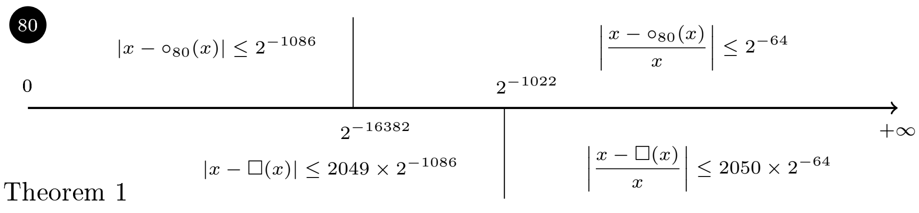

80
$|x - \circ_{80}(x)| \leq 2^{-1086}$
$\left|\frac{x - \circ_{80}(x)}{x}\right| \leq 2^{-64}$
0
2^{-1022}
2^{-16382}
$+\infty$
$|x - \square(x)| \leq 2049 \times 2^{-1086}$
$\left|\frac{x - \square(x)}{x}\right| \leq 2050 \times 2^{-64}$
Theorem 1

Fig. 5. Rounding error in 80-bit rounding vs. Theorem 1

Subnormal range. For $|x| < 2^{-16382}$, the rounding error in this case is determined by the absolute error

$$|x - \circ_{80}(x)| \leq \frac{1}{2}\eta_{80} \tag{9}$$

Normal range. For $|x| \geq 2^{-16382}$, based on (3), the relative error is:

$$\epsilon(x) = \left|\frac{x - \circ_{80}(x)}{x}\right| \leq 2^{-64} \tag{10}$$

We then split into the following cases:

- If $|x| \geq 2^{-1022}$, then x is a normal number in 80-bit rounding. Moreover, the bound 2^{-64} in (10) is much smaller than the 2050×2^{-64} of Theorem 1.
- If $2^{-16382} \leq |x| \leq 2^{-1022}$, then

$$\begin{aligned}\left|\frac{x - \circ(x)}{x}\right| \leq 2^{-64} &\Rightarrow |x - \circ(x)| \leq 2^{-64} \times |x| \\ &\Rightarrow |x - \circ(x)| \leq 2^{-64} \times 2^{-1022} = 2^{-1086}\end{aligned} \tag{11}$$

In conclusion, all bounds are much smaller than that of Theorem 1 so Theorem 1 is held in 80-bit rounding.

Case 3, $\Box(x) = \circ_{64}(\circ_{80}(x))$: Rounding error in double rounding
As $\circ_{64}$ and $\circ_{80}$ are monotone, $\Box$ is also monotone. And as 2^{-1022} is a floating-point number both in 64 and in 80 bits, if $|x| \leq 2^{-1022}$, then $|\circ_{64}(\circ_{80}(x))| \leq 2^{-1022}$. And if $|x| \geq 2^{-1022}$, then $|\circ_{64}(\circ_{80}(x))| \geq 2^{-1022}$.

The bounds for rounding errors are calculated as illustrated in Figure 6.

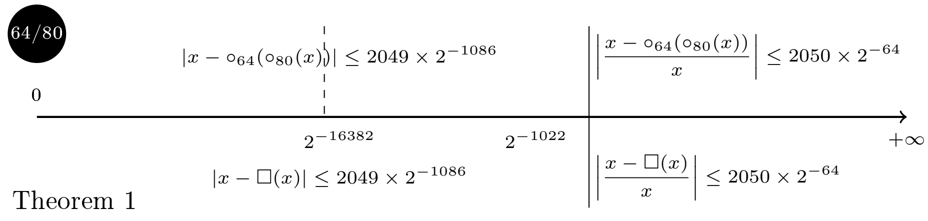

64/80
$|x - \circ_{64}(\circ_{80}(x))| \leq 2049 \times 2^{-1086}$
$\left|\frac{x - \circ_{64}(\circ_{80}(x))}{x}\right| \leq 2050 \times 2^{-64}$
0
2^{-16382}
2^{-1022}
$+\infty$
$|x - \Box(x)| \leq 2049 \times 2^{-1086}$
$\left|\frac{x - \Box(x)}{x}\right| \leq 2050 \times 2^{-64}$
Theorem 1

Fig. 6. Rounding error in double rounding vs. Theorem 1

Normal range. We first assume that $|x| \geq 2^{-1022}$, and that fact implies that $|\circ_{80}(x)| \geq 2^{-1022}$ and is also in the normal range for the 64-bit rounding. We then bound the relative error by some computations and the previous formulas:

$$\begin{aligned}
\left|\frac{x - \circ_{64}(\circ_{80}(x))}{x}\right| &\leq \left|\frac{x - \circ_{80}(x)}{x}\right| + \left|\frac{\circ_{80}(x) - \circ_{64}(\circ_{80}(x))}{x}\right| \\
&\leq \left|\frac{x - \circ_{80}(x)}{x}\right| + \left|\frac{\circ_{80}(x) - \circ_{64}(\circ_{80}(x))}{\circ_{80}(x)} \times \frac{\circ_{80}(x)}{x}\right| \\
&\leq \left|\frac{x - \circ_{80}(x)}{x}\right| + \left|\frac{\circ_{80}(x) - \circ_{64}(\circ_{80}(x))}{\circ_{80}(x)} \times (\frac{\circ_{80}(x) - x}{x} + 1)\right| \\
&\leq \left|\frac{x - \circ_{80}(x)}{x}\right| + \left|\frac{\circ_{80}(x) - \circ_{64}(\circ_{80}(x))}{\circ_{80}(x)}\right| \times \left(\left|\frac{\circ_{80}(x) - x}{x}\right| + 1\right) \\
&\leq 2^{-64} + 2^{-53} \times (2^{-64} + 1) \\
&\leq 2050 \times 2^{-64}
\end{aligned}$$

Of course, we are in the worst case and the value of 2050×2^{-64} is exactly the bound of Theorem 1.

Subnormal range. We now assume that $|x| \leq 2^{-1022}$. The absolute error to bound is $|x - \circ_{64}(\circ_{80}(x))|$. We have two cases depending on whether x is in the 80-bit normal or subnormal range.

If x is in the 80-bit subnormal range, then $|x| < 2^{-16382}$ and

$$\begin{aligned}
|x - \circ_{64}(\circ_{80}(x))| &\leq |x - \circ_{80}(x)| + |\circ_{80}(x) - \circ_{64}(\circ_{80}(x))| \\
&\leq 2^{-1086} + 2^{-1075} \\
&\leq 2049 \times 2^{-1086}
\end{aligned}$$

If x is in the 80-bit normal range, then $2^{-16382} \leq |x| < 2^{-1022}$ and

$$\begin{aligned}|x - \circ_{64}(\circ_{80}(x))| &\leq |x - \circ_{80}(x)| + |\circ_{80}(x) - \circ_{64}(\circ_{80}(x))| \\ &\leq 2^{-64} \times |x| + 2^{-1075} \\ &\leq 2^{-1086} + 2^{-1075} \\ &\leq 2049 \times 2^{-1086}\end{aligned}$$

Again, this is the worst case and this bound is the one of the Theorem 1. Theorem 1 is now proved in double rounding.

In conclusion, Theorem 1 is proved for all three roundings.

3.2 Proof in Coq

We use Coq with the help of the Gappa tactic [19] to prove the correctness of Theorem 1. The corresponding theorem in Coq is presented in Figure 7.

For technical reasons, we add the requirement that $|x| \leq 2^{35000}$. This value is large enough to satisfy all operations (addition, subtraction, multiplication, division, square root, negation and absolute value) in all types (64 or 80). We used 228 lines of Coq code to prove it, but the execution time is considerable (more than 7.5 hours). This is due to the 2^{35000} value that leads to very slow computations: if we replace it with 2^{1024}, the proof needs less than 11 seconds to be checked.

The Coq proof is exactly the one described in the preceding Section. It is not very difficult, but needs many computations and a very large number of subcases. The formal proof gives a very strong guarantee on this result, allowing its use without doubt in the Frama-C platform.

```
Theorem post_conditions_correctness: forall x f,
  Rabs x <= powerRZ 2 (35000) ->
  ( f = gappa_rounding (rounding_float roundNE 53 (1074)) x
    \/ f = gappa_rounding (rounding_float roundNE 64 (16445)) x
    \/ f = gappa_rounding (rounding_float roundNE 53 (1074))
           (gappa_rounding (rounding_float roundNE 64 (16445)) x)
    \/ f = x)
   ->
  (powerRZ 2 (-1022) <= Rabs x ->
        ( Rabs ((f-x)/x) <= 2050 * powerRZ 2 (-64)
           /\ powerRZ 2 (-1022) <= Rabs f))
   /\
  (Rabs x  <= powerRZ 2 (-1022) ->
        ( Rabs (f-x) <= 2049 * powerRZ 2 (-1086)
           /\ Rabs f <= powerRZ 2 (-1022))).
```

Fig. 7. Coq theorem certifying the correctness of Theorem 1

3.3 Hardware and compiler-independent proofs of numerical programs

Rounding error in presence of FMA

Theorem 1 gives rounding error formulas for various roundings denoted by $\Box$ (64-bit, 80-bit and double rounding). Now, let us consider the FMA that computes $x \times y + z$ with one single rounding. The question is whether a FMA was used in a computation. We therefore need an error bound that covers all the possible uses of a FMA.

The idea is very simple: we consider a FMA as a *rounded* multiplication followed by a rounded addition. And we only have to consider another possible "rounding" that is the identity: $\Box(x) = x$.

This specific "rounding" magically solves the FMA problem: the result of a FMA is then $\Box_1(x \times y + z)$, that may be considered as $\Box_1(\Box_2(x \times y) + z)$ with $\Box_2$ being the identity. So we handle in the same way all operations even in presence of FMA or not, by considering one rounding for each basic operation (addition, multiplication. . .).

Of course, this "rounding" easily verifies the formulas of Theorem 1 as we only consider rounding error. So, by considering the identity as a rounding like the others, we handle all the possible uses of the FMA in the same way as we handle multiple roundings.

Proofs of programs

Theorem 2. *If we assume the formulas of Theorem 1 on each operation (addition, subtraction, multiplication, division, square root, negation and absolute value), the final rounding error is correct whatever the architecture and the compiler, provided the compiler preserves the order of operations.*

One of the main point of the IEEE-754 standards [1] is that: *"Each of the computational operations that return a numeric result specified by this standard shall be performed as if it first produced an intermediate result correct to infinite precision and with unbounded range, and then rounded that intermediate result, if necessary, to fit in the destination's format"*. Using this property, our approach leads to a way to formally define what is the result of a hardware-independent computation: it is a value that verifies the formulas of Theorem 1.

We have put these formulas as postconditions of each floating-point operation (addition, subtraction, multiplication, division, square root, negation and absolute value) in the Frama-C platform to look into the rounding error of the whole program. This has been done from the basis of [16], by creating a new pragma called `multirounding` that implements this.

Note that absolute value and negation may produce a rounding if we take the absolute value or the negation of a 80-bit number to put into a 64-bit number.

4 A Case Study

We present a case study containing floating-point computations to illustrate our approach. The original code is in Figure 3 and may give various answers depending on the architecture/compilation. This example is part of KB3D [20][4], an aircraft conflict detection and resolution program. The idea of this case study is to make a decision corresponding to value −1 and 1 to decide if the plane will go to its left or its right. Note that KB3D is formally proved correct using PVS and assuming the calculations are exact [20]. However, in practice, when the value of the computation is small, the result may be inconsistent or incorrect.

We modified the program to provide an answer that may be 1, −1 or 0. The program is in Figure 8 is that, if the result is nonzero, then it is correct (meaning the same as if the computations were done on real numbers). If the result is 0, rounding errors may have jeopardized the result and the program is unable to give a certified answer.

[4] See also `http://research.nianet.org/fm-at-nia/KB3D/`.

```
#pragma JessieFloatModel(multirounding)
#pragma JessieIntegerModel(math)

//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;

/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  \abs(\result) <= 1 &&
  @          (\result != 0 ==> \result == l_sign(\exact(x)));
  @*/
int sign(double x, double e1, double e2) {

  if (x > e2)
    return 1;
  if (x < e1)
    return -1;
  return 0;
}

/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @    \result != 0
  @      ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @           * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/

int eps_line(double sx, double sy,double vx, double vy){
  int s1,s2;

  s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);

  return s1*s2;
}
```

Fig. 8. A case study

We define a logic function `logic integer l_sign (real x)` that gives the exact sign of a real number x, that is to say 1 if $x \geq 0$ and -1 otherwise. To mimic this perfect behavior when floating-point computations occurred, we use the function `int sign (double x, double e1, double e2)` that gives the sign of x provided we know its rounding error is between $e1$ and $e2$. In the other cases, the result is zero. The idea is to give a nonzero answer only when it is unquestionable. To illustrate this, the graphs of the two functions is in Figure 9.

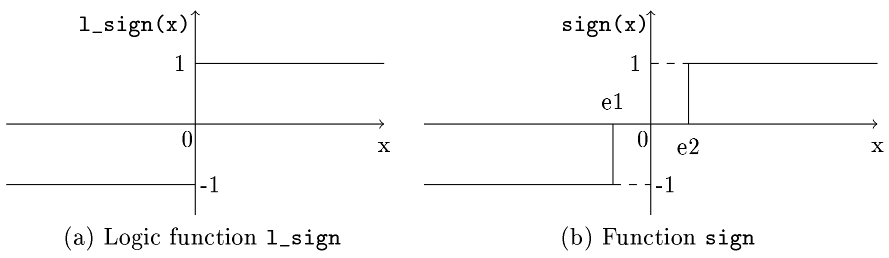


Fig. 9. Differences between two functions l_sign and $sign$

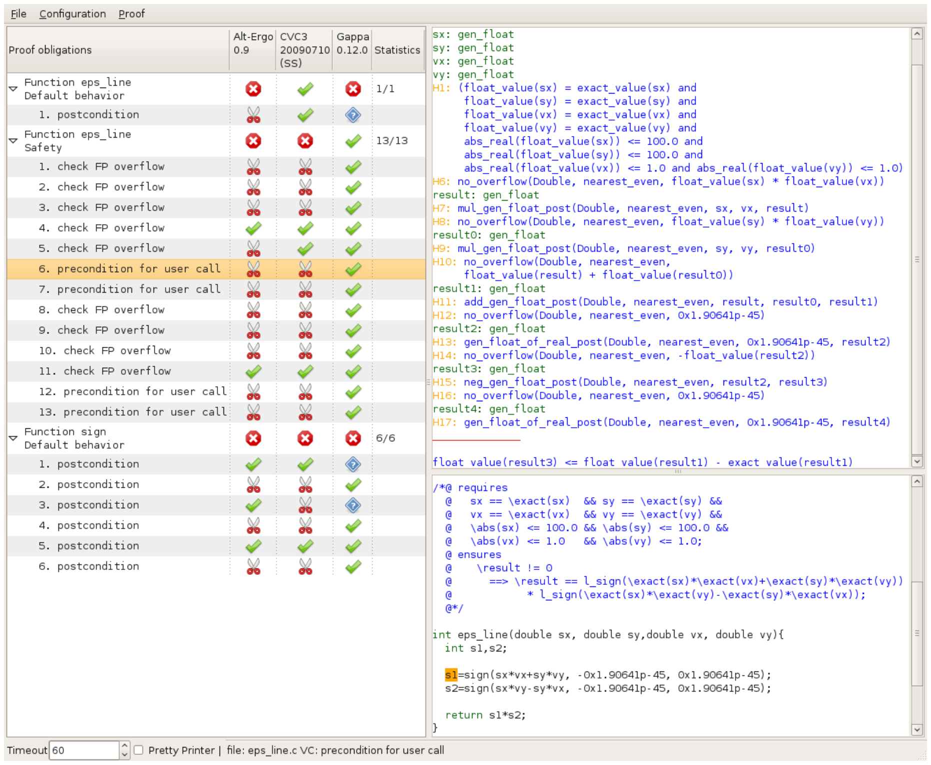
The function `int eps_line (double sx, double sy, double vx, double vy)` of Figure 8 then does the same computations as the one of Figure 3, but the result may be different. More precisely, if the modified function gives a nonzero answer, it is the correct one (it gives the correct sign). But it may answer zero (contrary to the original program) when it is unable to give a certified answer. In brief, the modified program gives the correct answer. As in interval arithmetic, the program does not lie, but it may not answer.

About the other assertions, the requirement on vx, sx... are reasonable values given the fact they are position or distance in given units. The assertions about $s1$ and $s2$ are here to help the automatic provers.

The most interesting parts are the value chosen for $e1$ and $e2$: they need to bound the rounding error of the computation $sx * vx + sy * vy$ (and its counterpart). For this, we will heavily rely on the Gappa tool [21, 22] that is intended to help verifying and formally proving properties on numerical programs. In particular, it will solve all the required proofs that no overflow occur.

In the usual formalization where all computations directly round to 64 bits, the values $e2 = -e1 = 0x1p-45$ are correct (it has been proved using the Gappa tool). With our approach and a generic rounding, we have proved that the values $e2 = -e1 = 0x1.90641p - 45$ are correct. This means that the rounding error of $sx * vx + sy * vy$ will always be smaller than this value whatever the architecture and the compiler choices. This means that, even if a FMA is used or if extended registers are used somewhere, this function *does not lie*.

The analysis of this program (obtained from the verification condition viewer gWhy [17]) is given in Figure 10. By using different automatic theorem prover:



Proof obligations	Alt-Ergo 0.9	CVC3 20090710 (SS)	Gappa 0.12.0	Statistics
Function eps_line Default behavior	✗	✓	✗	1/1
1. postcondition	✂	✓	?	
Function eps_line Safety	✗	✗	✓	13/13
1. check FP overflow	✂	✂	✓	
2. check FP overflow	✂	✂	✓	
3. check FP overflow	✂	✂	✓	
4. check FP overflow	✓	✓	✓	
5. check FP overflow	✂	✓	✓	
6. precondition for user call	✂	✂	✓	
7. precondition for user call	✂	✂	✓	
8. check FP overflow	✂	✂	✓	
9. check FP overflow	✂	✂	✓	
10. check FP overflow	✂	✂	✓	
11. check FP overflow	✓	✓	✓	
12. precondition for user call	✂	✂	✓	
13. precondition for user call	✂	✂	✓	
Function sign Default behavior	✗	✗	✗	6/6
1. postcondition	✓	✓	?	
2. postcondition	✂	✂	✓	
3. postcondition	✓	✂	?	
4. postcondition	✂	✂	✓	
5. postcondition	✓	✓	✓	
6. postcondition	✂	✂	✓	

```
sx: gen_float
sy: gen_float
vx: gen_float
vy: gen_float
H1: (float_value(sx) = exact_value(sx) and
     float_value(sy) = exact_value(sy) and
     float_value(vx) = exact_value(vx) and
     float_value(vy) = exact_value(vy) and
     abs_real(float_value(sx)) <= 100.0 and
     abs_real(float_value(sy)) <= 100.0 and
     abs_real(float_value(vx)) <= 1.0 and abs_real(float_value(vy)) <= 1.0)
H6: no_overflow(Double, nearest_even, float_value(sx) * float_value(vx))
result: gen_float
H7: mul_gen_float_post(Double, nearest_even, sx, vx, result)
H8: no_overflow(Double, nearest_even, float_value(sy) * float_value(vy))
result0: gen_float
H9: mul_gen_float_post(Double, nearest_even, sy, vy, result0)
H10: no_overflow(Double, nearest_even,
     float_value(result) + float_value(result0))
result1: gen_float
H11: add_gen_float_post(Double, nearest_even, result, result0, result1)
H12: no_overflow(Double, nearest_even, 0x1.90641p-45)
result2: gen_float
H13: gen_float_of_real_post(Double, nearest_even, 0x1.90641p-45, result2)
H14: no_overflow(Double, nearest_even, -float_value(result2))
result3: gen_float
H15: neg_gen_float_post(Double, nearest_even, result2, result3)
H16: no_overflow(Double, nearest_even, 0x1.90641p-45)
result4: gen_float
H17: gen_float_of_real_post(Double, nearest_even, 0x1.90641p-45, result4)

float value(result3) <= float value(result1) - exact value(result1)
```

```
/*@ requires
  @   sx == \exact(sx)  && sy == \exact(sy) &&
  @   vx == \exact(vx)  && vy == \exact(vy) &&
  @   \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @   \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @    \result != 0
  @      ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @            * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/

int eps_line(double sx, double sy,double vx, double vy){
  int s1,s2;

  s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);

  return s1*s2;
}
```

Timeout 60 ☐ Pretty Printer | file: eps_line.c VC: precondition for user call

Fig. 10. Result of the case study

Alt-Ergo [23], CVC3 [24], Gappa, we successfully prove all proof obligations in this program.

Nearly all the proof obligations are quick to prove. The proof that the values $e1$ and $e2$ bound the rounding error is much longer (about 60 seconds). This is due to the fact that, for each operation, we have to split into 2 cases: normal and subnormal and this creates a very large number of theorems to solve (exponential in the numbers of computations).

5 Conclusions and further work

We have proposed an approach to give correct rounding errors whatever the architecture and the choices of the compiler. This is implemented in the Frama-C framework from the Beryllium release for all basic operations: addition, subtraction, multiplication, division, square root, negation, absolute value and we have proved its correctness in Coq.

The time to run a program needs to be taken into account. With a program containing few floating-point operations, it works well. However, it will be slow with programs containing a large number of floating-point operations, but this may be enhanced in the future.

Another drawback is that we may only prove rounding errors. There is no way to prove, for example, that a computation is correct (even if it would be correct in all possible roundings). We are working on this limitation.

As we use the same conditions for all basic operations, it is both simple and efficient. Moreover, it handles both rounding according to 64-bit rounding in IEEE-754 double precision, 80-bit rounding in x87, double rounding in IA-32 architecture, and FMA in Itanium and PowerPC processors.

Note that we only talked about double precision numbers as they are the most used. This is easily applied to single precision computations the same way (with single rounding, 80-bit rounding or double rounding).

The next step would be to allow the compiler to do anything, including reorganizing the operations. This is a challenge as it may give very different results. for example, if $|e| \ll |x|$, then $(e + x) - x$ gives zero while $e + (x - x)$ gives e. Nevertheless, some ideas could probably be reused to give a loose bound on the rounding error.

Acknowledgements

We are grateful to Claude Marché for his suggestions, his help in creating a new pragma in Frama-C/Why and his constructive remarks. We thank Guillaume Melquiond for his help in the use of Gappa tool. We are grateful to César Muñoz for providing the case study and explanations. We also thank Pascal Cuoq for his helpful suggestions about the FMA.

References

1. Microprocessor Standards Subcommittee: IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008 (August 2008) 1–58
2. Monniaux, D.: The pitfalls of verifying floating-point computations. TOPLAS **30**(3) (May 2008) 12
3. : Jml-java modeling language `www.jmlspecs.org`.
4. Burdy, L., Cheon, Y., Cok, D.R., Ernst, M.D., Kiniry, J.R., Leavens, G.T., Leino, K.R.M., Poll, E.: An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT) **7**(3) (June 2005) 212–232
5. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. (2008) `http://frama-c.cea.fr/acsl.html`.
6. Barnett, M., Leino, K.R.M., Rustan, K., Leino, M., Schulte, W.: The Spec# Programming System: An Overview, Springer (2004) 49–69
7. Barrett, G.: Formal methods applied to a floating-point number system. IEEE Transactions on Software Engineering **15**(5) (1989) 611–621
8. Carreño, V.A., Miner, P.S.: Specification of the IEEE-854 floating-point standard in HOL and PVS. In: HOL95: 8th International Workshop on Higher-Order Logic Theorem Proving and Its Applications, Aspen Grove, UT (September 1995)

9. Russinoff, D.M.: A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS Journal of Computation and Mathematics **1** (1998) 148–200
10. Harrison, J.: Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory (1997)
11. O'Leary, J., Zhao, X., Gerth, R., Seger, C.J.H.: Formally verifying IEEE compliance of floating point hardware. Intel Technology Journal **3**(1) (1999)
12. Harrison, J.: Formal verification of floating point trigonometric functions. In Hunt, W.A., Johnson, S.D., eds.: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, Austin, Texas (2000) 217–233
13. Reeber, E., Sawada, J.: Combining ACL2 and an automated verification tool to verify a multiplier. In Manolios, P., Wilding, M., eds.: 6th International Workshop on the ACL2 Theorem Prover and its Applications. (August 2006) 63–70
14. Leavens, G.T.: Not a number of floating point problems. Journal of Object Technology **5**(2) (2006) 75–83
15. Boldo, S., Filliâtre, J.C.: Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, Montpellier, France (June 2007) 187–194
16. Ayad, A.: On formal methods for certifying floating-point C programs. Research Report RR-6927, INRIA (2009)
17. Filliâtre, J.C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification (CAV). Volume 4590 of LNCS., Springer (July 2007) 173–177
18. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys **23**(1) (1991) 5–47
19. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining Coq and Gappa for Certifying Floating-Point Programs. In: 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning. Volume 5625 of Lecture Notes in Artificial Intelligence., Grand Bend, Canada, Springer (July 2009) 59–74
20. Dowek, G., Muñoz, C., Carreño, V.: Provably safe coordinated strategy for distributed conflict resolution. In: Proceedings of the AIAA Guidance Navigation, and Control Conference and Exhibit 2005, AIAA-2005-6047, San Francisco, California (2005)
21. Daumas, M., Melquiond, G., Muñoz, C.: Guaranteed proofs using interval arithmetic. In Montuschi, P., Schwarz, E., eds.: Proceedings of the 17th IEEE Symposium on Computer Arithmetic, Cape Cod, MA, USA (2005) 188–195
22. Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. Transactions on Mathematical Software **37**(1) (2009)
23. Conchon, S., Contejean, E., Kanig, J.: CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers. In Krstic, S., Oliveras, A., eds.: SMT 2007: 5th International Workshop on Satisfiability Modulo. (2007)
24. Barrett, C., Tinelli, C.: CVC3. In Damm, W., Hermanns, H., eds.: Proceedings of the 19^{th} International Conference on Computer Aided Verification (CAV '07). Volume 4590 of Lecture Notes in Computer Science., Springer-Verlag (July 2007) 298–302 Berlin, Germany.