Proofs of numerical programs when the compiler optimizes

Sylvie Boldo, Thi Minh Tuyen Nguyen

Innovations Syst Softw Eng (2011) 7:151–160
DOI 10.1007/s11334-011-0151-6

SI: NFM 2010

Proofs of numerical programs when the compiler optimizes

Sylvie Boldo · Thi Minh Tuyen Nguyen

Received: 10 September 2010 / Accepted: 4 March 2011 / Published online: 18 March 2011

Abstract On certain recently developed 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 and the compiler choices. This approach is implemented in the Frama-C platform for static analysis of C code. Small case studies using this approach are entirely and automatically proved.

Keywords Floating-point arithmetic · Numerical program · Static analysis · Compiler optimization · Why platform · Frama-C platform

1 Introduction

Floating-point computations often appear in current critical systems from domains such as physics, aerospace system, nuclear simulation, etc. For such systems, hardware and software components play an important role.

This work was supported by the Hisseo project, funded by Digiteo (http://hisseo.saclay.inria.fr/) and by the Fost (ANR-08-BLAN-0246-01) project of the French national research organization (http://fost.saclay.inria.fr/).

S. Boldo (✉) · T. M. T. Nguyen
INRIA Saclay, Île-de-France, 91893 Orsay Cedex, France
e-mail: sylvie.boldo@inria.fr

T. M. T. Nguyen
e-mail: thi-minh-tuyen.nguyen@inria.fr

S. Boldo · T. M. T. Nguyen
LRI, Univ. Paris-Sud, 91405 Orsay Cedex, France

All current microprocessor architectures support an implementation of floating-point arithmetic that complies to the IEEE-754 standard [22]. However, there exist 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 features can introduce subtle inconsistencies between program executions. This means that the floating-point computations of a program running on different architectures may be different [23].

A small example is in Fig. 1. On a recent Intel processor, if compiled with default options, the result is $1+2^{-52}$. But if compiled with option `-mfpmath=387`, the compiler uses extended registers and the result is 1.

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 [8,20] for Java, ACSL [4] for C, Spec# [2] for C#, etc. to specify the requirements. For automatic analysis of floating-point codes, a successful approach is abstract interpretation based static analysis, that includes Astrée [11,24] and Fluctuat [14].

Floating-point arithmetic has been formalized since 1989 in order to formally prove hardware components or algorithms [9,18,26]. There exist less works on specifying and proving behavioral properties of floating-point programs in deductive verification systems. Leavens presented floating-point for JML in Java in 2006 [21]. Another proposal has been made in 2007 by Boldo and Filliâtre [5]. Ayad and Marché extended this to increase genericity and handle exceptional behaviors [1].

```
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. 1 A simple program giving different answers depending on the architecture

However, these works only follow the 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 options of compilers and processors. As a result, formal verification of such programs is a challenge. The purpose of this paper is to present an approach to prove numerical programs with few restrictions on the compiler and the processor.

More precisely, we require the compiler to preserve the order of operations of the C language, except additions and subtractions that may be reordered, and we only consider rounding-to-nearest mode, double precision numbers and computations. Our approach is implemented in the Frama-C platform[1] associated with Why [16] for static analysis of C code. Frama-C and its Jessie plug-in take as input an annotated C file and creates a Why file. Why then generates verification conditions that should be proved by automatic or interactive provers (Gappa, Alt-Ergo, Coq, etc.) to ensure the correctness of the C program with respect to its specification described in the annotations.

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 one computation in all possible cases (extended registers or not). Section 4 presents how to handle possible compiler optimizations, namely FMA and additions reordering. Two small case studies are presented in Sect. 5. These examples show the differences 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 [22] for floating-point arithmetic was developed to define formats and behaviors for floating-point numbers and computations. 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 so that $e_{min} \leq e \leq e_{max}$, m $(0 \leq m < 2)$ is the significand (in p bits) of the representation.

We only consider the binary 64-bit format (usually *double* in C or Java), that satisfies the format (s, m, e) with (53, –1,022, 1,023), as it concentrates all the problems. Our ideas could be re-used in other formats.

When approximating a real number x by its rounding $\circ(x)$, a rounding error happens. We here consider only round-to-nearest mode, that includes both the default rounding mode (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, a bound on the error is known [17].

If a floating-point $f = \circ(x)$ is such that$|f| \geq 2^{e_{min}}$, then f is called a normal number. We then bound the relative error: $\left|\frac{x-f}{x}\right| \leq 2^{-p}$.

For smaller f, the value of the relative error becomes large (up to 0.5). In that case, f is a subnormal number and we prefer a bound based on the absolute error: $|x - f| \leq 2^{e_{min}-p}$.

2.2 Floating-point computations depend on the architecture

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

A first cause is the fact that some processors (IBM PowerPC or Intel/HP Itanium) have a *fused multiply-add* (FMA) instruction which 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 $\circ(\circ(a \times b) + \circ(c \times d))$ which may give different results.

Another well-known cause of discrepancy happens in the IA32 architecture (Intel 386, 486, Pentium etc.) [23]. 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. 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

[1] http://frama-c.cea.fr/.

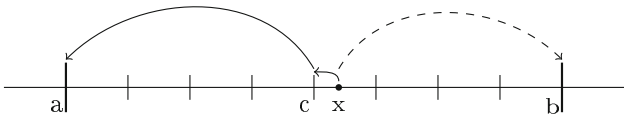

Fig. 2 Bad case for double rounding

precision type for storage in memory. Double rounding may yield different result from direct rounding to the destination type.

An example is given in Fig. 2: 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 two obtained results are different.

Let us go back to the program of Fig. 1. In this example, $y = 2^{-53} + 2^{-64}$ and x are exactly representable in double precision. With strict IEEE-754 computations for `double` type, the result obtained is $z = 1 + 2^{-52}$. Otherwise, on IA32, if the computations on `double` are 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 [23] is presented in Fig. 3. This example will be presented and reused in Sect. 5. 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 depending on a sign after several floating-point computations. If executed on the SSE unit, we obtain that `Result = 1`. When it is performed on IA32 inside x87 unit, the result is `Result = -1`.

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx,double sy,
             double vx,double vy){
  return sign(sx*vx+sy*vy)*sign(sx*vy-sy*vx);
}
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 r = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",r);
}
```

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

The last cause for discrepancies is the fact that compilers may optimize floating-point computations. This includes re-organizing additions or multiplication, use of distributivity, etc. Those mathematically correct identities usually do not hold for floating-point operations. Nevertheless, we may want to prove properties of a floating-point program, even with such optimizations. The chosen method will be explained in Sect. 4 and an example is given in Sect. 5.2.

3 Hardware-independent bounds for one floating-point operation

As we want both correct and interesting properties on a floating-point computation without knowing which rounding will be in fact executed, the chosen approach is to consider only the rounding error. This will be insufficient in some cases (exact operations for example), but we believe this can give useful and sufficient results in most cases.

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 and 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 $\circ_{64}(\circ_{80}(x))$. We have either*

$$\left(|x| \geq 2^{-1022} \text{ and } \left|\frac{x-\square(x)}{x}\right| \leq 2050 \times 2^{-64}\right)$$

or

$$\left(|x| \leq 2^{-1022} \text{ and } |x-\square(x)| \leq 2049 \times 2^{-1086}\right).$$

This theorem is the basis of our approach to correctly prove numerical programs whatever the hardware. These bounds are tight as they are reached in all cases where $\square$ is the double rounding. They are a little bigger than the ones for 64-bit rounding (2050 and 2049 instead of 2048) for both cases. These bounds are therefore both correct, very tight, and just above the 64-bit's.

In order to prove Theorem 1, we consider the rounding error with all possible values of $\square$: 64-bit rounding, 80-bit rounding and double rounding. For each case, we divide in two sub-cases: one in normal range and another in subnormal range separated by a vertical line corresponding to the underflow threshold (See Fig. 4). The detail of this proof can be found in [7]. To ensure its correctness, we formally

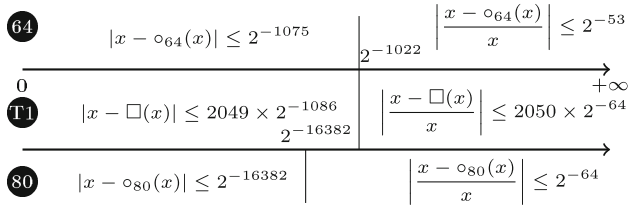

Fig. 4 Rounding error in 64-bit, 80-bit rounding versus Theorem 1

proved it. We used the Coq library developed with the help of the Gappa tactic [6] to prove the correctness of Theorem 1. The corresponding theorem and proof (228 lines) in Coq is available at http://www.lri.fr/~nguyen/research/rnd_64_80_post.html. The formal proof exactly corresponds to the one described in [7]. It is not very difficult, but many subcases and many computations are involved. The formal proof gives a very strong guarantee on this result.

In practice, we will use

$$|x - \square(x)| \leq \varepsilon \times |x| + \eta \tag{2}$$

with $\varepsilon = 2050 \times 2^{-64}$ and $\eta = 2049 \times 2^{-1086}$.

In strict IEEE-754, where inputs and outputs are on 64 bits, we can set $\eta = 0$ for addition and subtraction. Unfortunately here, inputs may be 80-bit numbers so η cannot be set to 0. Note also that absolute value and negation may produce a rounding if we put a 80-bit number into a 64-bit number.

4 Compiler choices

We have looked into what may happen for one operation. Now let us see what happens when various operations are involved.

4.1 FMA

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

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: $\square(x) = x$.

This specific "rounding" magically covers all the FMA possibilities: the result of a FMA is $\square_1(x \times y + z)$, that may be considered as $\square_1(\square_2(x \times y) + z)$ with $\square_2$ 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, etc.). Of course, the formulas of Theorem 1 easily hold for this "rounding".

What is the use of this odd rounding? The idea is that each *basic* operation (addition, subtraction, multiplication, division, square root, negation and absolute value) will be considered as rounded with a $\square$ that may be one of the four possible roundings $(\circ_{64}(x), \circ_{80}(x), \circ_{64}(\circ_{80}(x)), x)$. Let us go back to the computation of `a*b+c*d`: it becomes $\square(\square(a \times b) + \square(c \times d))$ with each $\square$ being one of the 4 roundings. It gives us 64 possibilities. In fact, only 45 possibilities are allowed (for example, the addition cannot be exact). But *all* the real possibilities are *included* in all the considered possibilities. And all of them have a rounding error bounded by Theorem 1.

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.

4.2 Associativity for the addition

For the sake of simplicity we will denote floating-point addition by $\oplus$ and floating-point subtraction by $\ominus$, when the precision is unknown (it may be 64- or 80-bit or double rounding).

Of course, floating-point addition is not associative even if compilers may re-associate additions. For example, if $|e| \ll |x|$, then $(e \oplus x) \ominus x$ gives zero while $e \oplus (x \ominus x)$ gives e. This catastrophic cancellation is the main problem for the reorganization of additions. The idea here is that we will change the rounding error formula for the addition in order to guarantee that, even if $(a + b) + c$ is transformed into $a + (b + c)$ by the compiler, the rounding error will still hold. For that, we use the following formula (with a given ε' and η'):

$$|a \oplus b - (a + b)| \leq \varepsilon' \cdot (|a| + |b|) + \eta' \tag{3}$$

Instead of an error proportional to $|a + b|$ as in (2), that is about the final result, the error is proportional to $|a| + |b|$. This is a huge difference that handles the cancellations, but may increase the rounding error.

To prove a program in multiple environments, we will change the definition of the result of an addition. More precisely, we change the operation post-condition, that is to say how an operation result is defined in the verification conditions. Here, we modify the post-conditions to cover all cases, including the fact that there are several possible results:

strict IEEE-754 standard
only one possible result: $a \oplus b = \circ_{64}(a + b)$
$\downarrow$
FMA and extended registers
$a \oplus b$ is any real s.t. $|a \oplus b - (a + b)| \leq \varepsilon |a + b| + \eta$
$\downarrow$
FMA, extended registers and addition reorganization
$a \oplus b$ is any real s.t. $|a \oplus b - (a + b)| \leq \varepsilon'(|a| + |b|) + \eta'$

Note that the strict IEEE-754 definition implies a rounding error formula of the same type but is moreover deterministic. The advantage of modifying the operation post-condition is that it also handles reordering when intermediate values are handled. For example, `x=a+b; y=x+c;` can be reordered into `y=a+(b+c)` if `x` is unused and `b+c` already computed.

To reason about any ordering of the additions, let us consider a generic algorithm for adding a sequence of numbers [19].

Algorithm 1 Let $\mathcal{S} = \{a_0, \ldots, a_n\}$.
Repeat while $\mathcal{S}$ contains more than one element
Remove two numbers x and y from $\mathcal{S}$
and add their sum $x \oplus y$ to $\mathcal{S}$.
Return the remaining element.

This generic algorithm is instantiated by the choice at each step of the two numbers that are removed from $\mathcal{S}$. We will denote by σ an ordering and by S_n^σ the result of Algorithm 1 for the ordering σ. For example, if you choose the preceding computed value and the a_i of smaller index, you get the left-associated summation $(((a_0 + a_1) + a_2) + \cdots) + a_n$.

To ensure the correctness of the approach of taking Formula 3 as post-condition, we proved the following theorem for a positive ε. We set $\varepsilon_n = (1+\varepsilon)^n - 1$.

Theorem 2 *Assume an integer n such that $n \leq \frac{1}{\varepsilon}$, a sequence of real numbers $(a_i)_{0\leq i\leq n}$ and a real I,*

We assume that, if we set the addition post-condition as: $x \oplus y$ is any real number r such that

$$|r - (x+y)| \leq \varepsilon_n \cdot (|x| + |y|) + n \cdot \eta,$$

we are able to deduce that $|S_n^{\sigma_1} - \sum_0^n a_i| \leq I$ for an ordering σ_1 of the additions.

Now we set the addition post-condition as: $x \oplus y$ is any real number r such that

$$|r - (x+y)| \leq \varepsilon \cdot |x+y| + \eta.$$

Then, whatever the ordering σ_2 of the additions, we have $|S_n^{\sigma_2} - \sum_0^n a_i| \leq I$.

This means that, if we are able to prove a bound on the rounding error for a sum in a program using our loose post-conditions (Formula (3)), then this bound is still correct whatever the compiler reorganization. The idea is what is proved using Frama-C and the loose post-conditions (Formula (3)) still holds with another ordering (in that case, we use the correct tight post-condition (Formula (2)) proved in the preceding Section).

Proof First, we prove an overestimation of $|S_n^{\sigma_2} - \sum_0^n a_i|$ with $\oplus$ having the $\varepsilon \cdot |x+y|$ property. We prove by induction on n (n being the number of operations, $n+1$ the number of operands), that whatever the ordering,

$$|S_n^{\sigma_2} - \sum_0^n a_i| \leq \varepsilon_n \sum_0^n |a_i| + n^2\eta.$$

If $n = 0$, then $|S_0^{\sigma_2} - \sum_0^0 a_i| = |a_0 - a_0| = 0 = \varepsilon_0|a_0| + 0^2\eta$ so the property holds even in this degenerate case.

If $n = 1$, then $|S_1^{\sigma_2} - \sum_0^1 a_i| = |a_0 \oplus a_1 - (a_0 + a_1)| \leq \varepsilon|a_0 + a_1| + \eta \leq \varepsilon \cdot (|a_0| + |a_1|) + \eta = \varepsilon_1 \sum_0^1 |a_i| + 1^2\eta$.

If $n = 2$, then

$$\begin{aligned}
|S_2^{\sigma_2} - \sum_0^2 a_i| &= |(a_0 \oplus a_1) \oplus a_2 - (a_0 + a_1 + a_2)| \\
&\leq \varepsilon|a_0 \oplus a_1| + \varepsilon|a_2| + \eta \\
&\leq \varepsilon \cdot (|a_0| + |a_1| + |a_2|) \\
&\quad + \varepsilon^2(|a_0| + |a_1|) + \eta + \varepsilon\eta \\
&\leq \varepsilon_2 \cdot (|a_0| + |a_1| + |a_2|) + 2\eta \\
&\leq \varepsilon_2 \sum_0^2 |a_i| + 2^2\eta.
\end{aligned}$$

The other orderings of course give the same property so the overestimation of $|S_n^{\sigma_2} - \sum_0^n a_i|$ holds for $n = 2$.

Assume that $n \geq 2$ and that the property holds for any value $m \leq n$ and let us consider a sequence $(a_i)_{0\leq i\leq n+1}$ and an ordering σ_2. As $n+1 > 2$, the value $S_{n+1}^{\sigma_2}$ is computed as the sum of two preceding computed values x and y. And x is a computed sum with a known ordering deduced from σ_2 of a part of the $\{a_0, \ldots, a_{n+1}\}$. Let I_1 be such that $x \approx \sum_{i\in I_1} a_i$. If $k = |I_1|$, then $1 \leq k < n+1$. Let us denote $I_2 = \{0, \ldots, n+1\} \setminus I_1$, then $|I_2| = n+1-k$. This includes the cases where x or y are input numbers with either k or $n+1-k$ being 1.

$$\begin{aligned}
|S_{n+1}^{\sigma_2} - \sum_0^{n+1} a_i| &= |x \oplus y - \sum_0^{n+1} a_i| \\
&\leq |x \oplus y - (x+y)| \\
&\quad + |x - \sum_{i\in I_1} a_i| + |y - \sum_{i\in I_2} a_i| \\
&\leq \varepsilon \cdot |x+y| + \eta + |x - \sum_{i\in I_1} a_i| + |y - \sum_{i\in I_2} a_i| \\
&\leq \varepsilon \sum_0^{n+1} |a_i| + \eta \\
&\quad + (1+\varepsilon)\left(|x - \sum_{i\in I_1} a_i| + |y - \sum_{i\in I_2} a_i|\right)
\end{aligned}$$

And x is the sum of $(a_i)_{i\in I_1}$ with k numbers that is less or equal to n so the induction hypothesis can be used. In a similar way, y is the sum of $(a_i)_{i\in I_2}$ with $n+1-k$ numbers that is less or equal to n. Both are using an ordering that can be deduced from σ_2.

$$\begin{aligned}
|S_{n+1}^{\sigma_2} - \sum_0^{n+1} a_i| &\leq \varepsilon \sum_0^{n+1} |a_i| + \eta + (1+\varepsilon) \\
&\quad \cdot \left(\varepsilon_k \sum_{i\in I_1} |a_i| + k^2\eta\varepsilon_{n+1-k} \sum_{i\in I_2} |a_i| + (n+1-k)^2\eta\right) \\
&\leq (\varepsilon + (1+\varepsilon) \cdot \max(\varepsilon_k, \varepsilon_{n+1-k}) \cdot \sum_0^{n+1} |a_i| \\
&\quad + \eta \cdot \left(1 + (1+\varepsilon) \cdot \left(k^2 + (n+1-k)^2\right)\right)
\end{aligned}$$

As (ε_i) is an increasing sequence, and as $k^2+(n+1-k)^2$ has its maximum value for $k=1$ or $k=n$, we have:

$$|S^{\sigma_2}_{n+1} - \sum_0^{n+1} a_i| \le (\varepsilon + (1+\varepsilon)\varepsilon_n) \cdot \sum_0^{n+1} |a_i| + \eta \cdot (1 + (1+\varepsilon)(n^2+1)) = \varepsilon_{n+1} \cdot \sum_0^{n+1} |a_i| + \eta \cdot (1 + (1+\varepsilon)(n^2+1))$$

The last equality is due to this fact: $\varepsilon + (1+\varepsilon)\varepsilon_n = \varepsilon + (1+\varepsilon)((1+\varepsilon)^n - 1) = \varepsilon + (1+\varepsilon)^{n+1} - (1+\varepsilon) = (1+\varepsilon)^{n+1} - 1 = \varepsilon_{n+1}$.

Now we bound the η term: $1 + (1+\varepsilon)(n^2+1) = n^2 + 1 + (1+\varepsilon) + \varepsilon n^2$. As $n \ge 2$, we have $1 + (1+\varepsilon)(n^2+1) \le n^2 + 1 + n + n \cdot (n\varepsilon)$. And as $n \le \frac{1}{\varepsilon}$, we deduce $1 + (1+\varepsilon)(n^2+1) \le n^2 + 1 + n + n = (n+1)^2$. Therefore,

$$|S^{\sigma_2}_{n+1} - \sum_0^{n+1} a_i| \le \varepsilon_{n+1} \cdot \sum_0^{n+1} |a_i| + (n+1)^2\eta,$$

so this overestimation property holds.

Next, we prove that $\varepsilon_n \sum_0^n |a_i| + n^2\eta \le I$. For that, we use the first hypothesis. The idea is that, if we were able to prove I with the given post-condition, then we may choose each result of an operation (fulfilling this post-condition) and see which error it creates.

For that, we will pose each operation result. More precisely,

- if neither x, nor y is an a_i, then we choose for $x \oplus y$ the value $x + y + n\eta$;
- if $x = a_i$ and y is not an a_i, then we choose for $x \oplus y$ the value $a_i + y + \varepsilon_n|a_i| + n\eta$;
- if $y = a_i$ and x is not an a_i, then we choose for $x \oplus y$ the value $x + a_i + \varepsilon_n|a_i| + n\eta$;
- if $x = a_i$ and $y = a_j$, then we choose for $x \oplus y$ the value $a_i + a_j + \varepsilon_n|a_i| + \varepsilon_n|a_j| + n\eta$.

All those results fulfill the post-condition requirements. Note also that there will be exactly n additions (whatever the ordering). Therefore,

$$I \ge |S^{\sigma_1}_n - \sum_0^n a_i| = \left|\varepsilon_n \sum_0^n |a_i| + n \cdot n\eta\right| = \varepsilon_n \sum_0^n |a_i| + n^2\eta \ge |S^{\sigma_2}_n - \sum_0^n a_i|$$

that ends the proof. □

We will use the preceding value $\varepsilon = 2050 \times 2^{-64}$ and $\eta = 2049 \times 2^{-1086}$ to handle any rounding of one operation. What is proved is that, if we put Formula (3) as post-condition of the addition and subtraction with $\varepsilon' = \varepsilon_n$ and $\eta' = n \cdot \eta$ for a sufficient n, then the produced properties will be correct, even if the compiler re-associates the additions. Note that Formula (3) subsumes Formula (2) for $n \ge 1$. Note also that $\varepsilon_n = n\varepsilon + O(\varepsilon^2)$ and that a similar value for bounding the rounding error of a sum can be found in [19].

How tight is the chosen post-condition? This is a reasonable question as we multiply the rounding error by about n. This is an intuitive demonstration of the optimality where we discard the ε^2 terms (which is reasonable as $\varepsilon \approx 2^{-53}$) and we discard underflows. We consider we are only allowed to modify the addition post-condition in the verification conditions. In that case, if we consider the post-condition of Formula (2) and if we study $(((a_0 \oplus a_1) \oplus a_2) \oplus a_3) \cdots$, then the final error is at least $n \cdot \varepsilon \cdot |a_0| + (n-1) \cdot \varepsilon \cdot |a_1| + \cdots + \varepsilon \cdot |a_n|$. To justify this, we consider an example using 64-bit computations: let $a_0 = 1$ and $a_i = 2^{-53}$, then $(((a_0 \oplus a_1) \oplus a_2) \oplus a_3) \cdots = a_0$ and the error is $n \cdot 2^{-53}$. We are only interested in the first term $(n \cdot \varepsilon \cdot |a_0|)$. Now let us assume we use Formula (3) as post-condition and that the program was written $a_0 \oplus (a_1 \oplus (a_2 \oplus (a_3 \cdots)))$ (but the compiler rewrote it in the inverse order). Then the error will be about $\varepsilon' \cdot |a_0| + 2 \cdot \varepsilon' \cdot |a_1| + \cdots + n \cdot \varepsilon' \cdot |a_n|$. As we want this last error to subsume the previous one, we need $\varepsilon' \gtrapprox n \cdot \varepsilon$ to make this approach work.

We need that the ε' of Formula (3) be ε_n and η' will be $n\eta$ but we do not know n beforehand. The question left is the choice of n. A solution is to look into the program before to have an overestimation of n. We did not put this idea in practice and decided that 16 will be enough. Of course, for linear algebra, n will be at least 100 so 16 will be insufficient, but for our examples, it will be correct. Moreover, this value can be changed if a bigger value is needed. We will therefore put in the addition post-condition $\varepsilon' = 2051 \cdot 2^{-60}$ so that $\varepsilon' \ge \varepsilon_{16} = 16\varepsilon + 256\varepsilon^2(1+\varepsilon)^{16}$ and we will put $\eta' = 16\eta = 2049 \times 2^{-1082}$.

If we constructed a post-condition for a n-term floating-point addition, the results would be better in some cases, but it would not handle the reordering that goes into intermediate values. This is why we chose to only modify the basic block of the numerical program, that is to say the operation post-conditions, and did so in the best possible way.

4.3 Other handled optimizations

Without any further work, our method handles other optimizations:

- commutativity: As we have only symmetric formulas for defining the result of an operation, an optimization such as $a + b \longrightarrow b + a$ does not endanger our analysis.
- expression factorization: As we only consider rounding errors for one operation, the fact that the compiler

factorizes or un-factorizes expressions is not a problem. This includes reordering inside intermediate results.

4.4 Conclusion on possible compiler choices

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.

The idea now is to do forward analysis of the rounding errors, that is to say propagate the errors and bound them at each step of the computation. Therefore, we have put as post-conditions the formulas of Theorem 2 for addition and subtraction and of Theorem 1 for the other operations in the Frama-C platform to look into the rounding error of the whole program. Based on [1,7], we create a new "pragma" called `multirounding` to implement this. Ordinarily, the pragma directive is the method specified by the C standard for providing additional information to the compiler, beyond what is conveyed in the language itself; here, it lets Frama-C know that floating-point computations may be done with extended registers and/or FMA and/or compiler optimizations.

In our pragma, each floating-point number is represented by two values, an exact one (a real value, as if no rounding occurred) and a rounded one (the true floating-point value). At each computation, we are only able to bound the difference between these two values, without knowing the true rounded value.

Theorem 3 *Let* $\varepsilon' = 2051 \cdot 2^{-60}$ *and* $\varepsilon = 2050 \times 2^{-64}$. *Let* $\eta' = 2049 \times 2^{-1082}$ *and* $\eta = 2049 \times 2^{-1086}$.
If we define each operation result as any real such that

$$|x \oplus y - (x + y)| \leq \varepsilon' \cdot (|x| + |y|) + \eta'$$
$$|x \ominus y - (x - y)| \leq \varepsilon' \cdot (|x| + |y|) + \eta'$$
$$|x \otimes y - (x * y)| \leq \varepsilon \cdot |x * y| + \eta$$
$$|x \oslash y - (x/y)| \leq \varepsilon \cdot |x/y| + \eta$$
$$\left|\circ\left(\sqrt{x}\right) - \sqrt{x}\right| \leq \varepsilon \cdot \left|\sqrt{x}\right| + \eta$$

and if this implies a property (*such as a rounding error*), *then this property holds whatever the architecture* (*extended registers or not, FMA or not*) *and the compiler optimizations among commutativity, addition / subtraction associativity* (*for less than 16 additions / subtractions*), *use of FMA, use of extended registers, expression factorization and unfactorization.*

This is proved from the two previous theorems.

The next question is the convenience of this approach. We have a collection of inequalities that might be useless. They are indeed useful and practical. We rely on the Gappa tool [12,13] that is intended to help verifying and formally proving properties on numerical programs. The preceding formulas have been chosen to be useful and Gappa is able to take advantage of them and give an adequate final rounding error.

5 Case studies

5.1 Avionics example

We now present a complex case study that includes possible FMA and/or extended registers use but not addition reordering. This example is part of KB3D [15],[2] an aircraft conflict detection and resolution program. The aim is to make a decision corresponding to value -1 and 1 to decide if the plane will go to its left or its right. The inputs are the position and speed of the other aircraft. Note that KB3D has been formally proved correct using PVS and under the assumption that the calculations are exact [15]. However, in practice, when the value of the computation is small, the result may be inconsistent or incorrect. The original code is in Fig. 3 and may give various answers depending on the architecture/compilation. To prove the correctness of this program which is independent to the architecture/compiler, we need to modify this program to know whether the answer is correct or not.

The modified program (See Fig. 5) provides an answer that may be $1, -1$ or 0. The idea is that, if the result is nonzero, then it is correct. If the result is 0, it means that the

```
#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=sign(sx*vx+sy*vy,-0x1.aap-42,0x1.aap-42);
  int s2=sign(sx*vy-sy*vx,-0x1.aap-42,0x1.aap-42);
  return s1*s2;
}
```

Fig. 5 Avionics program

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

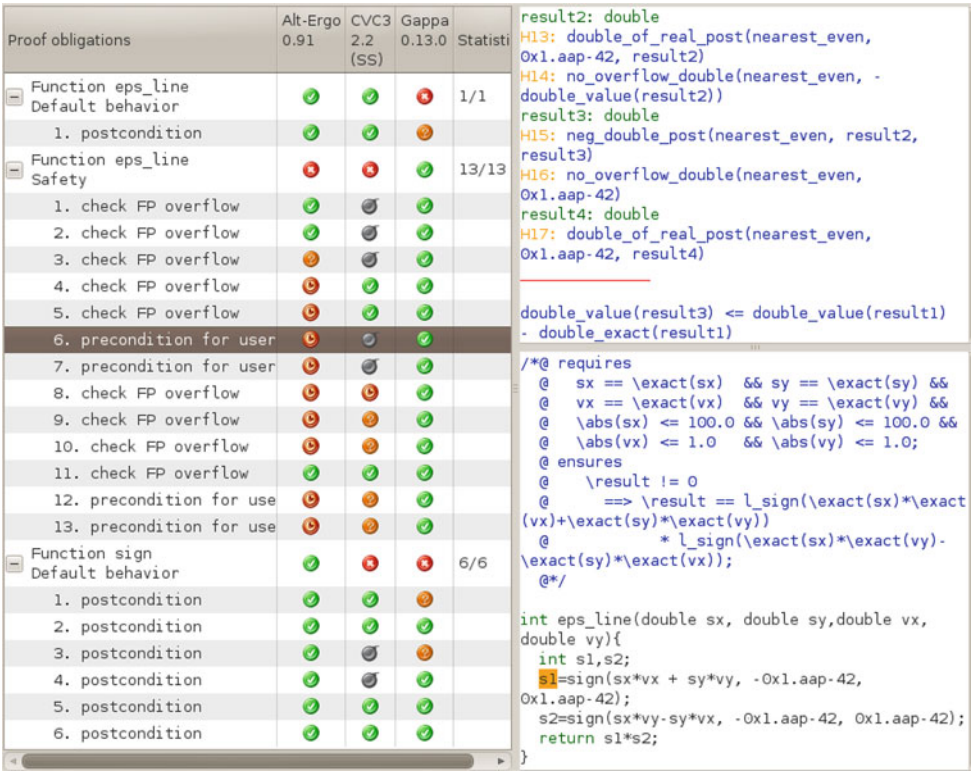

Fig. 6 Result of Fig. 5 program

result may be under the influence of the rounding errors and the program is unable to give a certified answer. The correctness of the modified program is proved with respect to the following specification: if the result is nonzero, it is the same as if the computations were done on real numbers.

In the original program, the discrepancy of the result is derived from the function `int sign(double x)`. To use this function only at the specification level, we define a logic function `logic integer l_sign (real x)` with the same meaning. Then we define another function `int sign (double x, double e1, double e2)` that gives the sign of x provided we know its rounding error is between e_1 and e_2. In the other cases, the result is zero.

The function `int eps_line (double sx, double sy, double vx, double vy)` of Fig. 5 then does the same computations as the one of Fig. 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. As in interval arithmetic, the program does not lie, but it may not answer.

About the other assertions, the given values of sx, vx, etc. are reasonable for the position and the speed of the plane. The assertions about $s1$ and $s2$ are here to help the automatic provers.

The most interesting parts are the values chosen for e_1 and e_2: they need to bound the rounding error of the computation $sx * vx + sy * vy$ (and its counterpart). For this, we will rely on the Gappa tool. 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 $e_2 = -e_1 = 0x1p - 45$ are correct (it has been proved using the Gappa tool). With our approach and a generic rounding, we have proved that the values $e_2 = -e_1 = 0x1.aap - 42$ are correct. This means that the rounding error of $sx * vx + sy * vy$ will always be smaller than this value whatever the architecture or 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 [16]) is given in Fig. 6. By combining different automatic theorem prover: Alt-Ergo [10], CVC3 [3], Gappa, we successfully prove all proof obligations in this program.

5.2 Summation

To demonstrate our choices about summation reordering, we use an example by Ogita, Rump and Oishi in [25]. Take $\delta = 2^{-54}$. Then we add $1, \delta, -1, \delta^2$ and $-\delta$. Let us assume only 64-bit roundings:

- exact computation: $1 + \delta + (-1) + \delta^2 + (-\delta) = \delta^2$
- left-associated floating-point additions: $(((1 \oplus \delta) \oplus (-1)) \oplus \delta^2) \oplus (-\delta) = -\delta$
- right-associated floating-point additions: $1 \oplus (\delta \oplus ((-1) \oplus (\delta^2 \oplus (-\delta)))) = 0$

From this example, we make a small program (see Fig. 7). Here, the reordering is critical. In the strict IEEE-754 mode (default pragma), we are able to prove that $a = -\delta$, that $b = 0$ and that the exact values of a and b are equal to δ^2, and that no overflow occur. But if the compiler reorders these additions (if we had not put parentheses for example), then these proved properties are fallacious. In the `multirounding` pragma, we are only able to prove that the rounding error of a and b is smaller than $0x1.0041p - 47$ and that no overflow occur. The rounding error is the same for a and b as this rounding error is big enough to cover all possible orderings, including the left- and the right-associated ones. Of course, the obtained error is bigger than what may really happen as there are cancellations, but this is *correct* whatever the order of operations. We noticed that Formula (3) is especially loose when cancellations happen as the error is proportional to $|x| + |y|$ instead of to $|x + y|$.

```
void main(){
  double delta = 0x1p-54;
  double a = (((1 + delta) + (-1))
              + delta*delta) + (-delta);
  double b = 1 + (delta + ((-1)
              + (delta*delta + (-delta))));
}
```

Fig. 7 Summation program

This program is fully proved by Gappa using Why/Frama-C using the `multirounding` pragma.

6 Conclusions and further work

We have proposed an approach to give correct rounding errors whatever the architecture and allowing many choices to the compiler. This is implemented in the Jessie plugin of the Frama-C framework[3] for all basic operations: addition, subtraction (with possible reordering), multiplication, division, square root, negation, absolute value.

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 and all possible reorganizations of additions and subtractions.

A 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 and compilations). This means that some subtle floating-point properties may be lost but bounding the final rounding error is usually what is wanted by engineers and this does not appear to be a big flaw.

Note that we only consider 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 idea would be to give similar formulas and to provide the basic operations with those post-conditions.

We only handle rounding-to-nearest (ties to even and ties away from zero). The reason is that directed roundings do not suffer from these problems: double rounding gives the correct answer and if some intermediate computations are done in 80-bit precision, the final result is more accurate, but still correct as it is always rounded in the correct direction. When additions are reordered, we may have different results, but all are smaller than the exact result, so the wanted property still holds whatever the order. Only rounding-to-nearest causes discrepancies.

This work is at the boundary between software and hardware for floating-point programs and this aspect of formal verification is very important. Moreover, this work deals both with normal and subnormal numbers, the latter ones being usually dismissed.

Among the many future works are the numerous other possible compiler optimizations. We have looked a little into multiplication reordering, but, due to underflow, the natural formulas are either wrong or unusable. It is very difficult to only modify the one operation formula to handle all possible underflows in a sequence of multiplications. If we had dismissed underflows, this would have been easy, but we are still trying to find a correct and useful solution. We are also interested in distributivity, meaning $a \otimes (b \oplus c) \longleftrightarrow (a \otimes b) \oplus (a \otimes c)$, and in the replacing of the division by the multiplication by the inverse: $a \oslash b \longleftrightarrow a \otimes (1 \oslash b)$ (this is known to be incorrect, but may speed up a lot of computations, such as Gaussian elimination). The interaction of all those optimizations with one another should be carefully studied.

Another interesting point is that our error bounds may be used by other tools. As shown here, considering a slightly bigger error bound for each operation suffices to give a correct final error. This means that if Fluctuat [14] for example would use them, it would also consider all our cases of hardware and of compilation.

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

[3] Available for downloading the svn version of Why on https://www.lri.fr/svn/demons/why2/trunk/ and in the next released version of Jessie and Why.

References

1. Ayad A, Marché C (2010) Multi-prover verification of floating-point programs. In: Giesl J, Hähnle R (eds) Fifth International Joint Conference on Automated Reasoning, LNAI. Springer, Edinburgh, Scotland
2. Barnett M, Leino KRM, Rustan K, Leino M, Schulte W (2004) The Spec# Programming System: an overview. Springer, Berlin, pp 49–69
3. Barrett C, Tinelli C (2007) CVC3. In: Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07). LNCS, vol 4590, pp 298–302. Springer, Berlin
4. Baudin P, Filliâtre JC, Marché C, Monate B, Moy Y, Prevosto V (2008) ACSL: ANSI/ISO C Specification Language. http://frama-c.cea.fr/acsl.html
5. Boldo S, Filliâtre JC (2007) Formal Verification of Floating-Point Programs. In: 18th IEEE International Symposium on Computer Arithmetic, pp 187–194. France
6. Boldo S, Filliâtre JC, Melquiond G (2009) Combining Coq and Gappa for Certifying Floating-Point Programs. In: 16th Symposium on the Integration of Symbolic Computation and Mechanised Reasoning. Lecture Notes in Artificial Intelligence, vol 5625, pp 59–74. Springer, Canada
7. Boldo S, Nguyen TMT (2010) Hardware-independent proofs of numerical programs. In: Muñoz C (ed) Proceedings of the Second NASA Formal Methods Symposium (NFM 2010), NASA/CP-2010-216215, pp 14–23. NASA, Langley Research Center, Hampton, USA
8. Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leavens GT, Leino KRM, Poll E (2005) An overview of JML tools and applications. Int J Softw Tools Technol Transf (STTT) 7(3):212–232
9. Carreño VA, Miner PS (1995) 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
10. Conchon S, Contejean E, Kanig J (2007) CC(X): Efficiently Combining Equality and Solvable Theories without Canonizers. In: SMT 2007—5th International Workshop on Satisfiability Modulo
11. Cousot P, Cousot R, Feret J, Mauborgne L, Miné A, Monniaux D, Rival X (2005) The ASTRÉE analyzer. In: ESOP, no. 3444 in LNCS, pp 21–30
12. Daumas M, Melquiond G (2010) Certification of bounds on expressions involving rounded operators. Trans Math Softw 37(1):2:1–2:20
13. Daumas M, Melquiond G, Muñoz C (2005) Guaranteed proofs using interval arithmetic. In: Montuschi P, Schwarz E (eds) 17th IEEE Symposium on Computer Arithmetic, pp 188–195. Cape Cod, USA
14. Delmas D, Goubault E, Putot S, Souyris J, Tekkal K, Védrine F (2009) Towards an industrial use of fluctuat on safety-critical avionics software. In: FMICS. LNCS, vol 5825. Springer, Berlin, pp 53–69
15. Dowek G, Muñoz C, Carreño V (2005) 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
16. Filliâtre JC, Marché C (2007) The Why/Krakatoa/Caduceus platform for deductive program verification. In: Computer Aided Verification (CAV). LNCS, vol 4590. Springer, Berlin, pp 173–177
17. Goldberg D (1991) What every computer scientist should know about floating point arithmetic. ACM Comput Surv 23(1):5–47
18. Harrison J (2000) Formal verification of floating point trigonometric functions. In: Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design, pp. 217–233. Austin, Texas
19. Higham NJ (2002) Accuracy and stability of numerical algorithms. SIAM, Philadelphia
20. JML-Java Modeling Language. http://www.jmlspecs.org
21. Leavens GT (2006) Not a number of floating point problems. J Object Technol 5(2):75–83
22. Microprocessor Standards Subcommittee (2008) IEEE Standard for Floating-Point Arithmetic. IEEE Std. 754-2008, pp 1–58. doi:10.1109/IEEESTD.2008.4610935
23. Monniaux D (2008) The pitfalls of verifying floating-point computations. TOPLAS 30(3):12. doi:10.1145/1353445.1353446
24. Monniaux D (2009) Analyse statique : de la théorie à la pratique. Habilitation to direct research. Université Joseph Fourier, Grenoble, France
25. Ogita T, Rump SM, Oishi S (2005) Accurate sum and dot product. SIAM J Sci Comput 26:1955–1988. doi:10.1137/030601818
26. Russinoff DM (1998) A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. LMS J Comput Math 1:148–200