Refining Abstract Interpretation-based Approximations with a Floating-point Constraint Solver

Olivier Ponsini, Claude Michel, and Michel Rueher

University of Nice–Sophia Antipolis, I3S/CNRS
BP 121, 06903 Sophia Antipolis Cedex, France
firstname.lastname@unice.fr

Abstract. Floating-point arithmetic differs from real arithmetic, which makes programming with floating-point numbers tricky. Estimating the precision of a floating-point computation in a program, i.e., estimating the difference with the result of the same sequence of operations in an idealized real number semantics, is then necessary. Tools like Fluctuat, based on abstract interpretation, have been designed to address this problem. However, these tools compute an over-approximation of the domains of the floating-point variables that may be very coarse on some tricky programs. In this paper, we use a constraint solver over floating-point numbers to refine the over-approximation computed by Fluctuat and reduce the domains of floating-point variables. Our approach could be successfully applied to C programs that are difficult for abstract interpretation techniques as implemented in Fluctuat.

Key words: Program verification; C programs; Abstract interpretation-based approximation; Floating-point computation; Floating-point constraint solver

1 Introduction

Binary floating-point numbers are the most widely-used approximation of real numbers on computers. They allow to represent a wide range of values, they are supported by dedicated efficient hardware units on most modern computer architectures, and their representation is standardized in the IEEE 754 standard. Numerous critical software applications rely on floating-point computations: in particular, simulation or control applications for physical systems in various domains as, for instance, transportation, nuclear energy, medicine, or avionics and aerospace. In these critical applications, floating-point numbers are an additional possible source of errors. Indeed, for a same computation, floating-point numbers do not behave identically to real numbers: many properties of real number arithmetic do not hold on floating-point arithmetic. The finite nature of floating-point numbers has consequences that are counterintuitive with regards to real number arithmetic [11, 19]. For instance, some decimal real numbers are

not representable (e.g., 0.1 has no exact representation with binary floating-point numbers), arithmetic operators are not associative and may be subject to phenomena such as absorption (e.g., $a + b$ is rounded to a when a is far greater than b) or cancellation (subtraction of nearly equal operands after rounding that only keeps the rounding error). Despite the IEEE 754 standard, these intrinsic phenomena of floating-point arithmetic also depend on various factors such as compilers, operating systems, or computer hardware architectures.

Although these phenomena are well known, ensuring that a program with floating-point computations reasonably approximates its original mathematical model on reals remains difficult. Indeed, numerous and infamous computer bugs, e.g., the Patriot missile failure, are due to errors in the floating-point computations. Thus, formal tools are needed for verifying programs using floating-point numbers. In this paper, we address estimating the domain, i.e., the range of values, of program floating-point variables. This is required for estimating the difference with the result of the same program in an idealized real number semantics. This may also show the absence of some runtime errors like arithmetic overflows, invalid operations, or division by zero.

Existing tools are mainly based on abstract interpretation techniques. In particular, the static analyzer Fluctuat [8] from CEA-List was successfully applied to critical industrial C programs[1] for studying rounding error propagation. Fluctuat computes two over-approximations of the domains of the variables of a program respectively considered with a semantics on real numbers and with a semantics on floating-point numbers. It is then able to bound the error made by floating-point computations. The computed approximations are precise enough for the analysis needs in many cases. However, some program constructs may lead to over-approximations so large that the analysis is not conclusive.

The contribution of this paper is an approach that complements the one of Fluctuat: we reduce the domains of the floating-point variables computed by Fluctuat thanks to our constraint solver correct over floating-point numbers FPCS [17]. We exploit the refutation power of the filtering algorithms of constraint solving to refine the results obtained by abstract interpretation. This approach is fast and efficient on programs that are problematic for Fluctuat.

Section 2 illustrates our approach with an example and presents related works. Section 3 describes our approach and details the tools it relies on: Fluctuat, and FPCS. In Sect. 4, we discuss the results of our approach on several representative programs.

2 Motivation

In this section, we illustrate our approach with a motivating example, then we briefly discuss how our approach relates to existing works.

[1] Critical applications of interest for formal verification of floating-point computations are often embedded and predominantly written in C language.

Fig. 1. Abstract domain intersection.

```
1 /* Pre−condition :  x ∈ [0, 10] */
2 double conditional(double x) {
3   double y = x*x - x;
4   if (y >= 0)
5     y = x/10;
6   else
7     y = x*x + 2;
8   return y;
9 }
```

2.1 Example

The program in Fig. 1 is cited in [10] as a difficult program for abstract interpretation based analyses. On floating-point numbers, as well as on real numbers, the value returned by the function is comprised in the interval $[0, 3[$. Indeed, the conditional statement of line 4 implies that:

- in the `if` branch, $x = 0$ or $x \in [1, 10]$, hence $y \in [0, 1]$;
- in the `else` branch, $x \in]0, 1[$, hence $y \in]2, 3[$.

Classical abstract domains (e.g., intervals, polyhedra), as well as the abstract domain of *zonotopes* used in FLUCTUAT, fail to obtain a good approximation of this value. The best interval obtained with these abstractions is $[0, 102]$.

The difficulty for these analyses is to intersect the abstract domains computed for y at lines 3 and 4. More precisely, they are unable to derive from these statements a constraint on x. As a consequence, in the `else` branch, they still estimate that x ranges over $[0, 10]$.

In contrast, a local consistency in a constraint solver is strong enough to propagate this kind of constraints over the domains of the variables. A local consistency is a property that allows to remove from the domains values that do not satisfy a constraint. Consider for instance the constraint system $\{y_0 = x_0 * x_0 - x_0, y_0 < 0, y_1 = x_0 * x_0 + 2, x_0 \in [0, 10]\}$ corresponding to the execution path through the `else` branch of the function `conditional`. From the constraints $y_0 = x_0 * x_0 - x_0$ and $y_0 < 0$, a constraint solver is able to reduce the initial domain of x_0 to $[4.94 \times 10^{-324}, 1.026]$. This reduced domain is then used to compute the one of y_1 *via* the constraint $y_1 = x_0 * x_0 + 2$, which yields $y_1 \in [2, 3.027]$. The fundamental difficulty for the constraint solver is then to perform correct computations on domains over floating-point numbers.

The principle of our approach is to build the constraint system corresponding to each executable path in a function. Next, we reduce the variable domains by $3B$-consistency filtering [15] thanks to our solver FPCS [17], a correct solver over floating-point numbers. Table 1 collects the results obtained by our approach and other analyses on the example of the function `conditional`. On this example, contrary to FLUCTUAT, our approach computes a very good approximation.

Table 1. Return domain of the `conditional` function.

Exact	$[0, 3]$
FLUCTUAT	$[0, 102]$
Constrained zonotopes	$[0, 9.72]$
FPCS	$[0, 3.027]$

In [10], the authors propose an extension to the zonotopes, the *constrained zonotopes*, which attempts to overcome the issue due to program conditional statements. This extension is defined for the real numbers and is not yet implemented in FLUCTUAT. Therefore, the result given in Tab. 1 for this extension is computed over the real numbers. The approximation is actually improved over that of FLUCTUAT, but remains less precise than that of our approach.

2.2 Related works

Three groups of methods address static validation of programs with floating-point computations: abstract interpretation based analyses, formalizations in proof assistants, and decision procedures in automatic solvers.

Analyses based on abstract interpretation represent rounding errors due to floating-point computation in their abstract domains. They are usually fast, automatic, and scalable. However, they may lack of precision and they do not generate any counter-example. ASTRÉE [6] is probably one of the most famous tool in this family of methods. By estimating the value of variables in every program point, ASTRÉE can show the absence of runtime errors, that is the absence of behaviors not defined by the programming language (e.g., division by zero, arithmetic overflow). FLUCTUAT, which is detailed in Sect. 3.1, estimates in addition the precision of the computations.

A second group of works endeavor to formalize floating-point arithmetic in proof assistants like Coq [2] or HOL [14]. Proofs of program properties are done manually in the proof assistant which guarantees proof correctness. These formalisms are not suitable for estimating the domains of program variables. Moreover, a property that cannot be proven is not necessarily false. Therefore, in these approaches, no counter-example can be generated.

The third group of works rely on theorem provers, boolean constraint solvers (SAT solvers), or solvers modulo theories (SMT solvers). The Gappa tool [9] combines in this way term rewriting from a base of theorems and interval arithmetic. The theorems rewrite arithmetic expressions so as to compensate for the loss of dependency between variables that affect interval arithmetic. Whenever the computed intervals are not precise enough, theorems can be manually introduced or the input domains can be subdivided. The cost of this semi-automatic method is then considerable. In [1], the authors propose an axiomatization of floating-point arithmetic within first-order logic that allows to automate the proofs conducted in proof assistants such as Coq by calling external SMT solvers and Gappa. Their experiments show that the automation is only partial and that human

interaction with the proof assistant is still required. Another axiomatization of floating-point arithmetic was designed for SMT solvers in [20], but the associated decision procedure is not yet available. Because of the size of floating-point variable domains, the classical bit-vector approach of SAT solvers is ineffective. An abstraction technique was devised for CBMC in [4]. It is based on under and over-approximation of floating-point numbers with respect to a given precision expressed as a number of bits of the mantissa. However, this technique remains slow.

Finally, most of these works assume a strict adherence to the IEEE 754 standard and do not take into account the effect of compilers (e.g., expression reordering) or the hardware architecture specificities (e.g., the extended precision of floating-point registers in x86 processors). Likewise, floating-point special values, such as NaN and infinities, are often ignored.

3 Proposed approach

Fluctuat computes intervals that over-approximate the domains of floating-point variables in a C program. However, in presence of some program constructs like conditional statements, Fluctuat may compute too large over-approximations. Our approach consist in refining the intervals computed by Fluctuat with a constraint solver over floating-point numbers.

We present the main characteristics of Fluctuat in Sect. 3.1, the key points of our constraint solver FPCS in Sect. 3.2, and the whole process of our approach in Sect. 3.3.

3.1 Fluctuat

Fluctuat [8] is a static analyzer for C programs specialized in estimating the precision of floating-point computations. The tool compares the behavior of the analyzed program over real numbers and over floating-point numbers. In other words, Fluctuat allows to specify ranges of values for the program input variables and computes for each program variable:

- bounds for the domain of the variable considered as a real number;
- bounds for the domain of the variable considered as a floating-point number;
- bounds for the maximum error between real and floating-point values;
- the contribution of each statement to the error associated with the variable;
- the contribution of the input variables to the error associated with the variable.

Fluctuat proceeds automatically by abstract interpretation. It uses the weakly relational abstract domain of zonotopes [12], which is a good trade-off between performance and precision. Zonotopes are sets of affine forms that improve over interval arithmetic: linear correlations between variables are preserved. To further improve the analysis precision, the tool allows to use arbitrary precision numbers or to subdivide input variable intervals.

FLUCTUAT is developed by CEA-LIST[2] and was successfully used for industrial applications of several tens of thousands of lines of code in transportation, nuclear energy, or avionics areas.

3.2 FPCS

Our approach relies on a constraint solver that was designed to solve correctly, i.e., without losing any solution, a set of constraints over floating-point numbers. To this end, FPCS uses a $2B$-consistency [15] along with projection functions adapted to floating-point arithmetic [18, 17, 3, 16]. The floating-point domains handled by FPCS are not limited to numerical values, they also include infinities. Moreover, FPCS handles all the basic arithmetic operations, as well as most of the usual mathematical functions. Type conversions are also correctly processed. FPCS targets C programs, compiled with GCC without any optimization option and intended to be run on an x86 architecture managed by a 32-bit Linux operating system.

Partial consistencies $2B$-consistency states a local property on the bounds of the domains of a variable at a single constraint level. The domain of a variable x is $2B$-consistent if, for any constraint c, there exist values in the domains of all other variables such that c can be satisfied when x is set to the upper or lower bound of its domain. More formally, the domain $D_x = [a, b]$ of x is $2B$-consistent for the constraint system S if and only if, for all n-ary constraint $c(x, x_1, \cdots, x_{n-1}) \in S$, there exist u_1 and $v_1 \in D_{x_1}, \cdots, u_{n-1}$ and $v_{n-1} \in D_{x_{n-1}}$ such that the constraints $c(a, u_1, \cdots, u_{n-1})$ and $c(b, v_1, \cdots, v_{n-1})$ hold. S is $2B$-consistent iff all its domains are $2B$-consistent.

Example For instance, let $S_1 = \{x + y = 2, y \leq x - 1, x \in [0, 100], y \in [0, 100]\}$ and $S_2 = \{x + y = 2, y \leq x - 1, x \in [1, 2], y \in [0, 1]\}$ be two constraint systems. S_1 is not $2B$-consistent. Indeed, the domain of x is not $2B$-consistent since $100 + y = 2$ is not satisfiable when $D_y = [0, 100]$. S_2 is $2B$-consistent. Indeed, D_x is $2B$-consistent since $1 + y = 2$, $2 + y = 2$, $y \leq 1 - 1$ and $y \leq 2 - 1$ are all satisfiable when $D_y = [0, 1]$. A similar reasonning can show that D_y also is $2B$-consistent in S_2.

The $2B$-consistency filtering algorithm proceeds by narrowing the domains of the variables. The approximation of the projection of a constraint c on the variables of c is the basic tool for narrowing domains. The projection $\Pi_x(c)$ of the constraint $c(x, x_1, \cdots, x_n))$ on x is the set defined as follows:

$$\Pi_x(c) = \{v \in D_x \mid \exists (v_1, \cdots, v_n) \in D_{x_1} \times \cdots \times D_{x_n} \text{ s.t. } c(v, v_1, \cdots, v_n) \text{ holds}\}.$$

The approximation of $\Pi_x(c)$ is the interval $[\min(\Pi_x(c)), \max(\Pi_x(c))]$. When the domain of a variable has been narrowed, all the constraints in which this

[2] FLUCTUAT web site: `http://www-list.cea.fr/labos/fr/LSL/fluctuat/index.html`

variable appears will be processed again. The filtering ends when no domain can be narrowed anymore. If a domain becomes empty, the system is said to be not $2B$-consistent.

Example For instance, $\Pi_x(x+y=2)$ is obtained from simple interval arithmetic as $D_x \cap (D_y \ominus_i [2,2])$ where $\ominus_i$ is the subtraction over intervals. The previous constraint system S_2 is the result of the $2B$-consistency filtering of S_1.

In general, approximations of projections cannot be computed directly because min and max of projection functions are difficult to define, especially with constraints that are not monotonic or contain multiple occurrences of the same variables. This problem is usually solved by decomposing the constraint system into a set of basic constraints for which the approximation can easily be computed. Decomposition does not change the semantics of the constraint system, however, a local consistency like $2B$-consistency is not preserved by such a rewriting.

Example For instance, $\{x+y-x=0, x \in [-1,1], y \in [0,1]\}$ is not $2B$-consistent while the semantically equivalent decomposed system $\{x+y-z=0, x=z, x \in [-1,1], y \in [0,1], z \in [-1,1]\}$ is $2B$-consistent.

On the ground of the $2B$-consistency, FPCS also implements stronger consistencies, as kB-consistencies [15], which can deal with the classical issue of multiple occurrences and reduce more substantially the bounds of the domains. A constraint system S is kB-consistent iff all its domains are kB-consistent. The domain $D_x = [a,b]$ of x is kB-consistent for S (with $k > 2$) iff the constraint systems S_a and S_b obtained from substituting a to x, resp. b to x, in S are $(k-1)B$-consistent.

Floating-point variable projections The specificity of FPCS is to implement these consistencies over the floating-point numbers, as described in [17] and extended in [3]. The main difficulty lies in computing inverse projection functions that preserve all the solutions. Indeed, if direct projections, i.e., computing the domain of y from the domain of x for a constraint like $y = f(x)$, only requires a slight adaptation of classical results on interval arithmetic, inverse projections, i.e., computing the domain of x from the one of y, do not follow the same rules because of the properties of floating-point arithmetic.

Each constraint is decomposed into an equivalent binary or ternary constraint by introducing new variables if necessary. A ternary constraint $x = y \odot_f z$, where $\odot_f$ is an arithmetic operator over the floating-point numbers, is decomposed into three projection functions:

- the direct projection, $\Pi_x(x = y \odot_f z)$;
- the first inverse projection, $\Pi_y(x = y \odot_f z)$;
- the second inverse projection, $\Pi_z(x = y \odot_f z)$.

Example For instance, suppose the rounding mode is to the nearest, the operator is the addition $\oplus_f$ and the domains are $D_x = [\underline{x}, \overline{x}]$, $D_y = [\underline{y}, \overline{y}]$ and $D_z = [\underline{z}, \overline{z}]$. The approximation of the direct projection $\Pi_x(x = y \oplus_f z)$ is $[\underline{y} \oplus_f \underline{z}, \overline{y} \oplus_f \overline{z}] \cap D_x$. However, the approximation of the first inverse projection $\Pi_y(x = y \oplus_f z)$ is more complicated and requires the definition of $\text{mid}(a, b)$ which is a floating-point number of greater format than a and b at the middle between a and b. Hence, all operations for computing the approximation will be computed in this greater format. $\Pi_y(x = y \oplus_f z)$ is then $[\text{mid}(\underline{x}, \underline{x}^-) \ominus_f \overline{z}, \text{mid}(\overline{x}, \overline{x}^+) \ominus_f \underline{z}] \cap D_y$, where x^- (resp. x^+) is the greatest (resp. smallest) floating-point number smaller (resp. greater) than x with respect to its format. Inverse projection functions for other arithmetic operators are computed in a similar way.

A binary constraint of the form $x \odot_f y$, where $\odot_f$ is a relational operator (among $==, !=, <, <=, >, >=$), is decomposed into two projection functions: $\Pi_x(x \odot_f y)$ and $\Pi_y(x \odot_f y)$. The computation of the approximation of these projection functions is mainly inspired from interval arithmetic and benefits from floating-point numbers being a totally ordered finite set.

Example For instance, suppose the operator is the strictly greater than comparator $>_f$ and the domains are $D_x = [\underline{x}, \overline{x}]$ and $D_y = [\underline{y}, \overline{y}]$. The approximation of the projection $\Pi_x(x >_f y)$ is $[\max(\underline{x}, \underline{y})^+, \overline{x}]$.

3.3 Process

The steps of the process we propose are the following ones:

1. For a given C program, we compute with FLUCTUAT a first approximation of the domains of the floating-point variables.
2. We parse the C program and build a constraint system for each executable path (see details below). We assign the domains estimated by FLUCTUAT to the program variables.
3. For each constraint system, we filter the domains using $3B$-consistency. For performance reasons, we tune the filtering to stop as soon as the domain reductions become too small in percentage of floating-point numbers.
4. We compute the union of all the domains of a variable obtained from all the constraint systems at the previous step.

We explore each execution path of a program, i.e., each path of the control flow graph, using a forward analysis (going from the beginning to the end of the program). Statements on the path are converted into SSA (Static Single Assignment) form [7], which simplifies sorting out definitions and uses of variables. Numbers and lengths of the paths are bounded since recursive function calls are forbidden and loops are unfolded a user-defined number of times. Possible states of the program at any point of an execution path are represented by a constraint system made up of a finite set of variables and constraints over these variables. Variables have domains that correspond to the implementation on machine of

the types of the C language (`int`, `float` and `double`). Rules define how each program statement modifies the possible program states by adding new constraints and variables. Execution paths are explored on-the-fly and interrupted as soon as the inconsistency of the associated constraint system is detected (simple $3B$-consistency filtering). This allows to limit the combinatorial explosion of the number of paths by not exploring most of the unreachable paths.

This technique for representing programs by constraint systems comes from our work on bounded verification of programs with CPBPV [5]. The implementation of the approach proposed in this paper relies on libraries developed for CPBPV.

4 Experiments and discussion

In this section, we present the results of our approach on characteristic programs showing when our approach is able to improve the approximation computed by FLUCTUAT. Results are rounded to three decimal places for the sake of readability and were obtained on an Intel Pentium 4 at 2.8 GHz with 1 Go memory running Linux.

4.1 Conditionals

We have presented in Sect. 2.1 the issue of intersecting abstract domains in abstract interpretation based analyses. Here, we illustrate how our approach performs in this case with a program computing the roots of a quadratic equation. This program is listed in Fig. 2 and was extracted from the GNU scientific library (GSL [22]). Roots are computed in `x0` and `x1`. The program calls two functions from the C library: `fabs` and `sqrt`, the absolute value and the square root function, respectively. The function `sqrt` is incidentally one of the few functions defined in the IEEE 754 standard. These functions are directly handled by our solver FPCS and thus can appear in constraints as is.

Table 2 shows analysis times and the approximations of the domains of variables `x0` and `x1` obtained with two configurations of the input variable domains. The first two lines in the table present the results of FLUCTUAT and FPCS when no subdivision of the input domains is done; following lines report the results of FLUCTUAT when the input domains are subdivided. In the first configuration, where $a \in [-1, 1]$, $b \in [0.5, 1]$ and $c \in [0, 2]$, the FLUCTUAT over-approximation is so large that it does not give any information on the domain of the roots, whereas our approach is able to show that $\texttt{x0} \leq 0$ and $\texttt{x1} \geq -8.064$. Nevertheless, intersection of abstract domains does not always impact so significantly on the bounds of all domains. This is illustrated by the domain of `x0` in the second configuration where $a, b, c \in [1, 1 \times 10^6]$. Even though the domain computed by FLUCTUAT is still an over-approximation, our approach does not succeed in reducing it. In contrast, for this same configuration, our approach performs again a very good reduction of the domain of `x1`.

Fig. 2. Quadratic equation roots.

```
int quadratic(double a, double b, double c) {
  double r, sgnb, temp, r1, r2
  double disc = b * b - 4 * a * c;
  if (a == 0) {
    if (b == 0)
      return 0;
    else {
      x0 = -c / b;
      return 1;
    }
  }
  if (disc > 0) {
    if (b == 0) {
      r = fabs (0.5 * sqrt (disc) / a);
      x0 = -r;
      x1 =  r;
    }
    else {
      sgnb = (b > 0 ? 1 : -1);
      temp = -0.5 * (b + sgnb * sqrt (disc));
      r1 = temp / a ;
      r2 = c / temp ;
      if (r1 < r2) {
        x0 = r1 ;
        x1 = r2 ;
      }
      else {
        x0 = r2 ;
        x1 = r1 ;
      }
    }
    return 2;
  }
  else if (disc == 0) {
    x0 = -0.5 * b / a ;
    x1 = -0.5 * b / a ;
    return 2 ;
  }
  else
    return 0;
}
```

In order to increase the analysis precision, FLUCTUAT allows to divide the domains of at most two input variables into an user-defined number of sub-domains. Analyses are then run over each combination of sub-domains and the

Table 2. Domains of the roots computed by the `quadratic` function.

	$a \in [-1,1]$ $b \in [0.5,1]$ $c \in [0,2]$			$a,b,c \in [1, 1 \times 10^6]$		
	`x0`	`x1`	time	`x0`	`x1`	time
FLUCTUAT	$[-\infty,\infty]$	$[-\infty,\infty]$	< 1 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	< 1 s
FPCS	$[-\infty, 0]$	$[-8.064,\infty]$	< 1 s	$[-2 \times 10^6, 0]$	$[-2{,}503.709, 0]$	< 1 s
FLUCTUAT a subdivided	$[-\infty, 0]$	$[-\infty,\infty]$	> 1 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	> 1 s
FLUCTUAT b subdivided	$[-\infty,\infty]$	$[-\infty,\infty]$	> 1 s	$[-2 \times 10^6, 0]$	$[-5 \times 10^5, 0]$	> 1 s
FLUCTUAT c subdivided	$[-\infty,\infty]$	$[-\infty,\infty]$	> 1 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	> 1 s
FLUCTUAT a & b subdivided	$[-\infty, 0]$	$[-\infty,\infty]$	> 10 s	$[-2 \times 10^6, 0]$	$[-1.834 \times 10^5, 0]$	> 10 s
FLUCTUAT a & c subdivided	$[-\infty, 0]$	$[-\infty,\infty]$	> 10 s	$[-2 \times 10^6, 0]$	$[-1 \times 10^6, 0]$	> 10 s
FLUCTUAT b & c subdivided	$[-\infty,\infty]$	$[-\infty,\infty]$	> 10 s	$[-2 \times 10^6, 0]$	$[-5 \times 10^5, 0]$	> 10 s

results are merged. Without any *a priori* knowledge of which sub-domains should be divided, all the combinations of one, and next, if necessary for the required precision, of two domains should be tried. The number of subdivisions of each domain is difficult to determine too: it must be the largest possible while maintaining an acceptable analysis time, and this without guarantee of improving the precision of the analysis. In Tab. 2, we set the subdivisions to 50 when only one domain is divided; otherwise, we set them to 30 for each domain. The cost in time of these subdivisions can be significant compared to the gain in precision:

- In the first configuration, subdivisions of the domain of a lead to a significant reduction of the domain of `x0` (identical to what is obtained with our approach). However, no subdivision combination could reduce the domain of `x1`.
- In the second configuration, the best reduction of the domain of `x1` is obtained by subdividing the domains of both a and b. The gain remains however quite small compared to the reduction performed by our approach. No subdivision combination could reduce the domain of `x0`.

Whenever it is necessary to subdivide all the input domains, the cost is prohibitive. Our approach turns out to be more efficient: it often improves the precision of the approximation, and its cost is low, whether the precision is improved or not. Moreover, our approach could also take advantage of the subdivision technique.

4.2 Non-linearity

The abstract domain used by FLUCTUAT is based on affine forms and thus does not allow an exact representation of non-linear operations: the image of a zono-

Fig. 3. 7^{th}-order Taylor series of function sinus.

```
double sinus(double x) {
  return x - x*x*x/6 + x*x*x*x*x/120
         + x*x*x*x*x*x*x/5040;
}
```

Fig. 4. Rump's polynomial.

```
double rump(double x, double y) {
  double f;
  f = 333.75*y*y*y*y*y*y;
  f = f + x*x*(11*x*x*y*y - y*y*y*y*y*y
              - 121*y*y*y*y - 2);
  f = f + 5.5*y*y*y*y*y*y*y*y;
  f = f + x / (2*y);
  return f;
}
```

tope by a non-linear function is not, in general, a zonotope. A non-linear operation is then approximated by introducing an error term. In contrast, our solver FPCS handles correctly most of the non-linear floating-point expressions. For instance, applied to the program of the 7^{th}-order Taylor series of function sinus, in Fig. 3, our approach improves significantly the approximation of Fluctuat, as shown in Tab. 3 in column `sinus`.

Nevertheless, this is not always the case: applied to the polynomial of the program in Fig. 4, extracted from [21], our solver does not succeed in reducing the domain computed by Fluctuat, as shown in Tab. 3 in column `rump`. This polynomial is however quite peculiar since it was devised for showing a catastrophic cancellation phenomenon on some specific hardware architectures whatever the floating-point number precision is.

Analysis times for these examples are under the second for each method.

4.3 Loops

We illustrate the behavior of our approach when applied to programs containing `while` loops with a program that computes an approximate value, with an error of 1×10^{-2}, of the square root of a number greater than 4. The algorithm is based on the so-called Babylonian method and is shown in Fig. 5.

The approximation of the return value of the function and the analysis times are reported in Tab. 4 for Fluctuat and FPCS.

Fluctuat unfolds loops a bounded number of times before applying the widening operator of abstract interpretation. The widening operator allows the analysis to quickly find a fixed point and terminate. However, combined with

Table 3. Domains of the return value of `sinus` and `rump` functions.

	`sinus` $x \in [-1, 1]$	`rump` $x \in [7 \times 10^4, 8 \times 10^4]$ $y \in [3 \times 10^4, 4 \times 10^4]$
FLUCTUAT	$[-1.009, 1.009]$	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$
FPCS	$[-0.853, 0.852]$	$[-1.168 \times 10^{37}, 1.992 \times 10^{37}]$

Fig. 5. Square root function.

```
double sqrt(double x) {
  double xn, xn1;

  xn = x/2;
  xn1 = 0.5*(xn + x/xn);
  while (xn-xn1 > 1e-2) {
    xn = xn1;
    xn1 = 0.5*(xn + x/xn);
  }
  return xn1;
}
```

the intersection of abstract domain issue, this operator may lead to very large over-approximations. This situation occurs for the configuration $x \in [5, 10]$ in the analysis of the return value of `sqrt`: FLUCTUAT over-approximates by the largest possible domain $[-\infty, \infty]$.

In our approach, we do not try to analyze the behavior of loops: we just unfold the loops N times, where N is a user-defined parameter[3]. We only try to reduce the domains computed by FLUCTUAT if, from a number of unfoldings k lesser than N, the entry conditions of the loops are always false. Table 4 shows the results we obtain with $N = 10$. Unfolding can quickly become time-consuming, but the gain in precision can be significant too: for instance, in the second configuration ($x \in [5, 10]$), we compute for `sqrt` the domain $[2.232, 3.168]$ instead of the domain $[-\infty, \infty]$ obtained by FLUCTUAT.

5 Conclusion

In this paper, we have introduced a new approach for refining the approximations of floating-point variable domains computed by the static analyzer FLUCTUAT for C programs. This approach relies on our constraint solver, FPCS, which is correct over floating-point numbers. We exploit the refutation ability of $3B$-consistency filtering to reduce the domains first computed by FLUCTUAT.

[3] We can also use FLUCTUAT to estimate a bound on the number of necessary unfoldings [13].

Table 4. Domain of the return value of the `sqrt` function.

	$x \in [4.5, 5.5]$		$x \in [5, 10]$	
FLUCTUAT	$[2.116, 2.354]$	< 1 s	$[-\infty, \infty]$	< 1 s
FPCS	$[2.121, 2.347]$	3 s	$[2.232, 3.168]$	4 s

We have shown that this approach is fast and efficient on programs that are representative of the difficulties of FLUCTUAT (conditional constructs and non-linearities). However, our approach does not substitute for FLUCTUAT. Indeed, if our solver FPCS does not start from the domains of the variables reduced beforehand by FLUCTUAT, i.e. if the initial domains are all $[-\infty, \infty]$, the results are often not as good as when starting from the domains computed by FLUCTUAT. Therefore, our approach is complementary to the one of FLUCTUAT. Stronger consistencies could also be used to further improve the over-approximation precision, but at the expense of the time of the analysis. Only the user can set the acceptable trade-off for its specific problems.

The natural extension of this work is to study how FLUCTUAT could benefit back from the more precise domains that our approach computes. The best strategy for FPCS and FLUCTUAT to cooperate has yet to be defined, e.g., only calling FPCS on identified program constructs that may lead FLUCTUAT to over-approximate.

Acknowledgments. The authors gratefully acknowledge Sylvie Putot and Éric Goubault from CEA-LIST for their advice and help on using FLUCTUAT.

References

1. Ayad, A., Marché, C.: Multi-prover verification of floating-point programs. In: IJCAR. LNCS, vol. 6173, pp. 127–141. Springer (2010)
2. Boldo, S., Filliâtre, J.C.: Formal verification of floating-point programs. In: 18th IEEE Symposium on Computer Arithmetic. pp. 187–194. IEEE (2007)
3. Botella, B., Gotlieb, A., Michel, C.: Symbolic execution of floating-point computations. Software Testing, Verification and Reliability 16(2), 97–121 (2006)
4. Brillout, A., Kroening, D., Wahl, T.: Mixed abstractions for floating-point arithmetic. In: 9th International Conference on Formal Methods in Computer-Aided Design. pp. 69–76. IEEE (2009)
5. Collavizza, H., Rueher, M., Hentenryck, P.V.: A constraint-programming framework for bounded program verification. Constraints Journal 15(2), 238–264 (2010)
6. Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of static analyzers: A comparison with ASTRÉE. In: 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering. pp. 3–20. IEEE (2007)
7. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems 13(4), 451–490 (1991)
8. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of Fluctuat on safety-critical avionics software. In: FMICS. LNCS, vol. 5825, pp. 53–69. Springer (2009)

9. de Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: ACM Symposium on Applied Computing. pp. 1318–1322. ACM (2006)
10. Ghorbal, K., Goubault, E., Putot, S.: A logical product approach to zonotope intersection. In: CAV. LNCS, vol. 6174, pp. 212–226. Springer (2010)
11. Goldberg, D.: What every computer scientist should know about floating point arithmetic. ACM Computing Surveys 23(1), 5–48 (1991)
12. Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: SAS. LNCS, vol. 4134, pp. 18–34. Springer (2006)
13. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. LNCS, vol. 6538, pp. 232–247. Springer (2011)
14. Harrison, J.: A machine-checked theory of floating-point arithmetic. In: TPHOLs. LNCS, vol. 1690, pp. 113–130. Springer-Verlag (1999)
15. Lhomme, O.: Consistency techniques for numeric CSPs. In: 13th International Joint Conference on Artificial Intelligence. pp. 232–238 (1993)
16. Marre, B., Michel, C.: Improving the floating point addition and subtraction constraints. In: CP. LNCS, vol. 6308, pp. 360–367. Springer (2010)
17. Michel, C.: Exact projection functions for floating-point number constraints. In: 7th International Symposium on Artificial Intelligence and Mathematics (2002), `http://rutcor.rutgers.edu/~amai/aimath02/PAPERS/21.ps`
18. Michel, C., Rueher, M., Lebbah, Y.: Solving constraints over floating-point numbers. In: CP. LNCS, vol. 2239, pp. 524–538. Springer Verlag (2001)
19. Monniaux, D.: The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems 30(3), 12:1–12:41 (2008)
20. Rümmer, P., Wahl, T.: An smt-lib theory of binary floating-point arithmetic. In: 8th International Workshop on Satisfiability Modulo Theories (SMT) at FLoC (2010)
21. Rump, S.M.: Verification methods: Rigorous results using floating-point arithmetic. Acta Numerica 19, 287–449 (2010)
22. The GSL Team: GNU Scientific Library Reference Manual, 1.14 edn. (2010), `http://www.gnu.org/software/gsl/`