Optimizing the accuracy of a rocket trajectory simulation by program transformation

Nasrine Damouche, Matthieu Martel, Alexandre Chapoutot

Optimizing the Accuracy of a Rocket Trajectory Simulation by Program Transformation[1]

Nasrine Damouche*,⋆, Matthieu Martel*,⋆
*University of Perpignan, DALI, 66860, France
⋆LIRMM, Univ. Montpellier & CNRS, France
first.last@univ-perp.fr

Alexandre Chapoutot†
†ENSTA ParisTech, Palaiseau, France
first.last@ensta-paristech.fr

ABSTRACT

Static analysis by abstract interpretation is one of the most successful techniques used to over-approximate the roundoff errors in numerical programs. In our case, we are interested in using this method to improve the accuracy of programs which perform floating-point computations, known for their sensitivity to the way formulas are written. We are interested in transforming automatically pieces of code by applying to them several rewriting rules. In this article, we demonstrate the effectiveness of our approach on a non-trivial numerical simulation code.

Categories and Subject Descriptors

F.3.2 [**Logics and Meaning of Programs**]: Semantics of Programming Languages—*Program Analysis*; G.1.0 [**Mathematics of Computing**]: Numerical Analysis—*Computer Arithmetic*.

General Terms

Algorithms, Languages, Theory, Verification.

Keywords

Program Transformation, Abstract Interpretation, Compiler Optimizations, Floating-Point Numbers, Accuracy.

1. INTRODUCTION

From the years 1990 and onwards, many industries have suffered from major worries after numerical disasters. Examples go from the sinking of the Sleipner A platform in 1991 in the North Sea to an error in measuring the results at the Olympic Games at London in 2012. One need of these industries is to improve the numerical accuracy of their programs in order to avoid dramatical consequences such as the ones mentioned earlier. Recently, a collection of work to optimize the accuracy of programs based on floating-point arithmetic[1] has been done, e.g., the work by A. Ioualalen [5] concerning the rewriting of arithmetic expressions. Our objective is to go one step further than transforming arithmetic expressions by handling pieces of code containing assignments, conditionals and loops [3]. To optimize programs, we use static analysis by abstract interpretation [2, 4] to over-approximate the roundoff errors as well as a set of rewriting rules for the transformation itself. In this article, we present our tool and we give experimental results to optimize a full application which computes the trajectory of a rocket around the Earth.

http://dx.doi.org/10.1145/2742854.2742894 ...$15.00.

2. ARITHMETIC EXPRESSIONS

The expressions accepted by our tool are constants, variables, operations $\diamond \in \{+, -, \times, \div, \sqrt{}\}$ and trigonometric functions. We have already mentioned former work [5] whose interest is to transform arithmetic expression using Abstract Program Expansion Graph (APEG). This structure is made of abstraction boxes, containing a large number of equivalent expressions up to associativity and commutativity. The APEGs also contain equivalence classes which consist of offering a choice of alternative nodes to build an expression. It copes with the combinatory problem by remaining it in polynomial size. An example of APEG $\mathcal{A}$ is given in Figure 1. An equivalence class is denoted by a dotted ellipse. The APEG $\mathcal{A}$ is:

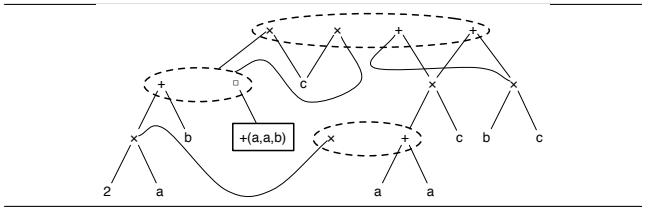

Figure 1: APEG for the expression $e = \big((a+a)+b\big)\times c$

$$\left\{ \begin{array}{c} \big((a+a)+b\big)\times c,\ \big((a+b)+a\big)\times c,\ \big((b+a)+a\big)\times c, \\ \big((2\times a)+b\big)\times c,\ c\times\big((a+a)+b\big),\ c\times\big((a+b)+a\big), \\ c\times\big((b+a)+a\big),\ c\times\big((2\times a)+b\big),\ (a+a)\times c+b\times c,\ldots \end{array} \right\}.$$

The last step of the transformation consists of evaluating the expressions in $\mathcal{A}$ with the abstract semantics to select the most accurate one. We compute safe bounds on the accuracy using abstract values defined by a pair of intervals denoting the range of the floating-point value seen by the program and the range of the error [3, 5].

3. COMMANDS

Our commands are made of assignments, conditionals, loops and sequences of commands. In order to optimize

[1]This work was supported by the ANR Project ANR-12-INSE-0007 "CAFEIN".

numerical programs, we use a set of transformation rules presented as sequents. The use of each of these rules needs some conditions to be satisfied. If we take the rules concerning the assignments, we find: Rule $(A1)$ discards an assignment after saving it in the memory and the second one, $(A2)$, allows one to rewrite an assignment by using the information memorized, to inline it in a second expression in order to build a larger expression. By inlining expressions in assignments when transforming programs, we create large formulas. In our implementation, we slice these formulas at a defined level of the syntactic tree and we assign the sub-expressions to intermediary variables. Finally, we inject them into the main program.

For example, we take a code with three variables x, y and z and constants $a = 0.1$, $b = 0.01$, $c = 0.001$ and $d = 0.0001$. We aim at optimizing z.

$$\langle \mathtt{x = a + b;\ y = c + d;\ z = x + y,}\ \delta\rangle$$
$$\xrightarrow[(A1)]{*} \langle \mathtt{z = x + y}, \delta'' = \delta'[\mathtt{y} \mapsto \mathtt{c + d}]\rangle \xrightarrow[(A2)]{} \langle \mathtt{z = ((d + c) + b) + a}, \delta''\rangle$$

We remove the variable x and memorize it in δ. So, the first assignment is discarded and the new environment is $\delta[x \mapsto a + b]$. We then repeat the same process by using $(A1)$ on y. We must not remove z because it is the variable to be optimized. Then, we substitute x and y by their value in δ and we transform the expression as seen in Section 2.

The second kind of rules deals with conditionals. If the condition is statically known, we execute the right branch, otherwise we rewrite both branches of the conditional. Other rules concerning the conditional consist of re-inserting variables that we have not to discard. For the while loop, one rule shows how to rewrite the body of the loop, and the other one is similar to the last one seen in conditionals. At last, we use some rules dealing with sequences of commands.

4. EXPERIMENTS RESULTS

In order to perform experiments with our tool, we have taken an example involving the positions of a rocket and a satellite in space. It consists of simulating their trajectories around the Earth using the Cartesian and polar systems, in order to project the gravitational forces in the system composed of the Earth, the rocket and the satellite. Note that the coordinates of the satellite u_i and of the rocket w_i, $1 \leq i \leq 4$ are computed by Euler's method.

The program corresponding to this example is given in Figure 3. The `else` branch is similar to the `then` branch at the difference that w'_2 and w'_4 are computed without the expression $A \cdot w_i/(M_f - A \cdot t) \cdot dt$. At the end of the loop, variables are updated, for example $u_1 = u'_1$, etc. Figure 2

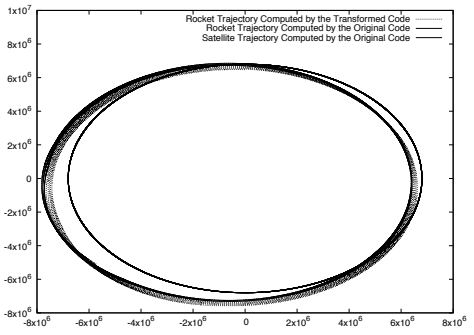

Figure 2: Simulated trajectories

Constants are: The radius of the Earth $R = 6.4 \cdot 10^6\ m$, the gravity $G = 6.67428 \cdot 10^{-11}\ m^3 \cdot kg^{-1} \cdot s^{-2}$, the mass of the Earth $M_t = 5.9736 \cdot 10^{24}\ kg$, the mass of the rocket $M_f = 150000\ kg$ and the gas mass ejected by seconde $A = 140\ kg \cdot s^{-1}$. The release rate of the rocket v_l is $0.7 \cdot \sqrt{\frac{G \cdot M_t}{D}}$ with $D = R + 4.0 \cdot 10^5\ m$ the distance between the rocket and the Earth. Other variables are set to 0.

$r_0 = D$; $viss = \sqrt{\frac{G \times M_t}{r_0}}$; $v\theta_0 = \frac{viss}{r_0}$; $nbsteps = \frac{T}{dt}$; $r_f = R$;
$vl_{rad} = \frac{v_l}{r_0}$; $v\theta_f = 1.1 \times vl_{rad}$; $mf_0 = M_f$;
`while` $(i < nbsteps)$ `do` {
  `if` $(m_f > 0.0)$ `then` {
    $u'_1 = u_2 \times d_t + u_1$; $u'_3 = u_4 \times d_t + u_3$;
    $w'_1 = w_2 \times d_t + w_1$; $w'_3 = w_4 \times d_t + w_3$;
    $u'_2 = -G \cdot M_t/(u_1 \cdot u_1) \cdot dt + u_1 \cdot u_4 \cdot u_4 \cdot dt + u_2$;
    $u'_4 = -2.0 \cdot u_2 \cdot u_4/u_1 \cdot dt + u_4$;
    $w'_2 = -G \cdot M_t/(w_1 \cdot w_1) \cdot dt + w_1 \cdot w_4 \cdot w_4 \cdot dt$
      $+(A \cdot w_2)/(M_f - A \cdot t) \cdot dt + w_2$;
    $w'_4 = -2.0 \cdot w_2 \cdot w_4/w_1 \cdot dt + A \cdot w_4/(M_f - A \cdot t) \cdot dt + w_4$;
    $m'_f = m_f - A \times t$; $t = t + d_t$; } `else` {[...]};
  $x = (\mathtt{cos}(w'_3) * w'_1)$; $y = (\mathtt{sin}(w'_3) * w'_1)$; $i = i + 1.0$; [...] }

Figure 3: Original simulation code

`while` $(i < nbsteps)$ `do` {
  `if` $(m_f > 0.0)$ `then` {
    $TMP_2 = (u_1 \cdot u_1)$; $TMP_4 = (59735.99e^{20}/(w_1 \cdot w_1))$;
    $TMP_{10} = (140.0 \cdot t)$; $m'_f = (m_f + (t \cdot (-140.0)))$;
    $u'_1 = (u_1 + (u_2 \cdot 0.1))$; $u'_3 = (u_3 + (u_4 \cdot 0.1))$;
    $w'_1 = (w_1 + (w_2 \cdot 0.1))$; $w'_3 = (w_3 + (w_4 \cdot 0.1))$;
    $u'_2 = ((-((0.66743e^{-10} \cdot (59735.99e^{20}/TMP_2)) \cdot 0.1)$
      $+(((u_1 \cdot u_4) \cdot u_4) \cdot 0.1)) + u_2)$;
    $u'_4 = (((-2.0 \cdot (u_2 \cdot (u_4/u_1))) \cdot 0.1) + u_4)$;
    $w'_2 = (((-((0.66743e^{-10} \cdot TMP_4) \cdot 0.1) + (((w_1 \cdot w_4) \cdot w_4) \cdot 0.1))$
      $+(((140.0 \cdot w_2)/(150000.0 - (140.0 \cdot t))) \cdot 0.1)) + w_2)$;
    $w'_4 = (((-2.0 \cdot (w_2 \cdot (w_4/w_1))) \cdot 0.1)$
      $+((140.0 \cdot ((w_4/(150000.0 - TMP_{10})) \cdot 0.1)) + w_4))$;
    $t = t + 0.1$; } `else` { [...] };
  $x = (cos(((0.1 * w'_4) + w'_3)) * (w'_1 + (w'_2 * 0.1)))$;
  $y = (sin(((0.1 * w'_4) + w'_3)) * (w'_1 + (w'_2 * 0.1)))$; $i = i + 1.0$; [...]}

Figure 4: Transformed simulation code

shows the difference between the trajectories before and after transformation, after 2.25 days of simulated time.

5. CONCLUSION

We have presented results obtained with our tool that rewrites codes to improve their accuracy. Future work consists of extending our tool to, first, other programming patterns like arrays and especially functions and, second, to deal with optimizing many reference variables at once.

6. REFERENCES

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[2] Cousot, P., and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs. In *POPL* (1977).

[3] Damouche, N., Martel, M., and Chapoutot, A. Intra-procedural optimization of the numerical accuracy of programs. In *FMICS* (2015). To be published.

[4] Goubault, E., and Putot, S. Static analysis of finite precision comp. In *VMCAI* (2011), vol. 6538 of *LNCS*.

[5] Ioualalen, A., and Martel, M. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS* (2012).