Optimizing the Accuracy of a Rocket Trajectory Simulation by Program Transformation

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

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.

2. ARITHMETIC EXPRESSIONS
The expressions accepted by our tool are constants, variables, operations $o \in \{+, -, \times, \div, \sqrt{\cdot}\}$ 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 $A$ is given in Fig.

\begin{figure}[h]
  \centering
  \includegraphics[width=0.5\textwidth]{apeg.png}
  \caption{APEG for the expression $e = ((a+a) + b) \times c$}
\end{figure}

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM International Conference on Computing Frontiers 2015, May 18-21, 2015, Ischia, Italy
Copyright 2015 ACM 978-1-4503-3358-0/15/05.
http://dx.doi.org/10.1145/2742854.2742894 ...$15.00.

\[\text{This work was supported by the ANR Project ANR-12-INSE-0007 "CAFEIN".}\]
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 information memorized, to inline it in a second expression one, (concerning the assignments, we find: Rule (A3)).

The use of each of these rules needs 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 information memorized, to inline it in a second expression one, (concerning the assignments, we find: Rule (A3)).

We aim at optimizing $z$.

$$\begin{align*}
(A1) &\quad (x = a + b; y = c + d; z = x + y, \delta) \\
(A2) &\quad (x = (d + c) + b) + a, \delta')
\end{align*}$$

We remove the variable $x$ and memorize it in $\delta$. 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 $\delta$ and we transform the expression as seen in Section 2.

The second kind of rules deals with conditions. 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 $w_j'$ and $w_k'$ 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

Constants are: The radius of the Earth $R = 6.4 \times 10^6$ m, the gravity $G = 6.6738 \times 10^{-11}$ m$^3$ kg$^{-1}$ s$^{-2}$, the mass of the Earth $M_e = 5.9736 \times 10^{24}$ kg, the mass of the rocket $M_f = 150000$ kg and the gas mass ejected by seconds $A = 140$ kg$^{-1}$ s$^{-1}$. The release rate of the rocket $v_1 = 0.7 \cdot \sqrt{\frac{2M_f}{M_e}}$ with $D = R + 4.0 \times 10^6$ m the distance between the rocket and the Earth. Other variables are set to 0.

$$\begin{align*}
\rho_0 &= D, \quad w_{vars} = \sqrt{\frac{2M_f}{M_e}}; \quad \rho_{0b} = \frac{2M_f}{M_e} \text{nbsteps} = \frac{2M_f}{M_e}; \quad r_f = R; \\
\rho_{vars} &= \rho_{0b}; \quad \rho_f = 1.1 \times \rho_{vars}; \quad m_{0b} = M_f; \\
\text{while} (t < \text{nbsteps}) \text{ do }
\begin{cases}
\text{if} (m_f > 0) \text{ then }
\begin{align*}
w_0' &= w_2 \times dt + w_1; & w_1' &= w_4 \times dt + w_3; \\
w_2' &= w_3 \times dt + w_1; & w_3' &= w_4 \times dt + w_2; \\
\end{align*}
\end{cases}
\begin{cases}
\text{else}
\begin{align*}
\delta &= -G / (M_f / (w_1 + u_1) \cdot dt + w_3); \\
\end{align*}
\end{cases}
\end{cases}
\begin{cases}
\text{if} (m_f > 0) \text{ then }
\begin{align*}
\delta &= \frac{w_1' - w_2'}{2}; \\
\delta &= \frac{w_3' - w_4'}{2}; \\
\end{align*}
\end{cases}
\end{cases}
\end{align*}$$

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


Figure 2: Simulated trajectories

Figure 3: Original simulation code

Figure 4: Transformed simulation code

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