Code Synthesis to Optimize Accuracy and Execution Time of Floating-Point Programs

Laurent Thévenoux, Matthieu Martel, Philippe Langlois

Code Synthesis to Optimize Accuracy and Execution Time of Floating-Point Programs*

Laurent Thévenoux[1], Matthieu Martel[2], and Philippe Langlois[2]

[1]Inria – Laboratoire LIP, (CNRS, ENS de Lyon, Inria, UCBL), Univ. de Lyon, France
[2]UPVD – Équipe DALI, (CNRS, LIRMM, UPVD, UM2), Univ. de Perpignan, France
laurent.thevenoux@ens-lyon.fr, {matthieu.martel, langlois}@univ-perp.fr

Keywords: code synthesis, compensation, execution-time performance, floating-point arithmetic, multi-criteria optimization, numerical accuracy.

Since the precision of standard floating-point arithmetic [6] is finite, numerical programs based on it can suffer from rounding errors. To avoid such accuracy losses, code writers must be careful by using floating-point numbers. Unfortunately, they can not take care of every accuracy issues, even if they are floating-point experts. Nevertheless, several methods have been proposed to help developers to improve the accuracy of their floating-point programs [2, 3, 5].

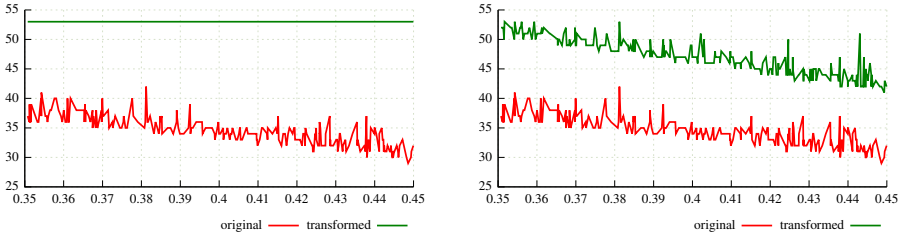
As the numerical accuracy, execution time is a critical matter for programs, especially in embedded systems where less execution time means less energy consumption or better reactivity. Taking into account both accuracy and execution time simultaneously is a difficult problem because these two criteria do not cohabit well: improving accuracy may be costly (execution-time improvement can impact accuracy as well). Our proposed solution is to allow tradeoffs between accuracy and time [4].

We present the SyHD software developed to perform source-to-source transformation improving accuracy without impacting execution time too much. SyHD synthesizes C source code for both accuracy and execution-time criteria. It uses compensation for improving accuracy and transformation strategies for reducing the impact of this improvement on execution time. The automatic compensation is provided by another software, CoHD [8]. Compensated algorithms are efficient but difficult to implement, so we automate this process with CoHD to benefit from a good execution time compared to other methods. Experimental results show that our automatically transformed programs have roughly the same accuracy and execution time than the existing compensated ones when the latter exist. To improve execution time, we have implemented several strategies of partial compensation. SyHD synthesizes a new program by automatically analyzing a set of transformed programs implementing these strategies for a given environment (defined by a target, some data, as well as some accuracy and

*Abstract of a manuscript in preparation.

execution-time constraints). Two kinds of transformation strategies are proposed: the ones which compensate the floating-point computations that generate the largest errors, and the ones which do loop transformations, such as loop fission [1].

We apply this approach to a significant set of examples, ranging from recursive summation, to polynomial evaluations, to iterative refinement for linear system solving. We successfully generate programs with accuracy and execution-time tradeoffs in a reasonable amount of time [7].



The following example illustrates the potential of our approach (results are automatically obtained by CoHD and SyHD). The figure above presents the number of significant bits when evaluating the polynomial $p(x) = (x - 0.75)^5(x - 1)^{11}$ with the Horner's scheme, where $x \in [0.35, 0.45]$. The leftmost part shows the result of the original and transformed programs when no tradeoff is allowed. We recover the full accuracy (53 bits for double precision) for approximately 300 cycles (more than 3 times better than an algorithm using double-double expansions). The rightmost part shows same results when tradeoff between accuracy and execution time is required. By sacrificing accuracy we can save here 50 more cycles for this polynomial evaluation.

References

[1] A. W. Appel. *Modern Compiler Implementation: In ML*. Cambridge University Press, New York, NY, USA, 1998.

[2] T. J. Dekker. A floating-point technique for extending the available precision. *Numer. Math.*, 18:224–242, 1971.

[3] S. Graillat, P. Langlois, and N. Louvet. Algorithms for accurate, validated and fast polynomial evaluation. *Japan Journal of Industrial and Applied Mathematics*, 26(2-3):191–214, 2009.

[4] P. Langlois, M. Martel, and L. Thévenoux. Automatic code transformation to optimize accuracy and speed in floating-point arithmetic. In *SCAN 2012 – 15th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2012.

[5] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Springer Disc. & Comp. Geometry*, 18(3):305–363, 1997.

[6] IEEE Computer Society. *IEEE 754 Standard for Floating-Point Arithmetic*. The Institute of Electrical and Electronics Engineers, Inc., 2008.

[7] L. Thévenoux. *Synthèse de code avec compromis entre performance et précision en arithmétique flottante IEEE 754*. PhD thesis, Université de Perpignan Via Domitia, Perpignan, France, July 2014. URL: `https://tel.archives-ouvertes.fr/tel-01143824`.

[8] L. Thévenoux, P. Langlois, and M. Martel. Automatic source-to-source error compensation of floating-point programs. Submitted to the 18th IEEE International Conference on Computational Science and Engineering (CSE 2015), Porto, Portugal., May 2015. URL: `https://hal.archives-ouvertes.fr/hal-01158399`.