Sound Approximation of Programs with Elementary Functions

Eva Darulova[1(✉)] and Anastasia Volkova[2]

[1] MPI-SWS, Saarland Informatics Campus, Saarbrücken, Germany
eva@mpi-sws.org
[2] Inria, Lyon, France
anastasia.volkova@inria.fr

Abstract. Elementary function calls are a common feature in numerical programs. While their implementations in mathematical libraries are highly optimized, function evaluation is nonetheless very expensive compared to plain arithmetic. Full accuracy is, however, not always needed. Unlike arithmetic, where the performance difference between for example single and double precision floating-point arithmetic is relatively small, elementary function calls provide a much richer tradeoff space between accuracy and efficiency. Navigating this space is challenging, as guaranteeing the accuracy and choosing correct parameters for good performance of approximations is highly nontrivial. We present a fully automated approach and a tool which approximates elementary function calls inside small programs while guaranteeing overall user given error bounds. Our tool leverages existing techniques for roundoff error computation and approximation of individual elementary function calls and provides an automated methodology for the exploration of parameter space. Our experiments show that significant efficiency improvements are possible in exchange for reduced, but guaranteed, accuracy.

1 Introduction

Numerical programs face an inherent tradeoff between accuracy and efficiency. Choosing a larger finite precision provides higher accuracy, but is generally more costly in terms of memory and running time. Not all applications, however, need a very high accuracy to work correctly. We would thus like to compute the results with only as much accuracy as is needed, in order to save resources.

Navigating this tradeoff between accuracy and efficiency is challenging. First, estimating the accuracy, i.e. bounding roundoff and approximation errors, is nontrivial due to the complex nature of finite-precision arithmetic which inevitably occurs in numerical programs. Second, the space of possible implementations is usually prohibitively large and thus cannot be explored manually.

Today, users can choose between different automated tools for analyzing accuracy of floating-point programs [7,8,11,14,18,20,26] as well as for choosing between different precisions [5,6,10]. The latter tools perform mixed-precision tuning, i.e. they assign different floating-point precisions to different operations,

I. Dillig and S. Tasiran (Eds.): CAV 2019, LNCS 11562, pp. 174–183, 2019.
https://doi.org/10.1007/978-3-030-25543-5_11

and can thus improve the performance w.r.t. a uniform precision implementation. The success of such an optimization is, however, limited to the case when uniform precision is just barely not enough to satisfy a given accuracy specification.

Another possible target for performance optimizations are elementary functions (e.g. `sin, exp`). Users by default choose single- or double-precision `libm` library function implementations, which are fully specified in the C language standard (ISO/IEC 9899:2011) and provide high accuracy. Such implementations are, however, expensive. When high accuracy is not needed, we can save significant resources by replacing `libm` calls by coarser approximations, opening up a larger, and different tradeoff space than mixed-precision tuning. Unfortunately, existing automated approaches [1,25] do not provide accuracy guarantees.

On the other hand, tools like Metalibm [3] approximate *individual* elementary functions by polynomials with rigorous accuracy guarantees given by the user. They, however, do not consider entire programs and leave the selection of its parameters to the user, limiting its usability mostly to experts.

We present an approach and a tool which leverages the existing whole-program error analysis of Daisy [8] and Metalibm's elementary function approximation to provide both *sound whole-program guarantees* as well as *efficient* C implementations for floating-point programs with elementary function calls. Given a target error specification, our tool automatically distributes the error budget among uniform single or double precision arithmetic operations and elementary functions, and selects a suitable polynomial degree for their approximation.

We have implemented our approach inside the tool Daisy and compare the performance of generated programs against programs using `libm` on examples from literature. The benchmarks spend on average 38% and up to 50% of time for evaluation of the elementary functions. Our tool improves the overall performance by on average 14% and up to 25% when approximating each elementary function call individually, and on average 17% and up to 31% when approximating compound function calls. These improvements were achieved solely by optimizing approximations to elementary functions and illustrate pertinence of our approach. These performance improvements incur overall whole-program errors which are only 2–3 magnitudes larger than double-precision implementations using `libm` functions and are well below the errors of single-precision implementations. Our tool thus allows to effectively trade performance for larger, but guaranteed, error bounds.

Contributions. In summary, in this paper we present: (1) the first approximation technique for elementary functions with sound whole-program error guarantees, (2) an experimental evaluation on benchmarks from literature, and (3) an implementation, which is available at https://github.com/malyzajko/daisy.

Related Work. Several static analysis tools bound roundoff errors of floating-point computations [7,18,20,26], assuming `libm` implementations, or verify the correctness of several functions in Intel's `libm` library [17]. Muller [21] provides

a good overview of the approximation of elementary functions. Approaches for improving the performance of numerical programs include mixed-precision tuning [5,6,10,16,24], and autotuning, which performs low-level real-value semantics-preserving transformations [23,27]. These leverage a different part of the trade-off space than `libm` approximation and are thus orthogonal. Herbie [22] and Sardana [7] improve accuracy by rewriting the non-associative finite-precision arithmetic, which is complementary to our approach. Approaches which approximate entire numerical programs include MCMC search [25], enumerative program synthesis [1] and neural approximations [13]. Accuracy is only checked on a small set of sample inputs and is thus not guaranteed.

2 Our Approach

We explain our approach using the following example [28] computing a forward kinematics equation and written in Daisy's real-valued specification language:

```
def forwardk2jY(theta1: Real, theta2: Real): Real = {
  require(-3.14 <= theta1 && theta1 <= 3.14 && -3.14 <= theta2 && theta2 <= 3.14)
    val l1: Real = 0.5; val l2: Real = 2.5
    l1 * sin(theta1) + l2 * sin(theta1 + theta2)
} ensuring(res => res +/- 1e-11)
```

Although this program is relatively simple, it still presents an opportunity for performance savings, especially when it is called often, e.g. during the motion of a robotics arm. Assuming double-precision floating-point arithmetic and library implementations for sine, Daisy's static analysis determines the worst-case absolute roundoff error of the result to be 3.44e-15. This is clearly a much smaller error than what the user requested (1e-11) in the postcondition (`ensuring` clause).

The two elementary function calls to `sin` account for roughly 40.7% of the overall running time. We can save some of this running time using polynomial approximations, which our tool generates in less than 6 min. The new double precision C implementation is roughly 15.6% faster than one with `libm`[1] functions, i.e. using around 40% of the available margin. This is a noteworthy performance improvement, considering that we optimized uniquely the evaluation of elementary functions. The actual error of the approximate implementation is 1.56e-12, i.e. roughly three orders of magnitude higher than the `libm` error. This error is still much smaller than if we had used a uniform single precision implementation, which incurs a total error of 1.85e-6.

We implement our approach inside the Daisy framework [8], combining Daisy's static dataflow analysis for bounding finite-precision roundoff errors, Metalibm's automated generation of efficient polynomial approximations, as well as a novel error distribution algorithm. Our tool furthermore automatically selects a suitable polynomial degree for approximations to elementary functions.

[1] There are various different implementations of `libm` that depend on the operating system and programming language. Here when referring to `libm` we mean the GNU `libc` implementation (https://www.gnu.org/software/libc/).

Unlike previous work, our tool *guarantees* that the user-specified error is satisfied. It soundly distributes the overall error budget among arithmetic operations and `libm` calls using Daisy's static analysis. Metalibm uses the state-of-the art minimax polynomial approximation algorithm [2] and Sollya [4] and Gappa [12] to bound errors of their implementations. Given a function, a target relative error bound and implementation parameters, Metalibm generates C code. Our tool does not guarantee to find the most efficient implementation; the search space of implementation and approximation choices is highly complex and discrete, and it is thus infeasible to find the optimal parameters.

The input to our tool is a straight-line program[2] with standard arithmetic operators $(=, -, *, /)$ as well as the most commonly used elementary functions $(\sin, \cos, \tan, \log, \exp, \sqrt{})$. The user further specifies the domains of all inputs, together with a target overall absolute error which must be satisfied. The output is C code with arithmetic operations in uniform single or double precision, and `libm` approximations in double precision (Metalibm's only supported precision).

Algorithm. We will use 'program' for the entire expression, and 'function' for individual elementary functions. Our approach works in the following steps.

Step 1 We re-use Daisy's frontend which parses the input specification. We add a pre-processing step, which decomposes the abstract syntax tree (AST) of the program we want to approximate such that each elementary function call is assigned to a fresh local variable. This transformation eases the later replacement of the elementary functions with an approximation.

Step 2 We use Daisy's roundoff error analysis on the entire program, assuming a `libm` implementation of elementary functions. This analysis computes a real-valued range and a worst-case absolute roundoff error bound for each subexpression in the AST, assuming uniform single or double precision as appropriate. We use this information in the next step to distribute the error and to determine the parameters for Metalibm for each function call.

Step 3 This is the core step, which calls Metalibm to generate a (piecewise) polynomial approximation for each elementary function which was assigned to a local variable. Each call to Metalibm specifies the local target error for each function call, the polynomial degree and the domain of the function call arguments. To determine the argument domains, we use the range and error information obtained in the previous step. Our tool tries different polynomial degrees and selects the fastest implementation. We explain our error distribution and polynomial selection further below.

Metalibm generates efficient double-precision C code including argument reduction (if applicable), domain splitting, and polynomial approximation with a guaranteed error below the specified target error (or returns an error). Metalibm furthermore supports approximations with lookup tables, whose size the user can control manually via our tool frontend as well.

[2] All existing approaches for analysing floating-point roundoff errors which handle loops or conditional branches, reduce the reasoning about errors to straight-line code, e.g. through loop invariants [9,14] or loop unrolling [7], or path-wise analysis [7,9,15].

Step 4 Our tool performs roundoff error analysis again, this time taking into account the new approximations' precise error bounds reported by Metalibm. Finally, Daisy generates C code for the program itself, as well as all necessary headers to link with the approximation generated by Metalibm.

Error Distribution. In order to call Metalibm, Daisy needs to determine the target error for each `libm` call. Recall that the user of our tool only specifies the *total* error at the end of the program. Hence, distributing the total error budget among arithmetic operations and (potentially several) elementary function calls is a crucial step. Consider again our running example which has two elementary function calls. Our tool distributes the error budget as follows:

$$|f(x) - \tilde{f}(\tilde{x})| \leq |f(x) - \hat{f}_1(x)| + |\hat{f}_1(x) - \hat{f}_2(x)| + |\hat{f}_2(x) - \tilde{f}(\tilde{x})|$$

where we denote by f the real-valued specification of the program; $\hat{f}_1$ and $\hat{f}_2$ have one and two elementary function calls approximated, respectively, and arithmetic is considered exact; and $\tilde{f}$ is the final finite-precision implementation.

Daisy first determines the budget for the finite-precision roundoff error ($|\hat{f}_2(x) - \tilde{f}(\tilde{x})|$) and then distributes the remaining part among `libm` calls. At this point, Daisy cannot compute $|\hat{f}_2(x) - \tilde{f}(\tilde{x})|$ exactly, as the approximations are not available yet. Instead, it assumes `libm`-based approximations as baseline.

Then, Daisy distributes the remaining error budget either equally among the elementary function calls, or by taking into account that the approximation errors are propagated differently through the program. This error propagation is estimated by computing the derivative w.r.t. to each elementary function call (which gives an estimation of the conditional number). Daisy computes partial derivatives symbolically and maximizes them over the specified input domain.

Finally, we obtain an error budget for each `libm` call, representing the total error due to the elementary function call *at the end of the program.* For calling Metalibm, however, we need the *local* error at the function call site. Due to error propagation, these two errors can differ significantly, and may lead to overall errors which exceed the error bound specified by the user. We estimate the error propagation using a linear approximation based on derivatives, and use this estimate to compute a *local* target error from the total error budget.

Since Metalibm usually generates approximations with slightly tighter error bounds than asked for, our tool performs a second roundoff analysis (step 4), where all errors (smaller or larger) are correctly taken into account.

Polynomial Degree Selection. The polynomial degree significantly and in a discrete way influences the efficiency of approximations, so that optimal prediction is infeasible. Hence, our tool performs a linear search, using the (coarse) estimated running time reported by Metalibm (obtained with a few benchmarking runs) to select the approximation with the smallest estimated running time. The search stops either when the estimated running time is significantly higher than the current best, or when Metalibm times out.

We do not automatically exploit other Metalibm's parameters, such as minimum subdomain width for splitting, since they give fine-grained control that is not suitable for *general* automatic implementations.

3 Experimental Evaluation

We evaluate our approach in terms of accuracy and performance on benchmarks from literature [9,19,28] which include elementary function calls, and extend them with the examples rodriguesRotation[3] and ex2* and ex3_d, which are problems from a graduate analysis textbook. While they are relatively short, they represent important kernels usually employing several elementary function calls[4]. We base target error bounds on roundoff errors of a `libm` implementation: middle and large errors, each of which is roughly three and four orders of magnitudes larger than the `libm`-based bound, respectively. By default, we assume double 64 bit precision.

Our tool provides an automatic generation of benchmarking code for each input program. Each benchmarking executable runs the Daisy-generated code on 10^7 random inputs from the input domain and measures performance in the number of processor clock cycles. Of the measured number of cycles we discard the highest 10%, as we have observed these to be outliers.

Experimental Results. By default, we approximate individual elementary function calls separately, use equal error distribution and allow table-based approximations with an 8-bit table index. For large errors we also measure performance for: (i) default settings but with the derivative-based errors distribution; (ii) default settings but without table usage; (iii) default settings but with compound calls with depth 1 and depth ∞ (approximation 'as much as possible').

Table 1 shows the performance improvements of approximated code w.r.t. `libm` based implementations of our benchmarks. We compare against `libm` only, as no approximation or synthesis tool provides error guarantees. By removing `libm` calls in initial programs we roughly estimate the elementary function overhead (second column) and give an idea for the margin of improvement. Figure 1 illustrates the overall improvement that we obtain for each benchmark (the height of the bars) and the relative distribution of the running time between arithmetic (blue) and elementary functions (green), for large errors with default settings but approximate compound calls with depth = ∞.

Our tool generates code with significant performance improvements for most functions and often reduces the time spent for the evaluation of elementary functions by a factor of two. As expected, the improvements are overall better for larger errors and vary on average from 10.7% to 13.8% for individual calls depending on the settings, and reach 17.1% on average when approximating compound calls as much as possible. However, increasing the program target error (for equal error distributions Metalibm target error increases linearly with it) does not necessarily lead to better performance, e.g. in case of axisRotationY and rodriguesRotation. This is the result of discrete decisions concerning the approximation degrees and the domain splittings inside Metalibm.

[3] https://en.wikipedia.org/wiki/Rodrigues27_rotation_formula.

[4] Experiments are performed on a Debian Linux 9 Desktop machine with a 3.3 GHz Intel i5 processor and 16 GB of RAM. All code for benchmarking is compiled with GNUs g++, version 6.3.0, with the `-O2` flag.

Table 1. Performance improvements (in percent) of approximated code w.r.t. a program with `libm` library function calls.

precision	double							single
	elem. func.	middle	large errors					middle
benchmark	overhead	equal	equal	deriv	no table	depth 1	depth ∞	equal
sinxx10	20.8	7.6	7.7	7.7	7.7	7.6	7.7	4.7
xu1	49.3	13.9	25.8	18.0	26.6	25.7	27.3	8.1
xu2	53.6	4.6	12.4	13.0	12.6	12.5	26.0	-1.4
integrate18257	52.8	15.2	19.4	15.1	-4.5	22.4	31.7	2.1
integStoutemyer	42.1	-1.0	6.5	1.4	0.4	4.8	21.9	6.4
axisRotationX	38.0	17.2	17.3	18.1	17.4	17.6	17.3	-10.5
axisRotationY	37.9	17.6	12.8	21.5	12.9	12.8	12.8	-14.1
rodriguesRotation	28.9	14.9	11.6	13.6	13.8	13.8	13.9	-7.6
pendulum1	24.4	-4.6	-2.9	-4.3	-4.2	11.0	11.7	-9.7
pendulum2	50.3	9.6	11.4	6.2	-0.8	20.2	20.5	-0.5
forwardk2jX	43.7	15.1	15.4	15.5	15.0	15.0	15.0	-10.2
forwardk2jY	40.7	10.7	15.6	15.6	15.6	15.6	15.6	7.4
ex2_1	34.6	12.8	12.8	12.3	12.3	12.3	12.1	8.4
ex2_2	34.9	5.9	14.8	15.4	15.1	15.0	15.3	3.6
ex2_3	42.1	23.5	24.5	24.5	24.1	24.8	24.3	3.9
ex2_4	31.8	11.9	12.5	12.5	12.6	14.3	14.3	7.9
ex2_5	40.6	22.5	24.4	24.5	24.4	24.4	24.3	10.2
ex2_9	35.0	7.2	7.1	7.4	7.2	7.0	9.4	-10.1
ex2_10	41.5	20.6	21.7	8.9	20.5	21.3	21.4	8.3
ex2_11	30.9	-6.8	-2.3	-4.9	-2.4	-4.8	-2.8	17.9
ex3_d	39.3	10.3	20.9	19.9	-1.1	19.9	20.3	4.9
average	38.7	10.9	13.8	12.5	10.7	14.9	17.1	1.4

Somewhat surprisingly, we did not observe an advantage of using the derivative-based error distribution over the equal one. We suspect that is due to the nonlinear nature of Metalibm's heuristics.

Table 1 further demonstrates that usage of tables generally improves the performance. However, the influence of increasing the table size must be studied on a case-by-case basis since large tables might lead to memory-bound computations.

We observe that it is generally beneficial to approximate 'as much as possible'. Indeed, the power of Metalibm lies in generating (piece-wise) polynomial approximations of compound expressions, whose behavior might be much simpler to evaluate than its individual subexpressions.

Finally, we also considered an implementation where all data and arithmetic operations are in single precision apart from the double-precision Metalibm-generated code (whose output is accurate only to single precision). We observe that slight performance improvements are possible, i.e. Metalibm can compete even with single-precision libm-based code, but to achieve performance improvements comparable to those of double-precision code, we need a single-precision code generation from Metalibm.

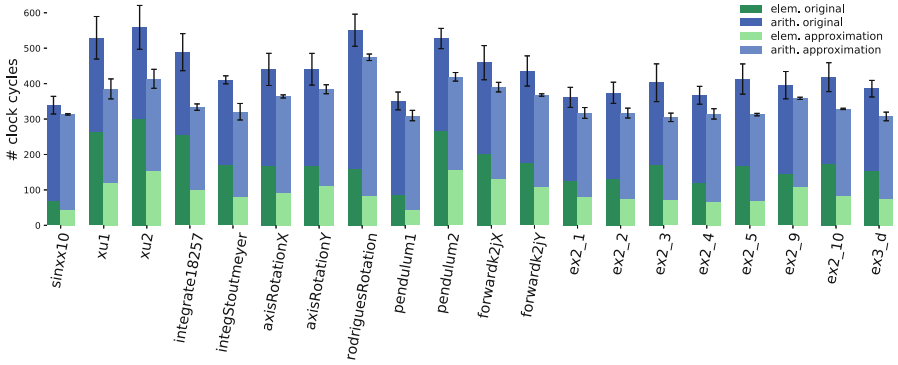

Fig. 1. Average performance and standard deviation. For each benchmark, the first bar shows the running time of the libm-based implementation and the second one of our implementation. Even relatively small overall time improvements are significant w.r.t. the time portion we can optimize (in green). Our implementations also have significantly smaller standard deviation (black bars). (Color figure online)

Analysis Time. Analysis time is highly dependent on the number of required approximations of elementary functions: each approximation requires a separate call to Metalibm whose running time in turn depends on the problem definition. Daisy reduces the number of calls to Metalibm by common expression elimination which improves the analysis time. Currently, we set the timeout for each Metalibm call to 3 min, which leads to an overall analysis time which is reasonable. Overall, our tool takes between 15 s and 20 min to approximate whole programs, with the average running time being 4 min 40 s per program.

4 Conclusion

We presented a fully automated approach which improves the performance of small numerical kernels at the expense of some accuracy by generating custom approximations of elementary functions. Our tool is parametrized by a user-given whole-program absolute error bound which is guaranteed to be satisfied by the generated code. Experiments illustrate that the tool efficiently uses the available margin for improvement and provides significant speedups for double-precision implementations. This work provides a solid foundation for future research in the areas of automatic approximations of single-precision and multivariate functions.

Acknowledgments. The authors thank Christoph Lauter for useful discussions and Youcef Merah for the work on an early prototype.

References

1. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing synthesis with metasketches. In: POPL (2016)

2. Brisebarre, N., Chevillard, S.: Efficient polynomial L-approximations. In: ARITH (2007)
3. Brunie, N., de Dinechin, F., Kupriianova, O., Lauter, C.: Code generators for mathematical functions. In: ARITH (2015)
4. Chevillard, S., Joldeş, M., Lauter, C.: Sollya: an environment for the development of numerical codes. In: Fukuda, K., Hoeven, J., Joswig, M., Takayama, N. (eds.) ICMS 2010. LNCS, vol. 6327, pp. 28–31. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15582-6_5
5. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous floating-point mixed-precision tuning. In: POPL (2017)
6. Damouche, N., Martel, M.: Mixed precision tuning with salsa. In: PECCS, pp. 185–194. SciTePress (2018)
7. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. Int. J. Softw. Tools Technol. Transfer **19**(4), 427–448 (2017)
8. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - framework for analysis and optimization of numerical programs (tool paper). In: Beyer, D., Huisman, M. (eds.) TACAS 2018. LNCS, vol. 10805, pp. 270–287. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89960-2_15
9. Darulova, E., Kuncak, V.: Towards a compiler for reals. ACM TOPLAS **39**(2), 8 (2017)
10. Darulova, E., Sharma, S., Horn, E.: Sound mixed-precision optimization with rewriting. In: ICCPS (2018)
11. De Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted verification of elementary functions using Gappa. In: ACM Symposium on Applied Computing (2006)
12. de Dinechin, F., Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. Comput. **60**(2), 242–253 (2011)
13. Esmaeilzadeh, H., Sampson, A., Ceze, L., Burger, D.: Neural acceleration for general-purpose approximate programs. In: IEEE/ACM International Symposium on Microarchitecture (2012)
14. Goubault, E., Putot, S.: Static analysis of finite precision computations. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 232–247. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_17
15. Goubault, E., Putot, S.: Robustness analysis of finite precision implementations. In: Shan, C. (ed.) APLAS 2013. LNCS, vol. 8301, pp. 50–57. Springer, Cham (2013). https://doi.org/10.1007/978-3-319-03542-0_4
16. Lam, M.O., Hollingsworth, J.K., de Supinski, B.R., Legendre, M.P.: Automatically adapting programs for mixed-precision floating-point computation. In: ICS (2013)
17. Lee, W., Sharma, R., Aiken, A.: On automatically proving the correctness of math.h implementations. In: POPL (2018)
18. Magron, V., Constantinides, G., Donaldson, A.: Certified roundoff error bounds using semidefinite programming. ACM Trans. Math. Softw. **43**(4), 34 (2017)
19. Merlet, J.P.: The COPRIN examples page. http://www-sop.inria.fr/coprin/logiciels/ALIAS/Benches/
20. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.A.: Automatic estimation of verified floating-point round-off errors via static analysis. In: Tonetta, S., Schoitsch, E., Bitsch, F. (eds.) SAFECOMP 2017. LNCS, vol. 10488, pp. 213–229. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66266-4_14
21. Muller, J.M.: Elementary Functions - Algorithms and Implementation, 3rd edn. Birkhäuser, Basel (2016)

22. Panchekha, P., Sanchez-Stern, A., Wilcox, J.R., Tatlock, Z.: Automatically improving accuracy for floating point expressions. In: PLDI (2015)
23. Püschel, M., et al.: Spiral - a generator for platform-adapted libraries of signal processing alogorithms. IJHPCA **18**(1), 21–45 (2004)
24. Rubio-González, C., et al.: Precimonious: tuning assistant for floating-point precision. In: SC (2013)
25. Schkufza, E., Sharma, R., Aiken, A.: Stochastic optimization of floating-point programs with tunable precision. In: PLDI (2014)
26. Solovyev, A., Jacobsen, C., Rakamarić, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with Symbolic Taylor Expansions. In: Bjørner, N., de Boer, F. (eds.) FM 2015. LNCS, vol. 9109, pp. 532–550. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-19249-9_33
27. Vuduc, R., Demmel, J.W., Bilmes, J.A.: Statistical models for empirical search-based performance tuning. Int. J. High Perform. Comput. Appl. **18**(1), 65–94 (2004)
28. Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., Lotfi-Kamran, P.: AxBench: a multiplatform benchmark suite for approximate computing. IEEE Des. Test **34**(2), 60–68 (2017)