Synthesizing Efficient Low-Precision Kernels

Anastasiia Izycheva[1], Eva Darulova[2], and Helmut Seidl[1]

[1] Fakultät für Informatik, TU München, `izycheva@in.tum.de,seidl@in.tum.de`
[2] MPI-SWS, Saarland Informatics Campus, `eva@mpi-sws.org`

Abstract. In this paper, we present a *fully automated* approach for synthesizing fast numerical kernels with *guaranteed* error bounds. The kernels we target contain elementary functions such as sine and logarithm, which are widely used in scientific computing, embedded as well as machine-learning programs. However, standard library implementations of these functions are often overly accurate and therefore unnecessarily expensive. Our approach trades superfluous accuracy against performance by approximating elementary function calls by polynomials and by implementing arithmetic operations in low-precision fixed-point arithmetic. Our algorithm soundly distributes and guarantees an overall error budget specified by the user. The evaluation on benchmarks from different domains shows significant performance improvements of 2.23x on average compared to state-of-the-art implementations of such kernel functions.

Keywords: Synthesis, Approximation, Finite Precision, Elementary Functions

1 Introduction

Automated program synthesis promises to simplify and speed up common programming tasks: the programmer only needs to write a specification of what should be computed and a synthesis tool automatically generates an implementation, whose correctness is guaranteed by construction. Today's techniques generate programs over algebraic data structures [24], APIs [19], string manipulation [22], probabilistic programs [39], linear arithmetic computations [25], etc.

Many of these synthesis techniques explicitly or implicitly generate programs optimized for a particular metric. For instance, enumerative techniques often choose the program of smallest size which satisfies the specification [3], or use an explicit static cost model to prune inefficient programs [32].

An additional way to improve the efficiency of a program is to introduce approximations. We can leverage the fact that many applications are tolerant to a certain amount of error or noise and thus need not compute their results exactly, but only as accurately as necessary [46]. Approximations introduce a tradeoff between accuracy and efficiency, which is in general challenging to navigate. In order to determine whether a particular approximation is suitable, we need to be able to verify the overall program accuracy. Furthermore, the space of possible approximations is in general prohibitively large, making manual program optimizations tedious, error prone, and inefficient.

Most of today's synthesis techniques consider only exact specifications, and are thus unsuitable to generate programs which are only correct up to some error bound [24,19,22,39,25,3,32]. Furthermore, most techniques do not explicitly optimize for efficiency of the generated program. A few approaches exist which do generate approximate programs, but which check correctness via testing on a few sample inputs and thus cannot provide accuracy guarantees [18,42].

We propose a fully automated synthesis approach for the domain of numerical kernels, which approximates both arithmetic operations as well as elementary functions, and which provides sound end-to-end accuracy guarantees. The user specifies an ideal, real-valued program together with a maximum error bound. Our technique generates an efficient finite-precision implementation with polynomial approximations of elementary functions.

The numerical kernels we handle cover widely used applications in various domains: for instance, embedded control to compute rotations of robotic components, scientific computing simulations to determine the state of a periodic event, and machine learning models with sigmoid activation functions. By default, programmers today implement the elementary functions in these kernels using library implementations. While these are convenient and optimized, they usually provide only a limited set of accuracies, e.g. single and double floating-point precision, severely limiting optimization opportunities.

We furthermore target fixed-point arithmetic implementations, which allow arbitrary bit-widths for individual operations and thus provide high flexibility in the accuracy-efficiency tradeoff space when executed on accelerators such as FPGAs. Compared to an implementation in floating-point arithmetic, this choice, however, increases the search space significantly and thus makes code synthesis more challenging.

In this paper, we present our synthesis algorithm, which distributes the overall error budget provided by the user among different operations in the kernel, and soundly takes care of the error propagation as well as finite-precision roundoff errors. To generate approximations, our algorithm extends and combines an existing polynomial approximation technique [27] and roundoff error analysis [14].

We implemented our algorithm and evaluate it on several embedded, scientific computing and machine learning kernels. Compared to implementations using default library functions, our synthesized programs on average need 2.23x less machine cycles to execute.

Contributions To summarize, this paper presents: a) the first sound and automated synthesis algorithm for efficient numerical kernels with both arithmetic and elementary function approximations, b) an experimental evaluation using existing and several new benchmarks, c) a prototype implementation of the synthesis algorithm, which we release as open-source: `https://github.com/malyzajko/daisy/tree/approx`.

2 Background

Fixed-Point Arithmetic Floating-point arithmetic [23] is a common and convenient choice to implement an approximation of real numbers on today's discrete computers. However, floating-point units usually support only a limited set of precisions (e.g. 16, 32, and 64 bit), and the exponent always occupies a fixed number of bits. For applications which operate over limited ranges, many of these bits remain unused, effectively wasting resources such as energy or time.

Fixed-point arithmetic allows a programmer to implement a computation in purely integer arithmetic (i.e. without special hardware) and with exactly as many bits for the exponent and precision as are actually needed. On an accelerator such as an FPGA, fixed-point arithmetic thus provides a resource-efficient implementation.

To compile a fixed-point arithmetic program (e.g. with a high-level synthesis tool such as Xilinx' Vivado [45]), the programmer has to select suitable fixed-point *formats* for a program's inputs and operations. A fixed-point format specifies at least the total wordlength W and the number of integer bits I. The latter have to be chosen such that no overflow can occur. The remaining $F = W - I$ bits are used to represent the fractional part of a number and determine the accuracy of the computation. A larger F makes the computation more accurate, but also more costly. Choosing a suitable tradeoff is challenging to do manually a) due to the large number of format options—we can choose different and arbitrary wordlengths for individual operations—and b) due to the need to verify the overall accuracy of the computation.

As part of the fixed-point format, usually one can also choose the rounding mode and the overflow behaviour. In this work, we consider the default truncation as the rounding mode. We also leave the overflow mode to default (wrap around), but note that our analysis guarantees that no overflow can occur.

Elementary Functions For both floating-point and fixed-point arithmetic, elementary functions are supported via library functions. Here, we focus on the implementation and specification provided and used by Xilinx Vivado, which we use in our experiments and which is widely used in industry. We note, however, that our approach is not tied to a particular choice of fixed-point compiler.

Xilinx Vivado supports 32 bit fixed-point implementations for sine and cosine, and 8 or 16 bit versions of the exponential function. The compiler further supports automated conversion to floating-point arithmetic, such that floating-point implementations of elementary function calls can be used within fixed-point arithmetic programs. These are provided for precisions 16, 32, and 64 bit. Thus, while some support for elementary functions is provided, it is only available for a small variety of precisions, effectively limiting optimization options.

Fixed-point Roundoff Error Analysis and Precision Assignment Current static analysis tools [14,21,17,11,15] provide automated dataflow analyses which compute finite-precision roundoff errors w.r.t. to a real-valued semantics using the interval [35] and affine [20] arithmetic abstract domains. These analyses are applicable to both floating-point as well as fixed-point arithmetic, whereas other tools use a different, global optimization-based approach and only support floating-point arithmetic [44,33,36]. Several of these tools can analyze programs with elementary function calls, but always assume library implementations. Reasoning about loops and conditionals is always reduced to reasoning about straight-line code, e.g. through loop unrolling [11], special loop invariants [15,21,36], or path-by-path analysis of conditionals [36,15]. In this work we thus focus on the core issue and consider straight-line programs only.

Because of the nature of fixed-point arithmetic, the error specification is fundamentally absolute: the fixed-point format for each operation is fixed at compile time, thus the worst-case error is the same no matter what the magnitude of the value actually is at runtime. In this work we thus consider absolute errors.

Several tools perform mixed-precision tuning [9,16,10], which assigns different precisions to individual operations. Due to the large number of possible precision assignments, the search for such an assignment is necessarily incomplete. These approaches only consider arithmetic operations, i.e. no elementary functions, and only Daisy [16] supports fixed-point arithmetic.

Polynomial Approximation in Metalibm Polynomials are a common choice for approximating complex functions. The approximation accuracy largely depends on the degree of the polynomial, larger degrees incurring a higher execution cost. Given an elementary function, an input domain and a target error which the approximation has to satisfy, the recent tool Metalibm [27] selects a suitable degree fully automatically. It employs Remez' algorithm [37], which guarantees the best possible polynomial approximation. It additionally performs domain splitting [26], which allows different polynomials and degrees to be used on different parts of the input domain, and supports a number of further features such as generation of tables for table lookup and range reduction.

Metalibm currently generates double floating-point C implementations which can outperform highly optimized library implementations [8]. It can be applied to individual or compound elementary functions as long as they are univariate (Remez' algorithm only supports univariate functions), though it usually times out after several hours on more complex compound functions. To summarize, Metalibm generates efficient individual floating-point approximations, but cannot be applied to entire programs and it does not support fixed-point arithmetic.

3 Our Synthesis Algorithm

In this section, we present our approach for synthesizing approximate programs with error guarantees for straight-line input programs with elementary function calls. We do not consider loops or branches, but note that our approach can

```
def xu1(x1: Real, x2: Real): Real = {
  require(0.01 <= x1 && x1 <= 0.75 && 0.01 <= x2 && x2 <= 1.5)
   2 * sin(x1) + 0.8 * cos(2 * x1) + 7 * sin(x2) - x1
} ensuring(res => res +/- 4.24e-06)
```

Fig. 1. Example input program with elementary function calls

be combined with previous techniques which reduce reasoning about loops and conditionals to straight-line code [11,15,21,36]. [3]

To illustrate our approach, Figure 1 shows an example synthesis specification of a program, which is taken from the benchmark set of the CORPIN project [1]. It consists of: a program with three elementary function calls `sin(x1)`, `cos(2*x1)` and `sin(x2)`, input ranges for variables in the `require` clause, and the maximum tolerated absolute error for the program in the `ensuring` clause: 4.24e-6.

Given this specification, our goal is to automatically synthesize a program which approximates the expensive elementary function calls and implements the arithmetic operations in a suitable fixed-point precision, while respecting the specified error bound.

The specified maximum tolerated error for the program can be seen as a budget, which has to be distributed between all the different sources of errors in the program, namely the elementary function approximations as well as the finite-precision arithmetic. Note that the approximation polynomials themselves have to be implemented in fixed-point arithmetic as well. In our example we thus need to assign a roundoff error budget to the four multiplications, two additions and one subtraction of the top-level program, as well as to the yet unknown polynomial approximations of `sin(x1)`, `cos(2*x1)` and `sin(x2)`.

Thus, in order to synthesize an approximate program which satisfies the specified error bound, we need to:

1. distribute the error budget, specified for the whole program, between arithmetic operations in the top-level function, potentially multiple elementary function calls, and the finite-precision implementation of the polynomials,
2. find a (piecewise-) polynomial approximation for every elementary function which stays within limits of the assigned approximation error budget, and
3. assign a finite precision to each arithmetic operation of the top-level function, as well as the polynomial approximations.

Each of the above challenges involves finding a solution in a large search space, and the search is furthermore complicated by the fact that individual errors interact in nonlinear and discrete ways. Every error introduced at one point

[3] These techniques are applied to underlying roundoff error analysis, and our approach can be combined with any sound roundoff error analysis. Therefore, the application domain for our technique is only limited by what a roundoff error analysis can handle. For programs with discrete decisions – like machine-learning classifiers – the effect of the approximations on decision errors can be obtained experimentally, or to a limited extent via static analysis [31].

```
Input: $\mathbb{S}$ - all variables, arithmetic operations and elementary function calls;
$s_{ef}$ - elementary function calls; $\epsilon_g$ - global error budget

1. $\forall s \in \mathbb{S}$ assign precision $p_s$ wrt. cost of $s$ and $\epsilon_g$
2. Based on $p_s$ assign local budget $\epsilon_i$ to all $s_{ef}$
3. $\forall s_{ef}$ and $k.0 \leq k \leq 5$ REPEAT:
    – Split $\epsilon_i$ into $\epsilon_{i\_approx}$ and $\epsilon_{i\_fp}$, for $k = 0$ $\epsilon_{i\_approx} = \epsilon_{i\_fp}$, $k \geq 1$:
      $\epsilon_{i\_fp} = \epsilon_{i\_fp} \circ \delta$, where $\delta = \epsilon_i / 2^{k+1}, \circ \in \{+, -\}$
    – Call Metalibm to generate a polynomial approximation wrt. $\epsilon_{i\_approx}$
    – Generate a finite-precision implementation, such that $e_{fp} \leq \epsilon_{i\_fp}$
    – Compute cost $c_k$ of the obtained finite-precision implementation $i_k$
    – Consider following cases:
        • $c_k > c_{k-1}$: if $k = 1$ choose the opposite $\circ \in \{+, -\}$, else RETURN $i_{k-1}$
        • $c_k = c_{k-1}$: if $k = 1$ REPEAT, else RETURN $i_k$
        • $c_k < c_{k-1}$: if $k < 5$ REPEAT, else RETURN $i_k$
```

Fig. 2. High-level synthesis algorithm

in the program gets propagated through the remaining part of the computation, in the course of which it may be magnified, or diminished.

Because we are explicitly aiming to synthesize a more efficient program, we furthermore have to keep in mind the accuracy-efficiency trade off. If we assign a significant portion of the error budget to elementary function calls, we might need to use a higher, and thus more expensive, precision for the rest of the operations in order to satisfy the error budget for the whole program. Thus, performance gained by approximation might be negated by the need for high finite precision. This is a multiple-objective optimization task, which is known to be difficult in general.

Previous work provides only partial solutions to some of these challenges, which furthermore only exist in isolation. While Metalibm generates polynomial solutions with guaranteed bounds, it requires the user to provide range bounds and target errors *at the call site* and thus does not consider the full program and error propagation. Additionally, Metalibm only generates double-precision floating-point implementations. The tool Daisy can assign uniform or mixed fixed-point precisions to arithmetic computations, but does not consider elementary function calls or their approximations.

3.1 High-level Algorithm

In this paper, we provide a complete solution for the above mentioned challenges and propose an overall algorithm which takes into account the interactions between different errors and synthesizes efficient numerical kernels, which are guaranteed to be accurate up to a specified total error bound. [4]

We distinguish two error budgets. The *global* budget covers errors of elementary function calls and roundoffs of arithmetic operations in the original program.

[4] We optimize for running time, but our algorithm is also applicable to other objectives such as energy, with an appropriate cost function. We note that running time often correlates with energy.

The *local* budget covers the approximation error of individual elementary function calls and roundoff errors introduced by their polynomial approximations.

Figure 2 shows our high-level algorithm. The algorithm operates top-down. It first distributes the global error budget (subsection 3.2), which assigns local error budgets to individual elementary function calls. The local budget is distributed itself in a feedback loop between approximation and implementation errors (subsection 3.3). The approximation error, as well as other information obtained using static analysis is used to call Metalibm to generate polynomial approximations (subsection 3.4). Finally, the implementation error budget is used to assign fixed-point precisions to the approximation polynomials (subsection 3.5). We discuss alternatives to this top-down approach in subsection 3.6.

3.2 Distributing the Global Error Budget

Given the global error budget ϵ_g we first distribute it to local budgets for each arithmetic operation, variable and elementary function call, taking into account error propagation. Our *key observation* for this distribution is that the accuracy of the elementary function calls is unlikely to be very different from the other arithmetic operations, otherwise the errors they introduce would dominate the overall error. Based on this observation, we *treat the approximation errors introduced by elementary functions as a kind of roundoff error* of a given finite precision. With this assumption, we can leverage a precision assignment algorithm to distribute the global error budget.

In particular, we use the two assignment strategies implemented in the tool Daisy, which provide a uniform- or mixed-precision assignment. They assign a fixed-point precision to every arithmetic operation and elementary function call. For elementary functions, we interpret the associated roundoff error with this fixed-point format as the local error budget.

Daisy's uniform precision assignment performs a linear search and selects the smallest uniform precision which satisfies the provided overall error bound. Mixed-precision tuning is more involved, as it introduces cast operations which incur a certain cost. Unlike uniform precision assignment, mixed-precision tuning thus requires a cost function to choose between efficient programs. However, at this point, we do not know the actual implementation of the elementary function approximations. Furthermore, the performance of fixed-precision implementations on an accelerator depends on the compilation algorithm, which is a highly complex, and generally unknown function (e.g. the commercial Xilinx Vivado compiler). Thus the cost function has to *estimate* the cost of elementary function calls and arithmetic operations as well as possible.

We extend Daisy's mixed-precision tuning to be parametric in the cost function, which allows us to explore different options. We consider three cost functions: 1) an area-based [28] one used by Daisy previously, 2) one obtained with machine learning, and 3) an equally weighted combination of 1) and 2)[5].

[5] All cost functions are available in the source code in *repo*/opt/CostFunctions.scala.

For 2), we learned a multi-layer perceptron regressor [2] from random precision assignments on a set of benchmarks, for which we obtained actual performance data by compiling them to an FPGA with Xilinx Vivado. We furthermore extended both the area-based and the machine learned cost function so that elementary function calls incur twice the cost of arithmetic operations. The factor 2 has been found empirically; it confirms our intuition that the error introduced by the elementary function call is comparable to errors of arithmetic operations.

We have empirically determined that the weighted combination (i.e. option 3) works best in general. We have also observed that whether uniform or mixed precision is best is highly application specific. Thus, our algorithm tries both a uniform and a mixed-precision assignment with a weighted cost function and returns the better result. For our running example, uniform precision assignment performs best overall and assigns precision `Fixed(26)` to `sin(x1)`, `cos(2*x1)` and `sin(x2)`. From this, we obtain local error budgets $\epsilon_0 = \epsilon_1 = \epsilon_2 = 5.96\text{e-}8$.

3.3 Distributing the Local Error Budget

Once a local error budget ϵ_i is assigned to each individual elementary function call, we have to decide how much of ϵ_i will be spent on the approximation ϵ_{i_approx} and how much on the finite-precision implementation of the approximation polynomial ϵ_{i_fp}.

To find an optimal split between the two local budgets we use a refinement loop. We start with an equal split, i.e. $\epsilon_{i\_approx} = \epsilon_{i\_fp} = 0.5\epsilon_i$, synthesize a polynomial approximation respecting ϵ_{i_approx} and assign finite precision such that ϵ_{i_fp} is satisfied (see sections below). We then estimate a cost c_0 of the obtained implementation using a cost function.

Then, our algorithm increases ϵ_{i_fp} by $\delta = \epsilon_i/2^{k+1}$, where k is the number of steps taken in one direction, and decreases ϵ_{i_approx} respectively. We repeat synthesis of a polynomial and finite-precision assignment for the new values of ϵ_{i_fp} and ϵ_{i_approx} and compute the updated cost c_k. The obtained cost c_k is used to determine the fitness of the local error budget distribution. We accept an implementation found at the step $k - 1$ if $c_k > c_{k-1} \wedge k > 1$. In case the cost increases at the very first step, we change the direction of the search, i.e. decrease ϵ_{i_fp}, reset k to 0 and repeat the refinement. If the cost has not changed $c_k = c_{k-1}$ at the beginning of the search ($k = 1$), we make one more refinement iteration, for $k > 1$ the k-th implementation is accepted. If after the k-th step we have $c_k < c_{k-1}$, this indicates that the performance of the implementation at the step k has improved. We then repeat the refinement until the $(k-1)$-step implementation has been accepted. To ensure termination we set the maximum number of steps to $k = 5$.

The quality of refinement depends on how accurately a cost function reflects the actual compiler behavior, i.e. how well it can predict the circuit that will be implemented. Our approach is parametric in the cost function, which allows flexibility in optimization for different objectives and hardware. Similarly to global budget distribution with mixed-precision tuning, we evaluated an area-based, machine-learned and a combined cost function, and found that an equally weighted combination of the area-based and machine-learned cost function had best performance overall.

For our running example, the refinement loop needed two iterations for `sin(x1)`, meaning that the optimal distribution found was $\epsilon_{0\_fp} = 3\epsilon_{0\_approx}$. The corresponding values are: ϵ_{0_fp} =4.47e-8 and ϵ_{0_approx}= 1.49e-8. For `cos(2*x1)` and `sin(x2)` the initial equal split already had a minimum cost, i.e. $\epsilon_{i\_fp} = \epsilon_{i\_approx} = 2.98\text{e-}8$ for $i \in \{1, 2\}$.

3.4 Synthesizing the Approximation Polynomial

For finding a polynomial approximation of each individual elementary function we leverage the tool Metalibm. To generate an approximation, we need to specify the folllowing parameters: a) the elementary function $f(x)$ to be approximated, b) the domain $x \in I$, on which $f(x)$ will be approximated, c) the assigned local approximation error budget ϵ_{i_approx}, and d) the maximum polynomial degree. Note that domain I is not the input domain specified by the user, but the local input domain of the function's parameter x. This domain should be computed as tightly as possible, as this may allow Metalibm to use polynomials of smaller degree or less internal domain subdivisions. In general, determining these domains is challenging to do manually. Our algorithm uses static analysis of ranges and finite-precision errors using interval and affine arithmetic to compute this information fully automatically. Whenever a program contains the same elementary function call several times, we check whether we have already synthesized an approximation for a given range and assigned local error budget ϵ_i. In this case, we reuse already generated approximation.

We have empirically found a suitable value for the maximum polynomial degree to be 7, although Metalibm does not necessarily generate polynomials of degree 7. If possible, it will choose a smaller degree. Limiting the polynomial degree influences the number of domain subdivisions, and thus one looks for a good tradeoff between a reasonable number of subdivisions and polynomial degrees. We leave the remaining parameters of Metalibm to their default values.

Metalibm generates the approximation as code optimized for double floating-point precision. Therefore, most of the implemented optimizations for range reduction, expression decomposition and meta-splitting are not applicable to fixed-point implementations. Our implementation thus extracts only the generated piece-wise polynomial from the generated C code, and adds it to our top-level program as a separate function. The elementary function call is then replaced by the call to the generated function.

We currently do not support automated range reduction; for some of our benchmarks we have reduced ranges manually during preprocessing. In general, many programs implemented in fixed-point arithmetic will not need automatic range reduction, as many kernels have by design limited ranges. For other cases, adding the automatic range reduction is only an engineering task, since we already handle all necessary operations and have the ranges computed by Daisy.

```
def cos_0_02to1_5_err2_9802322387695312em08(x: Real): Real = {
  require((0.02 <= x) && (x <= 1.5))
    if ((x < 1.165)) {
      c0 + (c1 + (c2 + (c4 + (c6 + (c7 + c8*x)* x)* x*x)* x)* x)* x*x
  } else {
    let t = (x - 1.33) in
       b0 + (b1 + (b2 + (b3 + (b4 + b5*t)* t)* t)* t)* t
  }
} ensuring (res => res +/- 2.9802322387695312e-08) // finite-precision budget
```

Fig. 3. Approximation polynomial parsed from Metalibm output.

Figure 3 shows the extracted polynomial approximating `cos(2*x1)` over the input domain $[0.02, 1.5]$ with an approximation target error of 2.98e-8 for our running example.

3.5 Assigning Finite Precision

Once the approximation polynomial has been generated, our algorithm assigns a finite precision to the generated polynomials. The goal is to find an assignment that satisfies the local roundoff error budget ϵ_{i_fp}, but uses as coarse precision as possible for performance reasons. The generated polynomials contain branching, but the branches are always at the top-level. For this simple structure there are no discontinuity errors, i.e. errors due to diverging control-flow between the finite-precision and real-valued execution, so that we can safely handle each branch separately.

For assigning the lowest possible finite precision to obtained polynomials, such that their roundoff error e_{fp} satisfies ϵ_{i_fp}, we again leverage the uniform or mixed-precision assignment of Daisy. Finally, we re-run the roundoff error analysis on the whole program, where elementary function errors are replaced by the sum of e_{fp} and e_{approx}. This error is potentially smaller than the originally allocated local error budget, as Metalibm or the precision assignment usually cannot exhaust the budget due to complex, discrete constraints. That is, our tool in the end reports the actually achieved error of the final implementation.

3.6 Alternative Algorithm Designs

Alternatively to the proposed error distribution strategy, we could have designed our algorithm bottom-up: first assign local error budgets for both approximation and roundoff errors for elementary function calls, generate their approximations, then distribute what is left of the global error budget between other operations and variables. Or, we could first assign an approximation error budget to each elementary function call, generate approximations, then use the rest of the global error budget to assign finite-precisions to the entire generated program at once. We note that for these alternatives it is unclear how to distribute the initial error budgets. Crucial information becomes available only later so backtracking would likely be necessary, which may be costly. In the top-down approach, we

Target errors	small				large			
Benchmark	baseline	approx	target	actual	baseline	approx	target	actual
axisRot.X	52-60	**24**	1.49e-10	7.52e-11	30-34	**14**	1.49e-6	5.5e-7
axisRot.Y	52-60	**24**	1.49e-10	7.52e-11	30-34	**14**	1.49e-6	5.5e-7
fwdk2jX	97-113	**23**	8.39e-11	2.98e-11	30-34	**24**	8.39e-7	2.41e-7
fwdk2jY	94-110	**22**	4.89e-11	1.49e-11	30-34	**12**	4.89e-7	1.06e-7
xu1	97-113	**43**	1.89e-10	2.47e-10	53-61	**14**	1.89e-6	1.93e-6
xu2	96-112	**44**	1.88e-10	2.3e-10	54-62	**13**	1.88e-6	1.86e-6
rodriguesRot.	52-60	**25**	1.70e-8	1.11e-8	31-35	**14**	1.70e-4	9.07e-5
sinxx10	52-60	**28**	2.51e-9	1.61e-9	31-35	**15**	2.51e-5	1.26e-5
pendulum1	33-37	**27**	4.79e-11	3.74e-11	32-36	**16**	4.79e-7	3.06e-7
pendulum2	53-61	**26**	1.07e-10	8.11e-11	32-36	**15**	1.07e-6	6.64e-7
pred.Gaus.	84	119-125	4.15e-7	4.07e-7	58	77	4.15e-3	4.08e-3
pred.SVC	22	20-28	1.46e-6	1.47e-7	21	21	1.46e-2	6.82e-4
pred.MLPLog.	195	**191**	2.15e-6	4.14e-10	143	**126**	2.15e-2	7.21e-7

Table 1. Running time in machine cycles of baseline and synthesized programs, and error budgets together with the achieved accuracy

still have to distribute the global error budget in the first step, but we do so on the top-level program, without it being prohibitively large.

4 Experimental Evaluation

We implemented our algorithm on top of the tools Daisy and Metalibm and evaluate it on several benchmarks from scientific computing, embedded and machine learning domains. We evaluate our approach on a commonly used FPGA board (Xylinx Zync 7000 with 10ns clock period), but note that our technique is not specific to any particular hardware and believe that our results qualitatively carry over. Synthesis has been performed on a MacBook Pro with an 3.1 GHz Intel Core i5 processor and 16 GB RAM, macOS Mojave 10.14.

Our set of benchmarks contains programs with up to 5 elementary function calls in straight-line code (all benchmarks are provided in the appendix). The number of elementary function calls for each benchmark is shown in Table 2. The benchmarks *predictGaussianNB*, *predictSVC* and *predictMLPLogistic* are machine learning classifiers generated by the python scikit-learn library on the standard Iris data set. The benchmarks *forwardk2j** are taken from the Axbench approximate computing benchmark suite [47] and compute a forward kinetics expression. We have created the benchmarks *axisRotation*, rodrigues-Rotation*, which rotate coordinate axes and a vector respectively. The *pendulum** benchmarks come from the Rosa project for analysis of finite-precision code [15]. Finally, benchmarks *xu** and *sinxx10* are from the CORPIN project [1].

We perform all experiments for two different sets of target errors—small and large. To obtain these error bounds, we first run roundoff analysis on the benchmarks with uniform fixed-point precision with 32 bits. Small and large target errors are by two orders of magnitude smaller, resp. larger than these computed roundoff errors. Both target errors are reported in Table 1.

For performance measurements we compile our generated programs using Xilinx Vivado HLS v.2019.1, which reports the minimum and maximum number

Benchmark	# elem. fnc calls	Small errors		Large errors	
		time	# arith. ops	time	# arith. ops
axisRot.X	2	3m 13.26s	142	41.54s	48
axisRot.Y	2	3m 1.61s	142	40.66s	48
fwdk2jX	2	5m 56.5s	222	1m 35.33s	102
fwdk2jY	2	1m 29.16s	71	24.75s	24
xu1	3	3m 50.24s	168	50.97s	61
xu2	3	6m 56.96s	212	1m 31.22s	73
rodriguesRot.	2	2m 40.15s	126	30.73s	45
sinxx10	1	1m 38.28s	71	25.8s	24
pendulum1	1	2m 18.36s	71	27.64s	22
pendulum2	1	1m 43.24s	71	23.98s	24
pred.Gaus.	5	1h 45m 27.7s	708	4m 26.231s	255
pred.SVC	1	21m 33.35s	247	1m 51.62s	95
pred.MLPLog.	1	3h 19m 48.57s	399	57m 29.185s	170

Table 2. Size of the generated polynomials and the running times for program synthesis

of machine cycles of the compiled design, and thus provides an exact performance measurement. We do not measure actual running time as such a measurement is necessarily noisy.

The baseline programs against which we compare correspond to the programs a user can implement with today's state of the art: by running Daisy on the input program without approximations to assign a uniform precision to all operations and then by compiling the generated code using Xilinx' elementary function library. The compiled programs can use either the fixed-point or the floating-point versions of library functions. For our baseline, we evaluate all valid versions (those which satisfy the overall error bound), and use the smallest number of cycles obtained.

Performance Improvements Table 1 compares the running time in terms of machine cycles of programs synthesized by our approach (columns 3 and 7) with the baseline implementation (columns 2 and 6) for small and large target errors. A pair '52-60' denotes minimum and maximum cycles; whenever these values coincide, we show only one number. We report the number of cycles for the fastest approximated program, obtained by distributing the global error budget using either uniform or mixed-precision assignment.

For all benchmarks, except *predictGaussianNB*, we observe a significant performance improvement when elementary function calls are replaced with piecewise polynomial approximations. Our synthesized approximate programs run on average 2.23x faster than the baseline, and up to 4.64x (4.46x) faster for small (large) target errors respectively.

For 10 out of 13 of the benchmarks the largest speedup was achieved when using uniform precision assignment for both top-level program and polynomial approximations. For three *predict** benchmarks the best performance has been achieved using mixed-precision tuning. We believe that mixed-precision can be improved further by using a more accurate cost function. Disabling the refinement loop produced slower programs for 3 benchmarks and did not change results for the rest. We observed the largest speedup when using a combination of the area-based and the machine-learned cost functions.

We noticed that on the benchmark *predictGaussianNB* the baseline programs run faster than the synthesized ones. We suspect the reason is that *predictGaussianNB* repeatedly calls the `log` function on slightly different, but largely overlapping, domains. Our implementation generates a different polynomial for each call, when in this scenario reusing the code seems to be beneficial. We leave the detection of such cases to future work. We noticed that the largest improvements are observed for benchmarks with `sin, cos`, whereas for the `exp` function in the *predictMLPLogistic* improvements are smaller, and for *predictSVC*, our approach cannot improve the running time. We suspect this effect is due to an efficient implementation of `exp` in the Xilinx math library.

Accuracy Comparison In Table 1 we also show the target errors (columns 4 and 8), as well as the errors of the best synthesized approximated programs (columns 5 and 9), for both the small and large error setting. We observe that not all of the available error budget is used up by our synthesized programs. This is to be expected, as the space of precisions is not continuous, and a precision even 1 bit less precise may just not be enough to satisfy the target error. Small error budgets are used up more than large ones: for small error budgets the average usage is 62.33%, while for the large budget it is only 41.12%. The coarser a finite precision gets, the greater becomes the difference between roundoff errors computed for two neighboring precisions.

Size of Generated Approximations Table 2 shows the number of elementary functions and the size of the generated polynomials (sum over all elementary functions) per benchmark (for the setting with the largest performance improvement, as reported in Table 1). Factors that influence the reported total size are: *a)* the number of elementary function calls with distinct input ranges and local error budgets, because we generate an approximation for each of them; *b)* the local error budget and thus approximation error budget, which influences the size of each polynomial inversely, the smaller the error budget, the larger the polynomial satisfying this budget needs to be. The largest total size of generated polynomials is 708. (We note that this size exceeds the sizes of benchmarks usually used in the area of sound roundoff error verification and optimization by an order of magnitude.)

Running times Table 2 shows the synthesis times of our implementation. As expected, synthesis of programs with small target errors is significantly slower than with large ones, but still reasonable as synthesis needs to be run only once. Smaller target errors usually require polynomial approximations with larger degrees and result in larger programs. Additionally, to satisfy smaller roundoff error bounds, finite-precision tuning has to consider higher precisions, thus searching a larger space for a suitable precision assignment.

5 Related Work

Program Synthesis Program synthesis [6] aims to automatically generate programs from (possibly declarative) specifications, and has had considerable success to generate programs from a variety of domains [24,4,12,32,39,19,22,38]. However, the vast majority of these techniques require that the generated program satisfies the user-given specification exactly. Furthermore, most approaches do not explicitly optimize for a non-correctness metric.

A branch of program synthesis – automated repair – allows to modify parts of a program to satisfy given criteria. The tool AutoRNP [48] repairs numerical programs by detecting an input subdomain that triggers high floating-point errors and replacing the matematical function by its piece-wise quadratic approximation on this subdomain. Opposite to our approach, AutoRNP aims to increase accuracy while introducing time overhead for repaired programs.

The Metasketches framework [7] searches for an optimal program with smallest cost according to a cost function. It has been used for synthesizing polynomial approximations, however, the accuracy of the generated programs is only verified based on a small set of test inputs, and thus without accuracy guarantees. In contrast, Metalibm's polynomial approximation algorithm is guaranteed to find the best polynomial approximation, and our entire approach guarantees end-to-end accuracy.

Approximate Computing Our approach trades acceptable accuracy loss for resource savings. This idea has been recently pursued extensively under the name of approximate computing [46]. Techniques in this domain span all layers of the computing stack from approximate hardware [29] to software-based approximations such as skipping loop iterations [43] or removing synchronization [41]. Most related to our work from this domain is another recent combination of Daisy and Metalibm [13]. However, it only considers floating-point arithmetic and, unlike our tool, does not optimize obtained approximations. Another approximate computing tool Chisel [34] optimizes arithmetic programs by selecting which operations can be run on approximate hardware. Its error analysis is a slightly simplified version of ours in this work. While Chisel considers also probabilistic specifications, it only optimizes arithmetic operations.

Other work allows programmers to specify several versions of a program with different accuracy-efficiency tradeoffs, and let a specialized compiler autotune a program to a particular environment [5]. While this approach handles programs of larger size than ours, it requires the library writer to provide the different versions, together with accuracy specifications. It furthermore ensures accuracy by testing, i.e. does not provide guarantees.

Approximations can be particularly efficient when run on custom hardware, such as neural processing units, for which one can learn an approximate program which mimics the original imperative one [18]. Verification is again performed only on a limited set of test inputs. STOKE is an autotuner which operates on low-level machine code, and has also been applied to generate approximate floating-point programs [42]. Its scalability is limited as it considers low-level code, and furthermore it also cannot guarantee accuracy.

Finally, approximations are naturally also applied manually, e.g. for obtaining efficient, low-resource heartbeat classifiers [40]. In particular, this work has approximated an exponential function by a piece-wise linear function, but due to the manual process without accuracy guarantees.

Numerical Program Analysis We reviewed roundoff error analysis tools in section 2; they all assume fixed library implementations and do not optimize for efficiency. Library functions themselves have been also verified for accuracy [30]. Mixed-precision tuning approaches do optimize programs, but are only applicable in a relatively restricted space where one floating-point precision is not enough. Our presented work leverages the much larger tradeoff space of fixed-point arithmetic *and* elementary function approximations, and achieves significantly larger performance savings.

6 Conclusion

We have presented a fully automated synthesis approach for generating efficient numerical kernels for accelerator hardware by approximating elementary function calls as well as individual arithmetic operations, while guaranteeing user-provided error bounds for the entire program. Our technique relies on an existing static analysis to verify the end-to-end accuracy of introduced approximations. The strength of the approach comes as a result of our key intuition that the approximation errors should not be vastly different from arithmetic errors, and our experiments confirmed this intuition. To navigate the large search space of possible approximations, we present a novel error distribution algorithm and extend existing search techniques for assigning finite precisions, as well as a synthesis technique which guarantees to generate the best polynomial approximation. In a suitable combination, these techniques allow us to generate programs which are significantly more efficient than current default implementations.

A Benchmarks

All benchmarks are provided below. Error specification given in the ensuring clause corresponds to the small error, the larger errors are given in comments.

```
def axisRotationX(x: Real, y: Real, theta: Real): Real =  {
  require(-2 <= x && x <= 2 && -2 <= y && y <= 2 && 0.01 <= theta && theta <= 1.5)
  x * cos(theta) + y * sin(theta)
} ensuring (res => res +/- 1.49e-10) // 1.49e-06

def axisRotationY(x: Real, y: Real, theta: Real): Real = {
  require(-2 <= x && x <= 2 && -2 <= y && y <= 2 && 0.01 <= theta && theta <= 1.5)
  -x * sin(theta) + y * cos(theta)
} ensuring (res => res +/- 1.49e-10) // 1.49e-06

def forwardk2jX(theta1: Real, theta2: Real): Real = {
  require(0.01 <= theta1 && theta1 <= 1.5 && 0.01 <= theta2 && theta2 <= 1.5)
  val l1: Real = 0.5
  val l2: Real = 0.5

  l1 * cos(theta1) + l2 * cos(theta1 + theta2)
} ensuring (res => res +/- 8.39e-11) // 8.39e-07

def forwardk2jY(theta1: Real, theta2: Real): Real = {
  require(0.01 <= theta1 && theta1 <= 1.5 && 0.01 <= theta2 && theta2 <= 1.5)
  val l1: Real = 0.5
  val l2: Real = 0.5

  l1 * sin(theta1) + l2 * sin((theta1 + theta2) / 2)
} ensuring (res => res +/- 4.89e-11) //4.89e-07

def rodriguesRotation(v1: Real, v2: Real, v3: Real,
     k1: Real, k2: Real, k3: Real, theta: Real): Real = {
  require(-2 <= v1 && v1 <= 2 && -2 <= v2 && v2 <= 2 &&
   -2 <= v3 && v3 <= 2 && -5 <= k1 && k1 <= 5 && -5 <= k2 &&
   k2 <= 5 && -5 <= k3 && k3 <= 5 && 0 <= theta && theta <= 1.5)

  val t1 = cos(theta)
  v1 * t1 + (k2 * v3 - k3 * v2) * sin(theta) +
            k1 * (k1 * v1 + k2 * v2 + k3 * v3) * (1 - t1)
} ensuring (res => res +/- 1.7e-08) // 1.7e-04


def sinxx10(x: Real): Real = {
  require(0.01 <= x && x <= 1.5)
  val t1 = sin(x)
  (3 * x * x * x - 5 * x + 2) * t1 * t1 + (x * x * x + 5 * x) * t1 - 2*x*x - x - 2
} ensuring(res => res +/- 2.51e-09) // 2.51e-05


def xu1(x1: Real, x2: Real): Real = {
  require(0.01 <= x1 && x1 <= 0.75 && 0.01 <= x2 && x2 <= 1.5)
  2 * sin(x1) + 0.8 * cos(2 * x1) + 7 * sin(x2) - x1
} ensuring(res => res +/- 4.24e-10) // 4.24e-06

def xu2(x1: Real, x2: Real): Real = {
  require(0.01 <= x1 && x1 <= 1.5 && 0.01 <= x2 && x2 <= 0.5)
  1.4 * sin(3 * x2) + 3.1 * cos(2 * x2) - x2 + 4 * sin(2 * x1)
} ensuring(res => res +/- 5.8e-10) // 5.8e-06

 def pendulum1(t: Real, w: Real): Real = {
   require(0.01 <= t && t <= 1.6 && -5 <= w && w <= 5)
   val h: Real = 0.01
   val L: Real = 2.0
   val m: Real = 1.5
   val g: Real = 9.80665
   val k1w = -g/L * sin(t)
   val k2t = w + h/2*k1w
   val tNew = t + h*k2t
   tNew
 } ensuring(res => res +/- 4.79e-11) // 4.79e-07

 def pendulum2(t: Real, w: Real): Real = {
   require(0.05 <= t && t <= 1.5 && -5 <= w && w <= 5)
   val h: Real = 0.01
   val L: Real = 2.0
   val m: Real = 1.5
   val g: Real = 9.80665
   val k1t = w
   val k2w = -g/L * sin(t + h/2*k1t)
   val wNew = w + h*k2w
   wNew
 } ensuring(res => res +/- 1.07e-10) // 1.07e-06

// Gaussian Naive Bayes classifier
def predictGaussianNB(f0: Real, f1: Real, f2: Real, f3: Real, sigma0: Real,
  sigma1: Real, sigma2: Real, sigma3: Real, theta0: Real, theta1: Real,
  theta2: Real, theta3: Real, prior: Real): Real = {
  require(0.12 <= sigma0 && sigma0 <= 0.40 && 0.09 <= sigma1 && sigma1 <= 0.15 &&
    0.02 <= sigma2 && sigma2 <= 0.30 && 0.01 <= sigma3 && sigma3 <= 0.08 &&
    5.0 <= theta0 && theta0 <= 6.6 && 2.7 <= theta1 && theta1 <= 3.5 &&
    1.4 <= theta2 && theta2 <= 5.6 && 0.2 <= theta3 && theta3 <= 2.1 &&
    0.25 <= prior && prior <= 0.5 && 4.0 <= f0 && f0 <= 8.0 &&
    2.0 <= f1 && f1 <= 4.5 && 1.0 <= f2 && f2 <= 7.0 && 0.0 <= f3 && f3 <= 2.5)

  val pi: Real = 3.141
  val sum = log(2.0 * pi * sigma0) + log(2.0 * pi * sigma1) +
                 log(2.0 * pi * sigma2) + log(2.0 * pi * sigma3)
  val nij = -0.5 * sum
  val sum2 = (f0 - theta0) * (f0 - theta0) / sigma0 +
    (f1 - theta1) * (f1 - theta1) / sigma1 +
    (f2 - theta2) * (f2 - theta2) / sigma2 +
    (f3 - theta3) * (f3 - theta3) / sigma3

  -0.5 * sum - 0.5 * sum2 + log(prior)
} ensuring (res => res +/- 4.15e-07) // 4.15e-03

// C-Support Vector Classification with rbf kernel
def predictSVC(f0: Real, f1: Real, f2: Real, f3: Real, vectors0: Real,
  vectors1: Real, vectors2: Real, vectors3: Real, coefficient: Real,
  intercept: Real, factor: Real): Real = {
```

```
  require(4.0 <= f0 && f0 <= 7.0 && 2.0 <= f1 && f1 <= 4.5 &&
    1.0 <= f2 && f2 <= 6.0 && 0.0 <= f3 && f3 <= 2.5 &&
    4.5 <= vectors0 && vectors0 <= 5.9 && 2.2 <= vectors1 && vectors1 <= 4.4 &&
    1.3 <= vectors2 && vectors2 <= 4.9 && 0.2 <= vectors3 && vectors3 <= 2.3 &&
    -0.12 <= intercept && intercept <= 0.06 && -1 <= coefficient && coefficient <= 1.0 &&
    5 <= factor && factor <= 50)

  val gamma: Real = 0.1
  val k = (vectors0 - f0) * (vectors0 - f0) + (vectors1 - f1) * (vectors1 - f1) +
                   (vectors2 - f2) * (vectors2 - f2) + (vectors3 - f3) * (vectors3 - f3)
  val kernel = exp(gamma * k)

  factor * coefficient * kernel + intercept
} ensuring (res => res +/- 1.46e-06) // 1.46e-02

// Multi-layer Perceptron with logistic activation function
def predictMLPLogistic(f0: Real, f1: Real, f2: Real, f3: Real, weights_0_0: Real,
  weights_0_1: Real, weights_0_2: Real, weights_0_3: Real, weights_1_0: Real,
  weights_1_1: Real, weights_1_2: Real, bias_0: Real, bias_1: Real): Real = {

  require(4.0 <= f0 && f0 <= 8.0 && 2.0 <= f1 && f1 <= 4.5 &&
    1.0 <= f2 && f2 <= 7.0 && 0.0 <= f3 && f3 <= 2.5 &&
    -0.3 <= weights_0_0 && weights_0_0 <= 0.3 && -0.5 <= weights_0_1 && weights_0_1 <= 0.0 &&
    -0.2 <= weights_0_2 && weights_0_2 <= 0.1 && 0.1 <= weights_0_3 && weights_0_3 <= 0.3 &&
    -0.4 <= weights_1_0 && weights_1_0 <= 0.8 && -0.3 <= weights_1_1 && weights_1_1 <= 0.3 &&
    0.0 <= weights_1_2 && weights_1_2 <= 0.4 &&
    0.0 <= bias_0 && bias_0 <= 0.5 && -0.4 <= bias_1 && bias_1 <= 0.5)

  val n1 = f0 * weights_0_0 + f1 * weights_0_1 + f2 * weights_0_2 + f3 * weights_0_3 + bias_0
  val hidden = 1.0 / (1.0 + exp(-n1))

  val n2 = hidden * weights_1_0 + hidden * weights_1_1 + hidden * weights_1_2 + bias_1
  1.0 / (1.0 + exp(-n2))

} ensuring (res => res +/- 2.15e-06) // 2.15e-02
```

References

1. Project CORPIN. https://www-sop.inria.fr/corpin/logiciels/ALIAS/Benches/
2. Python sklearn - multi-layer perceptron regressor. `https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPRegressor.html` (2019)
3. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: FMCAD. pp. 1–8. IEEE (2013)
4. Alur, R., Radhakrishna, A., Udupa, A.: Scaling enumerative program synthesis via divide and conquer. In: TACAS. pp. 319–336. Springer (2017)
5. Ansel, J., Wong, Y.L., Chan, C., Olszewski, M., Edelman, A., Amarasinghe, S.: Language and compiler support for auto-tuning variable-accuracy algorithms. In: CGO (2011)
6. Bodik, R., Jobstmann, B.: Algorithmic program synthesis: introduction. STTT **15**(5), 397–411 (2013)
7. Bornholt, J., Torlak, E., Grossman, D., Ceze, L.: Optimizing Synthesis with Metasketches. In: POPL (2016)
8. Brunie, N., d. Dinechin, F., Kupriianova, O., Lauter, C.: Code generators for mathematical functions. In: ARITH (2015)
9. Chiang, W.F., Baranowski, M., Briggs, I., Solovyev, A., Gopalakrishnan, G., Rakamarić, Z.: Rigorous Floating-Point Mixed-Precision Tuning. In: POPL (2017)
10. Damouche, N., Martel, M.: Mixed precision tuning with salsa. In: PECCS. pp. 185–194. SciTePress (2018)
11. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. STTT **19**(4), 427–448 (2017)
12. D'Antoni, L., Samanta, R., Singh, R.: Qlose: Program repair with quantitative objectives. In: CAV. pp. 383–401. Springer (2016)
13. Darulova, E., A., V.: Sound approximation of programs with elementary functions. In: CAV (2018)
14. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs. In: TACAS (2018)
15. Darulova, E., Kuncak, V.: Towards a Compiler for Reals. TOPLAS **39**(2) (2017)
16. Darulova, E., Sharma, S., Horn, E.: Sound mixed-precision optimization with rewriting. In: ICCPS (2018)
17. De Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted Verification of Elementary Functions using Gappa. In: ACM Symposium on Applied Computing (2006)
18. Esmaeilzadeh, H., Sampson, A., Ceze, L., Burger, D.: Neural acceleration for general-purpose approximate programs. In: MICRO (2012)
19. Feng, Y., Martins, R., Wang, Y., Dillig, I., Reps, T.W.: Component-based synthesis for complex apis. In: POPL (2017)
20. de Figueiredo, L.H., Stolfi, J.: Affine Arithmetic: Concepts and Applications. Numerical Algorithms **37**(1-4) (2004)
21. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: VMCAI (2011)
22. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: POPL (2011)
23. IEEE: IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2008 (2008)
24. Kneuss, E., Kuraj, I., Kuncak, V., Suter, P.: Synthesis modulo recursive functions. In: OOPSLA (2013)

25. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: PLDI (2010)
26. Kupriianova, O., Lauter, C.: A domain splitting algorithm for the mathematical functions code generator. In: Asilomar (2014)
27. Kupriianova, O., Lauter, C.: Metalibm: A mathematical functions code generator. In: ICMS (2014)
28. Lee, D.U., Gaffar, A.A., Cheung, R.C., Mencer, O., Luk, W., Constantinides, G.A.: Accuracy-guaranteed bit-width optimization. TCAD **25**(10), 1990–2000 (Oct 2006)
29. Lee, S., John, L.K., Gerstlauer, A.: High-level synthesis of approximate hardware under joint precision and voltage scaling. In: DATE. pp. 187–192 (2017)
30. Lee, W., Sharma, R., Aiken, A.: On automatically proving the correctness of math.h implementations. In: POPL (2018)
31. Lohar, D., Darulova, E., Putot, S., Goubault, E.: Discrete choice in the presence of numerical uncertainties. IEEE TCAD (Nov 2018)
32. Loncaric, C., Torlak, E., Ernst, M.D.: Fast synthesis of fast collections. ACM SIGPLAN Notices **51**(6), 355–368 (2016)
33. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds Using Semidefinite Programming. ACM Trans. Math. Softw. **43**(4) (2017)
34. Misailovic, S., Carbin, M., Achour, S., Qi, Z., Rinard, M.C.: Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In: OOPSLA (2014)
35. Moore, R.: Interval Analysis. Prentice-Hall (1966)
36. Moscato, M., Titolo, L., Dutle, A., Muñoz, C.: Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. In: SAFECOMP (2017)
37. N.Briesbarre, S.Chevillard: Efficient polynomial l-approximations. In: ARITH (2007)
38. Neider, D., Saha, S., Madhusudan, P.: Compositional synthesis of piece-wise functions by learning classifiers. ACM Trans. Comput. Logic **19**(2), 10:1–10:23 (May 2018)
39. Nori, A.V., Ozair, S., Rajamani, S.K., Vijaykeerthy, D.: Efficient synthesis of probabilistic programs. In: PLDI. ACM (2015)
40. R.Braojos, G.Ansaloni, D.: A methodology for embedded classification of heartbeats using random projections. In: DATE. EPFL (2013)
41. Renganarayana, L., Srinivasan, V., Nair, R., Prener, D.: Programming with relaxed synchronization. In: RACES. pp. 41–50 (2012)
42. Schkufza, E., Sharma, R., Aiken, A.: Stochastic Optimization of Floating-point Programs with Tunable Precision. In: PLDI (2014)
43. Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., Rinard, M.: Managing performance vs. accuracy trade-offs with loop perforation. In: ESEC/FSE (2011)
44. Solovyev, A., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-off Errors with Symbolic Taylor Expansions. In: FM (2015)
45. Xilinx: Vivado design suite. `https://www.xilinx.com/products/design-tools/vivado.html` (2018)
46. Xu, Q., Mytkowicz, T., Kim, N.S.: Approximate computing: A survey. IEEE Design Test **33**(1), 8–22 (Feb 2016)
47. Yazdanbakhsh, A., Mahajan, D., Esmaeilzadeh, H., Lotfi-Kamran, P.: Axbench: A multiplatform benchmark suite for approximate computing. IEEE Design Test **34**(2) (2017)
48. Yi, X., Chen, L., Mao, X., Ji, T.: Efficient automated repair of high floating-point errors in numerical libraries. In: POPL (2019)