Modular Optimization-Based Roundoff Error Analysis of Floating-Point Programs

Rosa Abbasi[1][0000−0003−1495−3470] and Eva Darulova[2][0000−0002−6848−3163](✉)

[1] MPI-SWS, Kaiserslautern and Saarbrücken, Germany rosaabbasi@mpi-sws.org
[2] Uppsala University, Uppsala, Sweden eva.darulova@it.uu.se

Abstract. Modular static program analyses improve over global whole-program analyses in terms of scalability at a tradeoff with analysis accuracy. This tradeoff has to-date not been explored in the context of sound floating-point roundoff error analyses; available analyses computing guaranteed absolute error bounds effectively consider only monolithic straight-line code. This paper extends the roundoff error analysis based on symbolic Taylor error expressions to non-recursive procedural floating-point programs. Our analysis achieves modularity and at the same time reasonable accuracy by automatically computing abstract procedure summaries that are a function of the input parameters. We show how to effectively use first-order Taylor approximations to compute precise procedure summaries, and how to integrate these to obtain end-to-end roundoff error bounds. Our evaluation shows that compared to an inlining of procedure calls, our modular analysis is significantly faster, while nonetheless mostly computing relatively tight error bounds.

Keywords: modular verification · floating-point arithmetic · roundoff error · Taylor approximation.

1 Introduction

One of the main challenges of automated static program analysis is to strike a suitable trade-off between analysis accuracy and performance [6]. This trade-off is inevitable, as a certain amount of abstraction and thus over-approximation is necessary to make an analysis feasible for unbounded (or very large) input domains. There are typically different ways to introduce abstractions; for instance by considering abstract domains that are more or less accurate [6,17,22], or in the context of procedural code, by abstracting procedure calls by summaries or specifications to obtain a *modular* analysis [8].

A modular analysis allows each procedure to be analyzed independently once, regardless of how often it is being called in an application, rather than being re-analyzed in possibly only slightly different contexts at every call site. This saves analysis time and thus increases the scalability of the analysis at the expense of some loss of accuracy: the procedure summaries need to abstract over different calling contexts.

This paper presents a modular roundoff error analysis for non-recursive procedural floating-point programs without conditional branches. Our approach extends the roundoff error analysis first introduced in the FPTaylor tool [27] that is based on symbolic Taylor expressions and global optimization and that has been shown to produce tight error bounds for straight-line arithmetic expressions. Our analysis first computes, for each procedure separately, error specifications that provide an abstraction of the function's behavior as a function of the input parameters. In a second step, our analysis instantiates the error specifications at the call sites to compute an overall roundoff error bound for each procedure.

The main challenge is to achieve a practically useful tradeoff between analysis accuracy and performance. A naive, albeit simple, approach would simply compute the worst-case roundoff error for each procedure as a constant, and would use this constant as the error at each call site. This approach is, however, particularly suboptimal for a roundoff error analysis, because roundoff errors depend on the magnitude of arguments. For reasonable analysis accuracy, it is thus crucial that the error specifications are *parametric* in the procedure's input parameters. At the same time, the error specifications need to introduce some *abstraction* as we otherwise end up re-analyzing each procedure at each call site.

We achieve this balance by computing error specifications that soundly over-approximate roundoff errors using *first-order* Taylor approximations separately for propagation of input and roundoff errors. By keeping first-order terms of both approximations unevaluated, we obtain parametric procedure summaries, and by eagerly evaluating higher-order terms we achieve abstraction that has a relatively small impact on accuracy.

Available sound floating-point roundoff error analyses have largely focused on abstractions for the (global) analysis of straight-line code and require function calls to be inlined manually [10,18,27,24] and are thus non-modular. The tool PRECiSA [26,28] analyzes function calls compositionally, however, does not apply abstraction when doing so. The analysis can thus be considered modular (in principle), but the computed symbolic function summaries can be very large and negatively affect the efficiency of the analysis. Goubault et al. [19] present a modular roundoff error analysis based on the zonotopic abstract domain that does apply abstraction at function calls. However, the implementation is not available and the roundoff error analyses based on zonotopes have been shown to be less accurate than the alternative approach based on symbolic Taylor expressions.

Like most existing roundoff error analyses, our analysis computes absolute roundoff errors for programs without loops or recursive procedure calls and without conditional branches; these remain an open, but orthogonal, challenge for floating-point roundoff error analysis [11,28]. The optimization-based approach

that we extend in this paper has been used for the computation of relative error bounds [21] as well, however, relative errors are fundamentally undefined for input domains that include zeros and are thus less widely applicable.

We implement our analysis in a tool called Hugo and evaluate it on two case studies that are inspired by existing floating-point benchmarks [2]. Our evaluation shows that compared to an approach based on procedure inlining and an analysis by state of the art roundoff analysis tools, our modular analysis provides an interesting tradeoff: it is significantly faster, while computing comparable error bounds that are often within the same order of magnitude and thus, in our opinion, practically useful.

Contributions To summarize, this paper presents the following contributions:

- a sound modular roundoff error analysis for non-recursive procedural code without conditionals that combines modularity and abstraction when analyzing function calls;
- a prototype implementation of our analysis is available open-source at https://doi.org/10.5281/zenodo.8175459;
- an empirical evaluation of the accuracy-performance tradeoff of our analysis.

2 Background

In this section, we provide necessary background on floating-point arithmetic and roundoff error analysis, focusing on the symbolic Taylor expression-based roundoff error analysis that has been implemented in several tools. We extend this analysis in Section 3 to support procedure calls. Throughout, we use bold symbols to represent vectors.

Floating-point Arithmetic The IEEE754 standard [1] formalizes floating-point numbers and the operations over them. A floating-point number is defined as a triple (sng, sig, exp) indicating its sign, significant, and exponent, with the numerical value being $(-1)^{sng} \times sig \times 2^{exp}$. The standard introduces four general binary formats (16, 32, 64 and 128 bits) varying on the sizes of sig and exp. We assume 64 bit double precision throughout this paper, but our approach generalizes to other formats as well.

The standard introduces several rounding operators that return the floating-point number that is closest to the input real number where the closeness is defined by the specific rounding operator. The most common rounding operator is rounding to nearest (ties to even), which we assume in this paper.

The distance between the real value and the floating-point representation is called the *roundoff error*. Computing this difference exactly is practically infeasible for all but very short computations. Instead, we and most other roundoff error analysis tools assume the following rounding model that holds for the rounding to nearest mode:

$$rnd(op) = op(1 + e) + d \quad \text{where } |e| \leq \epsilon, |d| \leq \delta \tag{1}$$

Where *op* is an arithmetic operation (or an input value or constant), ϵ bounds the relative error and δ bounds the absolute error. For the standard arithmetic operations $+, -, *, /$, the IEEE754 standard specifies for double precision $\epsilon = 2^{-53}$ and $\delta = 2^{-1075}$, where the latter captures the roundoff error of subnormal floating-point numbers, i.e. numbers very close to zero. For library function calls to common mathematical functions, e.g. *sin*, *exp*, etc., the library specification typically specifies the corresponding error(s); in this paper we assume $2 * \epsilon$ (most libraries provide this bound or better, but our analysis is parametric).

Sound Roundoff Error Analysis The goal of a roundoff error analysis is to compute the worst-case absolute error:

$$\max_{\mathbf{x},\tilde{\mathbf{x}} \in I} |f(\mathbf{x}) - \tilde{f}(\tilde{\mathbf{x}})| \tag{2}$$

where $f(\mathbf{x})$ denotes an idealized (purely) numerical program, where $\mathbf{x}$ is a possibly multivariate input, and $\tilde{f}(\tilde{\mathbf{x}})$ represents the function corresponding to the floating-point implementation, which has the same syntax tree but with operations interpreted in floating-point arithmetic. Note that the input to $\tilde{f}$, $\tilde{\mathbf{x}}$, is a rounded version of the real-valued input since that may not be exactly representable in finite precision and may need to be rounded.

We want to maximize the above equation for a set of meaningful inputs I that depends on a particular application. Bounding roundoff errors for unbounded input ranges is not practically useful as the error bounds are then in general unbounded.

In this paper, we consider programs that consist of several procedures and the goal is to compute an error bound for each of them. The procedure bodies consists of arithmetic expressions, mathematical library function calls, (immutable) variable declarations and (possibly) calls to procedures defined within the program.

To estimate the rounding error for such programs with existing roundoff error analyses, the procedure calls need to be effectively inlined—either manually by a user before an analysis tool is run, or automatically by the tool [26]. For larger programs, especially with more procedure calls, this can result in very large (symbolic) expressions and thus long analysis times. This approach is also fundamentally not suitable for integration into modular verification frameworks, such as KeY [3] or Frama-C [23].

For our modular analysis, the procedure calls do not need to be inlined. Instead, for each procedure of the program, our analysis first computes an error specification that is a function of the input parameters and that abstracts some of the error computation. Our analysis instantiates these error specifications at the call sites to compute an overall roundoff error bound (it also checks that the preconditions of the called procedures are respected).

Symbolic Taylor Expression-Based Roundoff Analysis The approach to roundoff error analysis (for straight-line code) that we extend in Section 3 was first proposed in the tool FPTaylor [27]. This approach abstracts the floating-point

function $\tilde{f}(\tilde{\mathbf{x}})$ using the rounding model from Equation 1 into a real-valued function $\hat{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$ to compute a bound on the roundoff error:

$$\max_{\mathbf{x} \in I} |f(\mathbf{x}) - \hat{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}))|$$

However, while now entirely real-valued, this expression is in general too complex for (continuous, real-valued) optimization tools to handle. To reduce complexity, FPTaylor applies a Taylor expansion:

$$f(\mathbf{x}) = f(\mathbf{a}) + \sum_{i=1}^{k} \frac{\partial f}{\partial x_i}(\mathbf{a})(x_i - a_i) + 1/2 \sum_{i,j=1}^{k} \frac{\partial^2 f}{\partial x_i \partial x_j}(\mathbf{p})(x_i - a_i)(x_j - a_j) \quad (3)$$

that allows to approximate an arbitrary sufficiently smooth function by a polynomial expression around some point $\mathbf{a}$. $\mathbf{p}$ is a point which depends on $\mathbf{x}$ and $\mathbf{a}$ and k is the number of input parameters of f. Taylor series define infinite expansions, however, in practice these are terminated after some finite number of terms, and a remainder term soundly bounds (over-estimates) the skipped higher-order terms. In Equation 3 the last term is the remainder.

Applying a first-order Taylor approximation to the abstracted floating-point function $\hat{f}(\mathbf{x}, \mathbf{e}, \mathbf{d})$ around the point $(\mathbf{x}, \mathbf{0}, \mathbf{0})$ we get:

$$\begin{aligned} \hat{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) &= \hat{f}(\mathbf{x}, 0, 0) + \sum_{i=1}^{k} \frac{\partial \hat{f}}{\partial e_i}(\mathbf{x}, 0, 0)(e_i - 0) + \sum_{i=1}^{k} \frac{\partial \hat{f}}{\partial d_i}(\mathbf{x}, 0, 0)(d_i - 0) + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}) \\ R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}) &= 1/2 \sum_{i,j=1}^{2k} \frac{\partial^2 \hat{f}}{\partial y_i \partial y_j}(\mathbf{x}, \mathbf{p}) y_i y_j \end{aligned} \quad (4)$$

where $y_1, \ldots y_{2k}$ range over $e_1, \ldots, e_k, d_1, \ldots, d_k$ respectively. Since $\hat{f}(\mathbf{x}, 0, 0) = f(\mathbf{x})$, one can approximate $|\hat{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) - f(\mathbf{x})|$ by:

$$|\hat{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}) - f(\mathbf{x})| = |\sum_{i=1}^{k} \frac{\partial \hat{f}}{\partial e_i}(\mathbf{x}, 0, 0) e_i + \sum_{i=1}^{k} \frac{\partial \hat{f}}{\partial d_i}(\mathbf{x}, 0, 0) d_i + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d})|$$

(FPTaylor Error)

To compute a concrete roundoff error bound, the above expression is maximized over a given input domain I using rigorous global optimization techniques such as interval arithmetic [25] or branch-and-bound [27].

The above model can be straight-forwardly extended to capture input errors on (particular) variables by increasing the bound on the corresponding error variables e_i and/or d_i. Similarly, library functions for mathematical functions such as *sin*, *cos*, *exp*, ..., are supported by setting the bound on their corresponding error variables according to the specification. Note that since the derivatives of the standard mathematical library functions are well-defined, the partial derivatives in the equations can be immediately computed.

3 Modular Roundoff Error Analysis

In principle, one can apply FPTaylor's approach (Equation FPTaylor Error) directly to programs with procedure calls by inlining them to obtain a single arithmetic expression. This approach, however, results in potentially many re-evaluations of the same or very similar expressions. In this section, we extend FPTaylor's approach to a modular analysis by considering procedure calls explicitly.

At a high-level, our modular error computation is composed of two stages:

1. The *abstraction* stage computes an error specification for each procedure of the input program (Section 3.1 and Section 3.2);
2. The *instantiation* stage instantiates the pre-computed error specifications for each procedure at their call-sites with their appropriate contexts.

Note that each procedure is processed only once in each of these stages, regardless of how often it is called in other procedures.

The main challenge is to compute the error specifications such that they, on one hand, abstract enough over the individual arithmetic operations to provide a benefit for the analysis in terms of performance, and on the other hand do not lose too much accuracy during this abstraction to still provide meaningful results.

A naive way to achieve modularity is to compute, for each procedure, a roundoff error bound as a constant value, and use that in the analysis of the procedure calls. This simple approach is, however, not enough, since in order to analyze a calling procedure, we do not only need to know which new error it contributes, but we also need to bound its effect on already existing errors, i.e. how it propagates them. The situation is even further complicated in the presence of nested procedure calls.

Alternatively, one can attempt to pre-compute only the derivatives from Equation FPTaylor Error and leave all evaluation to the call sites. This approach then effectively amounts to caching of the derivative computations, and does not affect the analysis accuracy, but its performance benefit will be modest as much of the computation effort will still be repeated.

Our approach rests on two observations from the above discussion. We first split the error of a procedure into the propagation of input errors and roundoff errors due to arithmetic operations, following [11]:

$$|f(\mathbf{x}) - \tilde{f}(\tilde{\mathbf{x}})| = |f(\mathbf{x}) - f(\tilde{\mathbf{x}}) + f(\tilde{\mathbf{x}}) - \tilde{f}(\tilde{\mathbf{x}})| \leq \underbrace{|f(\mathbf{x}) - f(\tilde{\mathbf{x}})|}_{\text{propagation error}} + \underbrace{|f(\tilde{\mathbf{x}}) - \tilde{f}(\tilde{\mathbf{x}})|}_{\text{round-off error}}$$

and compute error specifications for each of these errors separately. This allows us to handle the propagation issue from which the naive approach suffers. We employ suitable, though different, Taylor approximations for each of these parts.

Secondly, we pre-evaluate, at the abstraction stage already, part of the resulting Taylor approximations, assuming the context, resp. input specification of each procedure. This results in some accuracy loss when the procedure is called in a context that only requires a narrower range, but saves analysis time.

Naturally, our pre-computed error specifications are only sound if they are called from contexts that satisfy the assumed input specifications. Our implementation checks that this is indeed the case.

Running Example We use the following simple example for explaining and illustrating our technique:

$$g(x) = x^2 \quad \text{where} \quad x \in [0.0, 100.0]$$
$$f(y, z) = g(y) + g(z) \quad \text{where} \quad y \in [10.0, 20.0], z \in [20.0, 80.0] \tag{Running Example}$$

Here, g is being called twice with arguments with different input specifications, but which are both within the allowed range of g of $[0, 100]$. We will consider nested procedure calls in Section 3.4.

Notation We use f, g and h to denote procedures, $\mathbf{x}$, $\mathbf{y}$, $\mathbf{z}$, $\mathbf{w}$ and $\mathbf{t}$ for input parameters and also as input arguments if a procedure contains procedure calls, and $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ for input arguments. Bold symbols are used to represent vectors. Each error specification of a procedure f consists of a roundoff error function denoted by β_f and the propagation error function, denoted by γ_f. We use $\beta_f(\mathbf{a})$ and $\gamma_f(\mathbf{a})$ to denote the evaluation of roundoff and propagation error specifications for a procedure f with $\mathbf{a}$ as the vector of input arguments. We denote the initial errors of input parameters by $\mathbf{u}$, the relative error of a rounding operator by $\mathbf{e}$, and the absolute error by $\mathbf{d}$. The maximum values for the relative and absolute errors are represented by ϵ and δ respectively. We assume ϵ to denote the maximum error for our default precision, i.e. double precision. We will use the notation $\frac{\partial \hat{g}}{\partial e_i}\Big|_{x,0,0}$ to denote $\frac{\partial \hat{g}}{\partial e_i}(x, 0, 0)$ for readability reasons.

3.1 Roundoff Error Abstraction

In this section, we extend FPTaylor's approach with a *rounding model for procedure calls* and show how it can be used to compute roundoff error specifications. Since input errors are handled by the propagation error specification (Section 3.2), we assume here that procedure inputs have no errors.

One of the main challenges of such an extension is that contrary to how the library function calls are handled (see Section 2), there is no given derivative and fixed upper-bound on the rounding error for arbitrary procedure calls.

If g is a procedure with input arguments $\mathbf{a}$ at the call site, and β_g the corresponding roundoff error specification of g, then we extend the IEEE754 rounding model to procedure calls by:

$$\tilde{g}(\mathbf{a}) = g(\mathbf{a}) + \beta_g(\mathbf{a})$$

That is, we abstract the rounding error by an absolute error, whose magnitude is determined by the error specification of f that is a function of the input arguments.

With this, we can proceed to extend the (FPTaylor Error) with procedure calls. Suppose we have a procedure $f(\mathbf{x})$ that contains the procedure calls $g_1(\boldsymbol{a_1}), \dots g_l(\boldsymbol{a_l})$ to procedures $g_1, \dots g_l$, where $\boldsymbol{a_1}, \dots, \boldsymbol{a_l}$ are the input arguments, and $\boldsymbol{\beta_g}(\boldsymbol{a}) = (\beta_{g_1}(\boldsymbol{a_1}), \dots, \beta_{g_l}(\boldsymbol{a_l}))$ is the vector of corresponding roundoff error specifications. Then the roundoff error specification β_f for the procedure $f(\mathbf{x})$ is given by:

$$\begin{aligned}\beta_f &= \hat{f}(\mathbf{x}, \mathbf{e}, \mathbf{d}, \boldsymbol{\beta_g}(\boldsymbol{a})) - f(\mathbf{x}) \\ &= \sum_{i=1}^{k} \left.\frac{\partial \hat{f}}{\partial e_i}\right|_{\mathbf{x},\mathbf{0}} e_i + \sum_{i=1}^{k} \left.\frac{\partial \hat{f}}{\partial d_i}\right|_{\mathbf{x},\mathbf{0}} d_i + \sum_{i=1}^{l} \left.\frac{\partial \hat{f}}{\partial \beta_{g_i}(\boldsymbol{a_i})}\right|_{\mathbf{x},\mathbf{0}} \beta_{g_i}(\boldsymbol{a_i}) + R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}, \boldsymbol{\beta_g}(\mathbf{a})),\end{aligned}$$

where

$$R_2(\mathbf{x}, \mathbf{e}, \mathbf{d}, \boldsymbol{\beta_g}(\mathbf{a})) = 1/2 \sum_{i,j=1}^{2k+l} \left.\frac{\partial^2 \hat{f}}{\partial y_i \partial y_j}\right|_{\mathbf{x},\mathbf{p}} y_i y_j$$

(Roundoff Specification)

where $y_1, \dots y_{2k}$ define $e_1, \dots, e_k, d_1, \dots, d_k$ as before, and $y_{2k+1}, \dots, y_{2k+l}$ correspond to $\beta_{g_1}(\boldsymbol{a_1}), \dots, \beta_{g_l}(\boldsymbol{a_l})$ respectively.

Note that to derive the roundoff error specification for f, the concrete roundoff specifications for g_i are not required, i.e. we treat β_g as a symbolic variable in the same way as e_i and d_i. They are only instantiated at the evaluation phase, at which point all β_gs are available.

Correctness Note that if we were to inline all β_g roundoff specifications in β_f above (potentially recursively), we would reach the same roundoff error formula as given by (FPTaylor Error) for a program where all procedure calls are inlined. Depending on the nesting, one needs higher-order terms of the Taylor expansion to achieve such equivalence.

Running Example To see this, lets consider our (Running Example). In order to compute the roundoff specifications for procedures g and f, we first compute the real-valued abstractions of the floating-point procedures of g and f, (i.e., $\hat{g}(x, e_1, d_1)$ and $\hat{f}(y, z, e_2, \beta_g(y), \beta_g(z))$ respectively) by applying the floating-point rounding model and the rounding model for procedure calls, on the floating-point functions $\tilde{g}$ and $\tilde{f}$:

$$\begin{aligned}&\hat{g}(x, e_1, d_1) = x^2(1 + e_1) + d_1 \\ &\hat{f}(y, z, e_2, \beta_g(y), \beta_g(z)) = (g(y) + \beta_g(y) + g(z) + \beta_g(z))(1 + e_2)\end{aligned}$$

The next step is to compute the roundoff error specifications β_g and β_f. Since g does not contain any procedure calls, then β_g follows the (FPTaylor Error) formula directly:

$$\beta_g = \left.\frac{\partial \hat{g}}{\partial e_1}\right|_{x,0,0} e_1 + \left.\frac{\partial \hat{g}}{\partial d_1}\right|_{x,0,0} d_1$$

Next, we compute the roundoff specification for f:

$$\beta_f = \left.\frac{\partial \hat{f}}{\partial e_2}\right|_{y,z,\mathbf{0}} e_2 + \left.\frac{\partial \hat{f}}{\partial \beta_g(y)}\right|_{y,z,\mathbf{0}} \beta_g(y) + \left.\frac{\partial \hat{f}}{\partial \beta_g(z)}\right|_{y,z,\mathbf{0}} \beta_g(z) + R_2(y,z,e_2,\beta_g(y),\beta_g(z)), \tag{5}$$

where,

$$\begin{aligned} R_2(y,z,e_2,\beta_g(y),\beta_g(z)) = 1/2(& \left.\frac{\partial^2 \hat{f}}{\partial \beta_g(y)\partial e_2}\right|_{y,z,e_2,\beta_g(y),\beta_g(z)} \beta_g(y)e_2 + \\ & \left.\frac{\partial^2 \hat{f}}{\partial \beta_g(z)\partial e_2}\right|_{y,z,e_2,\beta_g(y),\beta_g(z)} \beta_g(z)e_2 + \\ & \left.\frac{\partial^2 \hat{f}}{\partial e_2\partial \beta_g(y)}\right|_{y,z,e_2,\beta_g(y),\beta_g(z)} e_2\beta_g(y) + \\ & \left.\frac{\partial^2 \hat{f}}{\partial e_2\partial \beta_g(z)}\right|_{y,z,e_2,\beta_g(y),\beta_g(z)} e_2\beta_g(z)). \end{aligned}$$

If we replace the β_g functions in Equation 5 by their respective Taylor expansions we reach the following:

$$\begin{aligned} \beta_f = R_2(y,z,e_2,\beta_g(y),\beta_g(z)) \; + & \left.\frac{\partial \hat{f}}{\partial e_2}\right|_{y,z,\mathbf{0}} e_2 \\ + & \left.\frac{\partial \hat{f}}{\partial \beta_g(y)}\right|_{y,z,\mathbf{0}} \underbrace{(\left.\frac{\partial \hat{g}(y)}{\partial e_1}\right|_{y,0,0} e_1 + \left.\frac{\partial \hat{g}(y)}{\partial d_1}\right|_{y,0,0} d_1)}_{\beta_g(y)} \\ + & \left.\frac{\partial \hat{f}}{\partial \beta_g(z)}\right|_{y,z,\mathbf{0}} \underbrace{(\left.\frac{\partial \hat{g}(z)}{\partial e_1}\right|_{z,0,0} e_1 + \left.\frac{\partial \hat{g}(z)}{\partial d_1}\right|_{z,0,0} d_1)}_{\beta_g(z)} \end{aligned} \tag{6}$$

Based on the rounding model for procedure calls, we can deduce that $\frac{\partial \hat{f}}{\partial \beta_g(\mathbf{a})} = \frac{\partial \hat{f}}{\partial \hat{g}(\mathbf{a})}$. If we replace $\frac{\partial \hat{f}}{\partial \beta_g(\mathbf{y})}$ and $\frac{\partial \hat{f}}{\partial \beta_g(\mathbf{z})}$ in Equation 6 and also apply the chain rule (e.g., $\frac{\partial \hat{f}}{\partial \hat{g}(y)} \times \frac{\partial \hat{g}(y)}{\partial e_1} = \frac{\partial \hat{f}}{\partial e_1}$), we reach a formula that is equal to applying the (FPTaylor Error) on

$$\hat{f}_{in} = (y^2(1+e_1) + d1 + z^2(1+e_2) + d_2)(1+e_3),$$

which is the abstraction of the floating-point inlined version of f, i.e. $\tilde{f}_{in}(y,z) = y^2 + z^2$. For simplicity, here we did not expand on the remainder. However, the reasoning is similar.

Partial Evaluation Besides abstraction that happens in the presence of nested function calls due to considering only first-order Taylor expansions and no higher-order terms, we abstract further by evaluating those error terms in (Roundoff Specification) that tend to be small already at the abstraction phase.

Specifically, we evaluate:

- the first-order derivatives w.r.t. absolute errors for subnormals, i.e. d_is,
- the remainder terms that do not contain any β terms themselves.

For this evaluation, we use the input specification of the procedure call. By doing so, we skip the re-instantiation of these small term at the call sites and over-approximate the (small) error of these terms. Since these terms are mostly of higher-order (especially the remainder terms) over-approximating them improves the analysis performance while having small impact on the analysis accuracy.

3.2 Propagation Error Abstraction

The goal is to compute the propagation error specification γ_f for a procedure f as a function of the input parameters, while achieving a reasonably tight error bound. We over-approximate the propagation error, i.e., $\max |f(\mathbf{x}) - f(\tilde{\mathbf{x}})|$ by following the approach proposed in [11] while extending it to also support procedure calls. We first explain how the propagation error specification is computed, when there are no procedure calls (or they are inlined) and then we explain our extension to support procedure calls.

Suppose $u_i, \dots u_k$ are the initial errors of the input variables $x_1, \dots, x_k$, i.e. $\tilde{\mathbf{x}} = \mathbf{x} + \mathbf{u}$. Similarly to the roundoff specification, we apply the Taylor expansion to $f(\tilde{\mathbf{x}})$, but this time we take the derivatives w.r.t. the input variables:

$$f(\tilde{\mathbf{x}}) - f(\mathbf{x}) = \sum_{i=1}^{k} \frac{\partial f}{\partial x_i} u_i + 1/2 \sum_{i,j=1}^{k} \frac{\partial^2 f}{\partial x_i \partial x_j} u_i u_j \tag{7}$$

Now consider the case where $f(\mathbf{x})$ contains procedure calls of $g_1(\boldsymbol{a_1}), \dots g_l(\boldsymbol{a_l})$, where $\boldsymbol{a_1}, \dots, \boldsymbol{a_l}$ are the input arguments and $\boldsymbol{\gamma}(\mathbf{a}) = (\gamma_{g_1}(\boldsymbol{a_1}), \dots, \gamma_{g_l}(\boldsymbol{a_l}))$ is the vector of corresponding propagation error specifications. We compute the propagation error specification for a procedure f as follows:

$$\gamma_f = \sum_{i=1}^{k} \frac{\partial f}{\partial x_i} u_i + \sum_{i=1}^{l} \frac{\partial f}{\partial g_i(\boldsymbol{a_i})} \gamma_{g_i}(\boldsymbol{a_i}) + R_2(\mathbf{x}, \mathbf{u}, \boldsymbol{\gamma}(\mathbf{a}))$$

(Propagation Specification)

where,

$$R_2(\mathbf{x}, \mathbf{u}, \boldsymbol{\gamma}(\mathbf{a})) = 1/2(\sum_{i,j=1}^{k} \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial x_j} u_i u_j + \sum_{i,j=1}^{k,l} \frac{\partial^2 f(\mathbf{x})}{\partial x_i \partial g_j} u_i \gamma_{g_j}(\boldsymbol{a_j}) + \sum_{i,j=1}^{l} \frac{\partial^2 f(\mathbf{x})}{\partial g_i \partial g_j} \gamma_{g_i}(\boldsymbol{a_i}) \gamma_{g_j}(\boldsymbol{a_j}) + \sum_{i,j=1}^{l,k} \frac{\partial^2 f(\mathbf{x})}{\partial g_i \partial x_j} \gamma_{g_i}(\boldsymbol{a_i}) u_j)$$

That is, we compute and add the propagation error of the called procedures by computing the derivatives of the calling procedure w.r.t the called procedures and multiplying such terms by their respective γ function, which is the propagation error of the called procedure. The remainder terms w.r.t called procedures are computed similarly.

Correctness Just as with the roundoff specifications, if we were to replace the γ_{g_i}s by their corresponding formulas in γ_f, we would reach the same propagation error specification as if we had computed it with Equation 7 for a program with all procedures inlined. Again, higher-order Taylor expansion terms may be needed for an equivalence.

Running Example To see this, lets consider our (Running Example). Suppose u_x, u_y, and u_z are the initial errors for procedures g and f respectively. The propagation specifications for g and f are computed as follows:

$$\gamma_g = \frac{\partial g}{\partial x} u_x + 1/2(\frac{\partial^2 g}{\partial x^2} u_x^2) = 2xu_x + u_x^2$$
$$\gamma_f = \frac{\partial f}{\partial g(y)} \gamma_g(y) + \frac{\partial f}{\partial g(z)} \gamma_g(z) = \gamma_g(y) + \gamma_g(z) = 2xu_y + 2yu_z + u_y^2 + u_z^2$$

Note that replacing the γ_g functions with their equivalent Taylor expansion in γ_f and applying the chain rule (e.g., $\frac{\partial f}{\partial g(y)} \times \frac{\partial g(y)}{\partial y} = \frac{\partial f}{\partial y}$), would result in the Taylor expansion of the inlined version of $f(\tilde{\mathbf{x}})$.

Partial Evaluation While computing the propagation specification γ, we evaluate the small error terms of the error specification and add them as constant error terms to the error specification. These small terms are the remainder terms that do not contain any γ terms themselves. Doing so, we skip the re-evaluation of these small terms at the call sites and therefore, speed-up the analysis.

3.3 Instantiation of Error

In the second step of our analysis, we instantiate the propagation and roundoff error specifications of each procedure of the program using its input intervals. In other words, for each procedure, we compute upper bounds for the β and γ error specifications. For the instantiation of the error specifications, one can use different approaches to maximize the error expressions. In Hugo, one can chose between interval arithmetic and a branch-and-bound optimization.

The instantiation of an error specification for a procedure is conducted in a recursive fashion. In order to compute an upper bound on the error for a procedure, we instantiate the error terms of the corresponding error specification using interval analysis. While instantiating the error, we may come across β or γ functions corresponding to the called procedures. In such cases, we fetch the

error specification of these called procedures and instantiate them using the input intervals of the calling procedure.

Note that in the first stage of the analysis and while computing the error specifications, we over-approximated the error by pre-evaluating the smaller terms there and adding them as constants to the error specifications. As a result, in this stage and before instantiating an error specification of a called procedure, we check that the input intervals of input parameters of the called procedure—for which the error specification function is computed—enclose the intervals of input arguments at the call site. This precondition check can also be applied post analysis.

For the (Running Example), instantiating the roundoff error specification of g results in the following evaluated β functions.

$$\beta_g = \epsilon \max |x^2| + \delta,$$
$$\beta_f = \epsilon \max |g(y) + g(z)| + (1 + 2\epsilon) \max |\beta_g(y) + \beta_g(z)|$$

3.4 Handling Nested Procedures

We now explain how our analysis extends beyond the simple case discussed so far, and in particular how it supports the case when a procedure argument is an arithmetic expression or another procedure call.

In such a case, one needs to take into account the roundoff and propagation error of such input arguments. We treat both cases uniformly by observing that arithmetic expression arguments can be refactored into separate procedures, so that we only need to consider nested procedure calls.

We compute the roundoff and propagation error specification of the nested procedure call in a similar fashion as before. Though, while computing the β and γ specifications with nested procedure calls we incorporate their respective β and γ functions in the solution. That is, we take the β function of a nested procedure into account while we create a rounding abstraction for a procedure call. For example, for the procedure call $f(g(\mathbf{a}))$, the rounding model is:

$$\tilde{f}(g(\mathbf{a})) = f(g(\mathbf{a}) + \beta_g(\mathbf{a})) + \beta_f(g(\mathbf{a}) + \beta_g(\mathbf{a}))$$

On the other hand, while computing the propagation error specification of a procedure call such as $f(g(\mathbf{a}))$, instead of multiplying the computed derivatives by their respected initial error, they get multiplied by the respective propagation error specification, i.e. $\gamma_g(\mathbf{a})$.

Example We illustrate how we handle nested procedure calls with a slightly more involved example:

$$\begin{aligned} &g(x) = x^2, \quad \text{where} \quad x \in [0.0, 500.0] \\ &h(y, z) = y + z, \quad \text{where} \quad y \in [10.0, 20.0], z \in [10.0, 20.0] \\ &f(w, t) = g(h(w, t)) \quad \text{where} \quad w \in [12.0, 15.0], t \in [12.0, 15.0] \end{aligned} \tag{8}$$

The roundoff error specification for g is computed as before for our (Running Example) and since h does not contain any procedure calls, β_h is computed straight-forwardly as before.

The abstraction of the floating-point procedure of f is as follows:

$$f(w,t,\beta_g,\beta_h) = g(\chi(w,t)) + \beta_g(\chi(w,t))$$

where,

$$\chi(w,t) = h(w,t) + \beta_h(w,t)$$

Next we compute β_f :

$$\beta_f = \frac{\partial \hat{f}}{\partial \beta_h(w,t)}\beta_h(w,t) + \frac{\partial \hat{f}}{\partial \beta_g(\chi(w,t))}\beta_g(\chi(w,t)) + 1/2\frac{\partial^2 \hat{f}}{\partial \beta_h(w,t)^2}\beta_h^2(w,t) = \frac{\partial g(\chi(w,t))}{\partial (h(w,t)\beta_h(w,t) + \beta_h(w,t))}\beta_h(w,t) + \beta_g(\chi(w,t)) + \beta_h^2(w,t)$$

If β_f is instantiated then we obtain:

$$\beta_f = \max |3\epsilon(w+t)^2 + 3\epsilon^2(w+t)^2 + \epsilon^3(w+t)^2|$$

If we compute the roundoff specification for the inlined version of f i.e., $(w+t)^2$, using the Taylor expansion, however up to the third-order derivative terms, we reach the same error specification as in β_f above.

To compute the propagation error, consider u_x as the initial error for g, u_y and u_z as initial errors for h and u_w and u_t as initial errors in f. The propagation error specification for g is as computed before for (Running Example) and is equal to $2xu_x + u_x^2$. The propagation error specifications of h and f are as follows:

$$\gamma_h = \frac{\partial h}{\partial y}u_y + \frac{\partial h}{\partial z}u_z = u_y + u_z$$

$$\gamma_f = \frac{\partial f}{\partial g(h(w,t))}\gamma_g(h(w,t)) = \gamma_g(h(w,t)) = 2h(w,t)\gamma_h(w,t) + \gamma_h^2(w,t)$$

Therefore,

$$\gamma_f = 2(w+t)(u_w + u_t) + (u_w + u_t)^2$$

The inlined version of f has the same propagation error specification.

4 Implementation

We have implemented our proposed modular error analysis technique in a prototype tool that we call Hugo in the Scala programming language. We did not find it feasible to extend an existing implementation of the symbolic Taylor expression-based approach in FPTaylor [27] (or another tool) to support procedure calls.

We thus opted to re-implement the straight-line code analysis inside the Daisy analysis framework [10] which supports function calls at least in the frontend. We implement our modular approach on top of it and call it Hugo in our evaluation.

Our implementation does not include all of the performance or accuracy optimizations that FPTaylor includes. Specifically, it is *not* our goal to beat existing optimized tools in terms of result accuracy. Rather, our aim is to evaluate the feasibility of a modular roundoff error analysis. We expect that most, if not all, of FPTaylor's optimizations (e.g. detecting constants that can be exactly represented in binary and thus do not incur any roundoff error) to be equally beneficial to Hugo. Nevertheless, our evaluation suggests that our re-implementation is reasonable.

Hugo takes as input a (single) input file that includes all of the procedures. Integrating Hugo into a larger verification framework such as KeY [3] or Frama-C [23] is out of scope of this paper.

In Hugo, we use intervals with arbitrary-precision outer bounds (with outwards rounding) using the GNU MPFR library [15] to represent all computed values, ensuring a sound as well as an efficient implementation. Hugo supports three different procedures to bound the first-order error terms in equations Roundoff Specification and Propagation Specification: standard interval arithmetic, our own implementation of the branch-and-bound algorithm or Gelpia [4], the branch-and-bound solver that FPTaylor uses. However, we have had difficulties to obtain reliable (timely) results from Gelpia. Higher-order terms are evaluated using interval arithmetic.

5 Evaluation

We evaluate our modular roundoff error analysis focusing on the following research questions:

RQ1: What is the trade-off between performance and accuracy of our modular approach?
RQ2: How does the modular approach compare to the state-of-the-art?

5.1 Experimental Setup

We evaluate Hugo on two case studies, `complex` and `matrix`, that reflect a setting where we expect a modular analysis to be beneficial. Each case study consists of a number of procedures; some of these would appear as library functions that are (repeatedly) called by the other procedures. Each procedure consists of arithmetic computations and potentially procedure calls, and has a precondition describing bounds on the permitted input arguments.

Our two case studies are inspired by existing floating-point benchmarks used for verifying the absence of floating-point runtime errors in the KeY verification framework [2]. We adapted the originally object-oriented floating-point Java programs to be purely procedural. We also added additional procedures and

Table 1. Case study statistics

benchmark	# top level procedures	# procedure calls	# arith. ops.	# arith. ops. inlined
matrix	5	15	26	371
matrixXL	6	33	44	911
matrixXS	4	6	17	101
complex	15	152	98	699
complexXL	16	181	127	1107
complexXS	13	136	72	464

procedure calls to reflect a more realistic setting with more code reuse where a modular analysis would be expected to be beneficial. Note that the standard floating-point benchmark set FPBench [9] is not suitable for our evaluation as it consists of only individual procedures.

matrix The `matrix` case study contains library procedures on 3×3 matrices, namely for computing the matrix' determinant and for using this determinant to solve a system of three linear equations with three variables, using Cramer's Rule. Finally, we define a procedure (`solveEquationsVector`) that solves three systems of equations and computes the average of the returned values, representative of application code that uses the results of the systems of equations. See Listing 1.2 in the Appendix for the (partial) `matrix` code.

complex The `complex` case study contains library procedures on complex numbers such as division, reciprocal and radius, as well as procedures that use complex numbers for computing properties of RL circuits. For example, the `radius` procedure uses Pythagoras' theorem to compute the distance to the origin of a point represented by a complex number in the complex plane. The `computeRadiusVector` demonstrates how the `radius` library procedure may be called to compute the radius of a vector of complex numbers. The `approxEnergy` procedure approximates the energy consumption of an RL circuit in 5 time steps.

Listing 1.1 shows partial code of our `complex` case study. The procedure `_add` is a helper procedure that implements an arithmetic expression (and not just a single variable) that is used as argument of a called procedure; see Section 3.4 for how our method modularly incorporates the roundoff and propagation errors resulting from such an expression. For now this refactoring is done manually, but this process can be straight-forwardly automated.

Table 1 gives an overview of the complexity of our case studies in terms of the number of procedures and procedure calls, as well as the number of arithmetic operations in both the inlined and the procedural (original) versions of the code. We inline all procedure calls for comparison with state-of-the-art tools FPTaylor [27] and Daisy [10], since they do not handle them.

Listing 1.1. `complex` case study

```
object complex {
 def _add(rm1: Real): Real = {...}
 def divideRe(re1: Real, im1: Real, re2: Real, im2: Real): Real = {...}
 def divideIm(re1: Real, im1: Real, re2: Real, im2: Real): Real = {...}
 def reciprocalRe(re1: Real, im1: Real): Real = {...}
 def reciprocalIm(re1: Real, im1: Real): Real = { ... }
 def impedanceIm(frequency5: Real, inductance: Real): Real = {...}
 def instantVoltage(maxVoltage: Real, frequency4: Real, time: Real): Real = {...}
 def computeCurrentRe(maxVoltage: Real, frequency3: Real, inductance: Real,
  resistance: Real): Real = {...}
 def computeCurrentIm(maxVoltage: Real, frequency2: Real, inductance: Real,
  resistance: Real): Real = {...}
 def radius(re: Real, im: Real): Real = {...}
 def computeInstantCurrent(frequency1: Real, time: Real, maxVoltage: Real,
  inductance: Real, resistance: Real): Real = {...}

 def approxEnergy(frequency: Real, maxVoltage: Real, inductance: Real,
  resistance: Real): Real = {
  require(((frequency >= 1.0) && (frequency <= 100.0) && (maxVoltage >= 1.0) &&
  (maxVoltage <= 12.0) && (inductance >= 0.001) && (inductance <= 0.004) &&
  (resistance >= 1.0) && (resistance <= 50.0)))

  val t1: Real = 1.0
  val instCurrent1: Real = computeInstantCurrent(frequency, t1, maxVoltage,
       inductance, resistance)
  val instVoltage1: Real = instantVoltage(maxVoltage, frequency, t1)
  val instantPower1: Real = instCurrent1 * instVoltage1
  val t2: Real = _add(t1)
  val instCurrent2: Real = computeInstantCurrent(frequency, t2, maxVoltage,
       inductance, resistance)
  val instVoltage2: Real = instantVoltage(maxVoltage, frequency, t2)
  val instantPower2: Real = instCurrent2 * instVoltage2
  ...
  (0.5 * instantPower1) + (0.5 * instantPower2) + (0.5 * instantPower3) +
   (0.5 * instantPower4) + (0.5 * instantPower5)
 }
 def computeRadiusVector(re: Real, im: Real): Real = {
  require(((re >= 1) && (re <= 2.0) && (im >= 1) && (im <= 2.0)))

  val v1 = radius(re, im)
  val re2 = _add(re)
  val im2 = _add(im)
  val v2 = radius(re2, im2)
  ...
  v1 + v2 + v3 + v4 + v5 + v6 + v7 + v8 + v9 + v10
 }
...
}
```

We additionally create extended and shortened versions of our two case studies, denoted with the suffixes `XL` and `XS`, respectively. The `XL` versions contain one additional procedure that is an extended version of an existing procedure with twice as many procedure calls. In `matrix`, we extend `solveEquationsVector`, and for `complex` we extend `approxEnergy`. In the `XS` version of `matrix`, we remove `solveEquationsVector` and for `complex` we remove the two procedures that were particularly problematic for FPTaylor (it times out), i.e. `computeInstantCurrent` and `approxEnergy`.

We run our experiments on a server with 1.5 TB memory and 4x12 CPU cores at 3 GHz. However, Hugo runs single-threadedly and does not use more than 8GB of memory. We consider a timeout of one hour for analyzing each case study. We assume uniform 64 bit double precision for all floating-point operations.

5.2 RQ1: Accuracy-Performance Trade-off

We first evaluate the effectiveness of our modular approach in terms of the trade-off between the performance of the analysis and the accuracy of the computed error bounds. To do so, we compare Hugo's computed error bounds and performance on our case studies in an ablation study with and without inlining procedures, and by varying the specified input ranges of procedure's parameters.

The accuracy of our modular analysis is influenced by the input range specifications of procedures. Wider input ranges will typically lead to looser error bounds, but will enable procedures to be used in more contexts. We thus define two versions of our case studies: one with tighter input parameter bounds and one with wider ones. We widen the input specifications of the procedures such that for each input interval $[a, b]$ we generate the interval $[a-(b-a), b+(b-a)]$, resulting in a new interval three times as wide as the original interval. We do this widening only for the library procedures (i.e. procedures without procedure calls that are called in other procedures) and only when it is feasible; occasionally it results in division by zero and illegal argument to library procedures errors, and in those cases we only do a more limited widening.

The most accurate error bounds will be computed by inlining all procedure calls at their call site, since by doing so no over-approximation is committed due to procedure summaries. This effectively corresponds to always having the tightest input range bounds at each call site. This comes at the expense of having to potentially repeatedly re-analyze the same procedure many times and thus increase the analysis time.

The results of this experiment are shown in Table 2. The running time (in seconds) is the time for analyzing an entire case study with all procedures and including the check that preconditions are satisfied. We ran each experiment three times and recorded the average runtime in Table 2. We only show the error bounds for procedures containing procedure calls, since those are the ones with over-approximated errors with a modular analysis.

As expected, inlining the procedures for the `matrix` case study results in smaller errors compared to the original procedures. However, the runtime increases more

Table 2. Hugo runtimes (with precondition check) for original procedures and procedures with widened and tightened input intervals

case	procedure	original		inlined proced.		3× interval	
		err	time(s)	error	time(s)	err	time(s)
matrix	solveEquationX	4.14e-15	3.9	1.50e-15	519.0	3.59e-14	3.9
	solveEquationY	4.68e-15		2.12e-15		4.04e-14	
	solveEquationZ	5.16e-15		2.57e-15		4.46e-14	
	solveEquationsVector	4.73e-15		2.88e-16		4.04e-14	
complex	computeCurrentRe	6.12e-10	239.7	-	TO	8.00e-10	239.2
	computeCurrentIm	6.71e-10		-		2.55e-09	
	computeInstantCurrent	3.34e-03		-		3.77e-03	
	approxEnergy	1.00e-01		-		1.13e-01	
	computeRadiusVector	1.47e-11		-		5.84e-11	
	computeDivideVector	2.39e-10		-		2.39e-10	
	computeReciprocalRadiusV.	3.12e-14		-		3.12e-14	

than 130 times. For the inlined version of the `complex` case study the runtime exceeds the timeout and hence no errors are reported.

Widening the input intervals for library procedures in both case studies results—also as expected— in equal or less accurate error bounds, however, the difference is mostly quite small. We thus conclude that our modular analysis is clearly more efficient than the baseline analysis with inlined procedures, and effectively supports procedures with wider input ranges, while producing reasonable error bounds.

5.3 RQ2: Comparison with the State of the Art

We next compare Hugo in terms of performance and accuracy with the state of the art tools FPTaylor [27] and Daisy [10]. We choose FPTaylor as it implements the baseline symbolic Taylor expression approach to roundoff error analysis and has been shown to generally outperform other tools. Additionally, we include Daisy which is also open-source and implements a different, dataflow-based, roundoff error analysis that has been shown to be generally less accurate, but often faster than FPTaylor—it thus represents a different point in the accuracy/performance tradeoff space.

We use Daisy's default settings that implement a dataflow-based roundoff error analysis using interval arithmetic to track ranges and affine arithmetic to track errors. For FPTaylor, we used for the most part the default configuration setting with the following exceptions:

1. We set the option for the improved rounding model to false to reduce running time; we haven't observed a noticeable effect on accuracy for our case studies.
2. We set FPTaylor to compute the maximum possible initial rounding error for all input variables to match Hugo's behavior.
3. We turned the debugging option off to decrease running time.

Table 3. Comparison of Hugo's, Daisy's and FPTaylor's runtimes and computed errors

case study	procedure	Hugo		Daisy		FPTaylor	
		err	time(s)	error	time(s)	err	time(s)
matrix	solveEquationX	4.14e-15	**3.9**	1.07e-15	10.5	**3.83e-16**	539.7
	solveEquationY	4.68e-15		1.55e-15		**6.11e-16**	
	solveEquationZ	5.16e-15		1.90e-15		**4.96e-16**	
	solveEquationsVector	4.73e-15		2.09e-16		**1.83e-16**	
matrixXL	solveEquationsVectorXL	4.78e-15	**5.9**	2.53e-16	24.2	**2.27e-16**	1342.0
matrixXS			**3.5**		4.0		158.9
complex	computeCurrentRe	6.12e-10	**239.7**	4.90e-10	439.1	**9.65e-14**	TO
	computeCurrentIm	6.71e-10		2.46e-11		**2.42e-13**	
	computeInstantCurrent	**3.34e-03**		5.57e+01		-	
	approxEnergy	**1.00e-01**		1.67e+03		-	
	computeRadiusVector	1.47e-11		**6.20e-14**		7.26e-14	
	computeDivideVector	2.39e-10		8.26e-14		**3.85e-14**	
	computeReciprocalRadiusV.	3.12e-14		3.89e-14		**4.67e-15**	
complexXL	approxEnergyXL	**2.00e-01**	**969.3**	3.34e+03	1315.1	-	TO
complexXS			181.7		**13.4**		140.7

Since neither Daisy nor FPTaylor support procedure calls, we inline all procedure calls and call the tools on the fully inlined code. Just like for Hugo, we report running times for computing roundoff error bounds for all procedures in a case study. For Daisy, we prepare one file with all procedures, for FPTaylor we sum up the running times for analyzing each procedure separately, since FPTaylor supports only a single expression per input file (we report the running times for individual procedures in the appending in Table 4). We ran each experiment three times and report the average runtimes.

The results of this experiment are shown in Table 3. As before, we only show the error bounds for procedures containing procedure calls. For the `XL` versions we only report the error of the additional procedure, and for the `XS` version we do not report errors, since this version has a procedure removed.

Hugo is faster than Daisy and FPTaylor on all but the `complexXS` case study, and often significantly so. For `matrix`, Hugo is 2.6x and 138x faster than Daisy and FPTaylor, respectively. For `matrixXL`, the improvements are 4.1x and 227x. These improvements come with error bounds that are within an order of magnitude of those of Daisy and FPTaylor.

FPTaylor is not able to compute errors for two of the longest procedures of `complex`, reporting infinite errors using the default settings. We changed FPTaylor's configuration for these two procedures to be more precise (evaluating second-order terms with a more accurate procedure), however, with this setting FPTaylor timed out, i.e. it took more than one hour for *each* procedure.

For the `complexXS` case study, which does not include the two longest procedures, Hugo is slower than both Daisy and FPTaylor. This is not unexpected, as

Hugo's modular analysis has a certain (implementation) overhead and is thus most effective if there are many procedure calls with a lot of code reuse.

For the full `complex` and `complexXL` case studies, Hugo is faster than Daisy by 1.8x and 1.3x, respectively. We suspect that the improvements for the `XL` version are smaller than for the original one due to inefficiencies in our implementation (e.g. due to missed caching opportunities). That said, Hugo can even produce *tighter error bounds* than Daisy for 4 procedures. This is due to Hugo using a different, generally more accurate, type of analysis. FPTaylor also uses this more accurate analysis, but as our experiments show, the non-modular version does not scale well for larger programs.

Hugo can potentially produce tighter error bounds by applying the branch and bound algorithm instead of the interval analysis for range evaluation. However, for the current set of procedures Hugo was only able to produce (slightly) tighter bounds for four procedures, while taking significantly longer.

In conclusion, compared to FPTaylor and Daisy, Hugo's modular analysis is significantly faster for code with many procedure calls. While Hugo generally does not match the accuracy of existing tools exactly (it fundamentally cannot), our evaluation shows that it nonetheless produces error bounds that are reasonably close to be useful for many (though obviously not all) applications. For the largest procedures, our modular analysis even enables to compute significantly tighter error bounds, resp. any error bounds at all.

6 Related Work

Automated sound static analyses for floating-point arithmetic programs have recently seen much interest. Dataflow-based techniques track floating-point ranges and errors using abstract domains, typically using interval or affine arithmetic, in a forward analysis through a program [18,10,13]. The advantage of these techniques is that they are relatively efficient [10,27]. Alternative approaches construct symbolic constraints that are then solved using global optimization techniques [27,26,24]. These have been shown to produce, in general, tighter error bounds [27,26]. We extend the approach implemented in the tool FPTaylor [27] that applies Taylor approximations to make the optimization problem computationally feasible. The tool PRECiSA generates different constraints (though for addition, subtraction and multiplication they coincide) but also solves them with branch-and-bound techniques. PRECiSA supports function calls in its input programs; it computes the error constraints compositionally but inlines them before evaluation of concrete error bounds and does not perform additional abstractions.

The tool Satire [12] also implements the symbolic Taylor expression-based approach, with additional optimizations for efficiency. Some of these, such as dropping higher-order terms, make the analysis unsound, i.e. the computed error bounds are not guaranteed to be an over-approximation. Our analysis is sound.

The only work that we are aware of that proposes a floating-point roundoff error analysis combining modularity with function summary abstraction [19]

extends the less accurate data-flow-based analysis approach with the zonotope abstract domain. It uses the zonotopes to compute summaries of procedure bodies, which is effectively a first-order—though different—approximation of the roundoff errors that we apply in our approach. Unfortunately, the implementation is not available for comparison. The paper describes an approach to re-compute summaries when the preconditions of the called procedures are not valid at particular call sites. Our implementation currently assumes that preconditions provided by users are sufficiently weak, but a more automated procedure, similar to the above, could be integrated with our approach as well.

Floating-point programs have also been analyzed by deductive verification techniques that are fundamentally modular [14,2]. These tools generate verification conditions that are typically discharged by external SMT solvers or theorem provers, and do not automatically compute (over-approximations of) floating-point roundoff errors. As a consequence, automated roundoff verification is limited to relatively simple computations [2], or requires substantial user interaction in form of annotations [16].

The tools FPTaylor [27], PRECiSa [26], Daisy [5] and real2Float [24] can generate proof certificates that can be independently checked by an interactive theorem prover to ensure the soundness of the computed error bounds. Interactive theorem provers have also been used to prove complex functional properties about floating-point programs, including roundoff errors [7,20]. While they provide an additional level of assurance by their proofs being verified in Coq or HOL, such proofs are manual and require substantial expertise in both theorem proving as well as floating-point arithmetic.

7 Conclusion

We showed how to extend the optimization-based roundoff error analysis for floating-point arithmetic to effectively support (nested) procedure calls. Our evaluation shows that our analysis provides an interesting tradeoff between analysis accuracy and performance, offering substantially smaller analysis times for programs with many procedure calls. Our prototype implementation allows to analyze purely procedural programs; but we expect our approach to be useful in the future as a building block in (existing) modular verification tools.

A Appendix

The code for the `matrix` case study is shown (partially) in Listing 1.2. The runtimes of FPTaylor for individual procedures are shown in Table 4.

References

1. IEEE Standard for Floating-Point Arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019). https://doi.org/10.1109/IEEESTD.2019.8766229

Listing 1.2. `matrix` case study

```
object matrix {
 def determinant(a: Real, b: Real, c: Real, d: Real, e: Real, f: Real, g: Real,
  h: Real, i: Real): Real = {
  require((0.8 <= a) && (a <= 20.4) && (0.8 <= b) && (b <= 75.9) && (0.8 <= c) &&
   (c <= 50.4) && (0.8 <= d) && (d <= 57.3) && (-60.0 <= e) && (e <= 10.2) &&
   (-92.0 <= f) && (f <= 10.2) && (0.8 <= g) && (g <= 93.6) && (-3.6 <= h) &&
   (h <= 10.2) && (-15.3 <= i) && (i <= -2.4))

  a * ((e * i) - (f * h)) - b * ((d * i) - (f * g )) + c * ((d * h) - (e * g))
 }
 // solves a system of equations using the Cramer's rule (variable x)
 def solveEquationX(a1: Real, b1: Real, c1: Real, d1: Real, a2: Real, b2: Real,
  c2: Real, d2: Real, a3: Real, b3: Real, c3: Real, d3: Real): Real = {
  require((19.3 <= a1) && (a1 <= 20.3) && (74.8 <= b1) && (b1 <= 75.8) &&
   (49.3 <= c1) && (c1 <= 50.3) && (0.9 <= d1) && (d1 <= 10.1) && ...))

  val d: Real = determinant(a1, b1, c1, a2, b2, c2, a3, b3, c3)
  val d_x: Real = determinant(d1, b1, c1, d2, b2, c2, d3, b3, c3)
  val x: Real = d_x / d
  x
 }
 // solves a system of equations using the Cramer's rule (variable y)
 def solveEquationY(a1: Real, b1: Real, c1: Real, d1: Real, a2: Real, b2: Real,
  c2: Real, d2: Real, a3: Real, b3: Real, c3: Real, d3: Real): Real = {...}
 // solves a system of equations using the Cramer's rule (variable z)
 def solveEquationZ(a1: Real, b1: Real, c1: Real, d1: Real, a2: Real, b2: Real,
  c2: Real, d2: Real, a3: Real, b3: Real, c3: Real, d3: Real): Real = {...}
// solves three sytems of equations
 def solveEquationsVector(a1: Real, b1: Real, c1: Real, a2: Real, b2: Real, c2: Real,
  a3: Real, b3: Real, c3: Real, aa1: Real, bb1: Real, cc1: Real, aa2: Real,
  bb2: Real, cc2: Real, aa3: Real, bb3: Real, cc3: Real, ...): Real = {
  require((19.49 <= a1) && (a1 <= 19.69) && ...)

  val x: Real = solveEquation_x(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3)
  val y: Real = solveEquation_y(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3)
  val z: Real = solveEquation_z(a1, b1, c1, d1, a2, b2, c2, d2, a3, b3, c3, d3)
  val x2: Real = solveEquation_x(aa1, bb1, cc1, d1, aa2, bb2, cc2, d2, aa3, bb3,
   cc3, d3)
  ...
  (x + y + z + x2 + y2 + z2 + x3 + y3 + z3) / 9.0
 }
}
```

Table 4. Runtimes of FPTaylor for individual procedures

case study	procedure	time (s)
matrix	solveEquationX	56.3
	solveEquationY	51.2
	solveEquationZ	50.8
	solveEquationsVector	380.9
complex	computeCurrentRe	18.8
	computeCurrentIm	22.1
	computeRadiusVector	1.5
	computeDivideVector	53.1
	computeReciprocalRadiusV	31.4

2. Abbasi, R., Schiffl, J., Darulova, E., Ulbrich, M., Ahrendt, W.: Deductive Verification of Floating-Point Java Programs in KeY. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2021). https://doi.org/10.1007/978-3-030-72013-1_13
3. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M. (eds.): Deductive Software Verification - The KeY Book - From Theory to Practice, Lecture Notes in Computer Science, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
4. Baranowski, M.S., Briggs, I.: Global Extrema Locator Parallelization for Interval Arithmetic. https://github.com/soarlab/gelpia (2023), accessed: 20 April 2023
5. Becker, H., Zyuzin, N., Monat, R., Darulova, E., Myreen, M.O., Fox, A.C.J.: A Verified Certificate Checker for Finite-Precision Error Bounds in Coq and HOL4. In: Formal Methods in Computer Aided Design (FMCAD) (2018). https://doi.org/10.23919/FMCAD.2018.8603019
6. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: Programming Language Design and Implementation (PLDI) (2003). https://doi.org/10.1145/781131.781153
7. Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P.: Wave Equation Numerical Resolution: A Comprehensive Mechanized Proof of a C Program. Journal of Automated Reasoning **50**(4) (2013). https://doi.org/10.1007/s10817-012-9255-4
8. Cousot, P., Cousot, R.: Modular Static Program Analysis. In: Compiler Construction (CC) (2002). https://doi.org/10.1007/3-540-45937-5_13
9. Damouche, N., Martel, M., Panchekha, P., Qiu, C., Sanchez-Stern, A., Tatlock, Z.: Toward a Standard Benchmark Format and Suite for Floating-Point Analysis. In: International Workshop on Numerical Software Verification (NSV) (2016). https://doi.org/10.1007/978-3-319-54292-8_6
10. Darulova, E., Izycheva, A., Nasir, F., Ritter, F., Becker, H., Bastian, R.: Daisy - Framework for Analysis and Optimization of Numerical Programs. In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS) (2018). https://doi.org/10.1007/978-3-319-89960-2_15
11. Darulova, E., Kuncak, V.: Towards a Compiler for Reals. ACM Transactions on Programming Languages and Systems (TOPLAS) **39**(2), 1–28 (2017). https://doi.org/10.1145/3014426
12. Das, A., Briggs, I., Gopalakrishnan, G., Krishnamoorthy, S., Panchekha, P.: Scalable yet Rigorous Floating-Point Error Analysis. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC) (2020). https://doi.org/10.1109/SC41405.2020.00055
13. De Dinechin, F., Lauter, C.Q., Melquiond, G.: Assisted Verification of Elementary Functions Using Gappa. In: ACM Symposium on Applied Computing (2006). https://doi.org/10.1145/1141277.1141584
14. Filliâtre, J., Paskevich, A.: Why3 - Where Programs Meet Provers. In: European Symposium on Programming (ESOP) (2013). https://doi.org/10.1007/978-3-642-37036-6_8
15. Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., Zimmermann, P.: MPFR: A multiple-precision binary floating-point library with correct rounding. ACM Trans. Math. Softw. **33**(2), 13 (2007). https://doi.org/10.1145/1236463.1236468
16. Fumex, C., Marché, C., Moy, Y.: Automating the Verification of Floating-Point Programs. In: Verified Software: Theories, Tools, and Experiments (VSTTE) (2017). https://doi.org/10.1007/978-3-319-72308-2_7

17. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.T.: AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In: Symposium on Security and Privacy (SP) (2018). https://doi.org/10.1109/SP.2018.00058
18. Goubault, E., Putot, S.: Static Analysis of Finite Precision Computations. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2011). https://doi.org/10.1007/978-3-642-18275-4_17
19. Goubault, E., Putot, S., Védrine, F.: Modular Static Analysis with Zonotopes. In: Static Analysis Symposium (SAS) (2012). https://doi.org/10.1007/978-3-642-33125-1_5
20. Harrison, J.: Floating Point Verification in HOL Light: The Exponential Function. Formal Methods in System Design **16**(3) (2000). https://doi.org/10.1023/A:1008712907154
21. Izycheva, A., Darulova, E.: On Sound Relative Error Bounds for Floating-Point Arithmetic. In: Formal Methods in Computer Aided Design (FMCAD) (2017). https://doi.org/10.23919/FMCAD.2017.8102236
22. Jeannet, B., Miné, A.: Apron: A Library of Numerical Abstract Domains for Static Analysis. In: Computer Aided Verification (CAV) (2009). https://doi.org/10.1007/978-3-642-02658-4_52
23. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A Software Analysis Perspective. Formal Aspects Comput. **27**(3), 573–609 (2015). https://doi.org/10.1007/s00165-014-0326-7
24. Magron, V., Constantinides, G., Donaldson, A.: Certified Roundoff Error Bounds using Semidefinite Programming. ACM Transactions on Mathematical Software (TOMS) **43**(4), 1–31 (2017). https://doi.org/10.1145/3015465
25. Moore, R.E., Kearfott, R.B., Cloud, M.J.: Introduction to Interval Analysis. Society for Industrial and Applied Mathematics (2009). https://doi.org/10.1137/1.9780898717716
26. Moscato, M., Titolo, L., Dutle, A., Munoz, C.A.: Automatic Estimation of Verified Floating-Point Round-off Errors via Static Analysis. In: International Conference on Computer Safety, Reliability, and Security (SAFECOMP). pp. 213–229 (2017). https://doi.org/10.1007/978-3-319-66266-4_14
27. Solovyev, A., Baranowski, M.S., Briggs, I., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous Estimation of Floating-Point Round-Off Errors with Symbolic Taylor Expansions. ACM Trans. Program. Lang. Syst. **41**(1), 2:1–2:39 (2019). https://doi.org/10.1145/3230733
28. Titolo, L., Feliú, M.A., Moscato, M.M., Muñoz, C.A.: An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs. In: Verification, Model Checking, and Abstract Interpretation (VMCAI) (2018). https://doi.org/10.1007/978-3-319-73721-8_24