A Compiler for Sound Floating-Point Computations using Affine Arithmetic

Joao Rivera
Computer Science
ETH Zurich, Switzerland
hectorr@inf.ethz.ch

Franz Franchetti
Electrical and Computer Engineering
Carnegie Mellon University, USA
franzf@ece.cmu.edu

Markus Püschel
Computer Science
ETH Zurich, Switzerland
pueschel@inf.ethz.ch

***Abstract*—Floating-point arithmetic is extensively used in scientific and engineering applications to approximate real arithmetic. Unfortunately, floating-point arithmetic is not a sound implementation of real arithmetic, i.e., it may produce different results, does not provide error guarantees, and the errors can become arbitrarily large. In this paper, we introduce SafeGen, a source-to-source compiler that rewrites a given C program using floating-point arithmetic to an efficient C program performing the same computation soundly, i.e., it returns an error bound that is guaranteed to contain the correct result of the program if it had been executed in real arithmetic. Equivalently, it gives a precision certificate on the number of correct bits in the result. SafeGen uses affine arithmetic (AA) that keeps accuracy high compared to interval arithmetic by preserving linear correlations between variables. To mitigate its high cost, SafeGen combines a novel form of static analysis to identify these correlations with a flexible policy-based approach for their selection. SafeGen supports SIMD intrinsics in the input and can output SIMD-optimized code. Our results show that SafeGen-generated code is 30–70 times faster than manually rewritten code using AA libraries. Equivalently, SafeGen can offer many more bits of certified accuracy within a reduced time budget.**

***Index Terms*—floating-point arithmetic, source-to-source compiler, affine arithmetic, guaranteed computations.**

I. Introduction

Floating-point arithmetic is natively supported across many processor lines, including in the embedded space, and thus widely used in many scientific and engineering applications. When working with floating-point, developers often think of it as a proxy for real arithmetic and expect close to exact results. Unfortunately, while often true, the result may also deviate arbitrarily far due to accumulating round-off errors: floating-point arithmetic is not a sound implementation of real arithmetic, i.e., there is no accuracy guarantee. In the worst case the result can become meaningless or even render the application unsafe.

Applications in different domains have identified the need for soundness in floating-point computations. For example, the work in [1] derives bounds on the round-off error for a safety-critical system in avionics. An implementation of sound floating-point arithmetic for a collision-avoidance monitor is presented in [2]. In Model Predictive Control, round-off errors need to be accounted for to guarantee stability [3], [4]. The implementation of abstract domains [5] for analyzing floating-point programs needs soundness to guarantee overapproximation [6]–[8]. Further, floating-point soundness is also required in the robustness analysis of neural networks [9]–[11] to prevent round-off errors from invalidating the analysis [12], [13].

Static analysis. Most of the work on floating-point soundness uses static code analysis to automatically derive a bound on the maximum round-off error of a given program [14]–[20]. However, these tools are very limited: they only support straight-line programs with a few dozens of arithmetic operations, most cannot handle loops and conditional branches, and often produce pessimistic bounds since the error bound is derived over an entire range of inputs. The work in [21] improves scalability but is unable to analyze loops and conditionals. Only [14] and [16] offer support for simple loops, but they require additional information regarding the ranges of variables or the bounds become too loose.

Interval arithmetic. An alternative approach to achieve soundness is to rewrite the program to account for ranges. The most cost-effective approach is to use interval arithmetic [22] (IA) that represents the values of each floating-point variable as intervals containing the exact result. Each floating-point operation is then replaced by its respective interval operation implemented either from scratch or using libraries [23]–[26]. In addition, it is also possible to use source-to-source compilers [27] to rewrite the code automatically for IA, thus relieving the work from the developer while guaranteeing soundness. Although IA can be implemented very efficiently on modern processors, it does not maintain correlations between variables. So ranges can only grow, which results in (very) pessimistic bounds for long computations.

Affine arithmetic. To overcome the overestimation of IA, a more precise approach is to rewrite the program using affine arithmetic [28] (AA), which maintains linear dependencies between variables. In AA, the variable ranges are represented by a central value and a linear combination of error symbols that indicate deviations from it. Variables become correlated if they share a symbol, which enables cancellations and thus tighter ranges. The cost is high however since every operation introduces a new symbol, which squares the arithmetic complexity of the program. It is possible to alleviate this problem by limiting the number of symbols as it is done in some libraries [29], [30], which however, reduces precision. Thus, the challenges with using AA for a given program are combining fast execution with high precision, ideally obtained automatically.

Contributions. In this paper, we address these challenges with a source-to-source compiler named SafeGen (**S**ound **Af**fine **Gen**erator) that rewrites a given floating-point program into a corresponding sound, precise, and efficient program that uses AA. Sound means that the transformed program computes output ranges that are guaranteed to contain the result of the original program if it had been executed in real arithmetic. The sizes of the ranges yield a guaranteed precision. We offer the following specific contributions:

- The design and implementation of the SafeGen source-to-source compiler that translates a given program using floating-point arithmetic into an efficient sound version using AA. SafeGen supports SIMD intrinsics in the input and can output SIMD-optimized code.
- An efficient and flexible policy-based approach to the selection of error symbols that avoids the explosion of their number, and thus the high cost of AA.
- A novel form of static analysis to determine which error symbols are likely to cancel out and thus can be used to ultimately improve accuracy. SafeGen uses this analysis as a preprocessing step to inform the variable selection policies. In our benchmarks, this analysis improves accuracy by up to 8 bits while incurring a runtime penalty of only 20%–30% compared to the sound program without the analysis.
- A thorough evaluation of accuracy and performance of SafeGen on various benchmarks. As expected, SafeGen can offer significantly higher certified accuracy in cases where IA fails. Further, SafeGen produces code that is usually 30–70 times faster than the manual use of AA libraries, i.e., it can often deliver 10, 20, or more bits of certified accuracy within a reduced time budget.

II. Background

We introduce necessary background on IA [22], AA [28], and the IGen compiler [27], which is closest related to our work and will serve as one baseline later.

A. Interval Arithmetic (IA)

Interval arithmetic [22] is the original approach for sound floating-point computation. It computes for each operation a bounding interval that overapproximates the error margin. So in essence, operations on floating-point numbers a are replaced by operations on intervals $\overline{a} = [a_\ell, a_u]$, where the exact value is always guaranteed to be contained in $\overline{a}$ (which is the soundness condition). For example, the addition of $\overline{a}$ and $\overline{b}$ can be computed as:

$$\overline{a} + \overline{b} = [\mathrm{RD}(a_\ell + b_\ell), \mathrm{RU}(a_u + b_u)],$$

where $\mathrm{RU}(x)$ rounds towards $+\infty$, and $\mathrm{RD}(x)$ towards $-\infty$.[1]

IA can be implemented efficiently, but can quickly lose too much precision as intervals always grow. The main reason is its inability to track correlation between variables, a phenomenon known as the dependency problem [22]. To illustrate, assume $\overline{a} = [0, 1]$. The simple computation $\overline{b} = \overline{a} - \overline{a}$ will result in $\overline{b} = [-1, 1]$ although the result is in fact equal to zero. The reason is that the interval representation has no information about the origin of the range.

[1]When using IEEE 754 upward rounding, RD can be efficiently implemented using the identity $\mathrm{RD}(x) = -\mathrm{RU}(-x)$.

B. Affine Arithmetic (AA)

Affine arithmetic [28] addresses the loss of correlation in IA by representing variables as affine forms:

$$\widehat{a} = a_0 + \sum_{i=1}^{k} a_i \epsilon_i, \quad \text{with } \epsilon_i \in [-1, 1], \tag{1}$$

where a_0 is the central value of the interval enclosing the affine variable, ϵ_i are symbolic variables called *error symbols* that represent independent deviations from the central value, and the coefficients a_i, $i > 0$, are the magnitudes of those deviations. The interval enclosing an affine expression is computed as follows:

$$\overline{a} = [a_0 - r(\widehat{a}), a_0 + r(\widehat{a})], \quad \text{where } r(\widehat{a}) = \sum_{i=1}^{k} |a_i| \tag{2}$$

is the radius of the affine expression. AA improves over IA when error symbols are shared by multiple variables, since then cancellations become possible and ranges can shrink. In our previous example, assume $\overline{a} = [0, 1]$ is represented as $\widehat{a} = 0.5 + 0.5\epsilon_1$. Computing $\widehat{b} = \widehat{a} - \widehat{a}$ results in $\widehat{b} = 0.5 + 0.5\epsilon_1 - (0.5 + 0.5\epsilon_1) = 0$, which is even the exact result. Most cancellation will be only partial, which still improves over IA.

Affine operations such as addition and subtraction can be easily derived from (1), but again round-off errors have to be taken into account. This is done by adding a new error symbol with each operation [28]. We show addition and multiplication as examples.

Affine addition is computed as:

$$\widehat{a} + \widehat{b} = (a_0 + b_0) + \sum_{i=1}^{k} (a_i + b_i)\epsilon_i + x\epsilon_{k+1}, \tag{3}$$

where x is the coefficient of the new symbol with the accumulated round-off error of the intermediate operations. For affine addition, x is derived as:

$$x = \sum_{i=0}^{k} \mathrm{RU}(a_i + b_i) - \mathrm{RD}(a_i + b_i). \tag{4}$$

The summation is performed using upward rounding to preserve soundness.

For multiplication, the exact result cannot be represented in general and AA overapproximates the coefficient of the new symbol, which takes the form

$$\widehat{a} \cdot \widehat{b} = a_0 b_0 + \sum_{i=1}^{k} (a_0 b_i + b_0 a_i)\epsilon_i + \big(r(\widehat{a}) r(\widehat{b}) + x\big)\epsilon_{k+1}. \tag{5}$$

Similar as for addition, x accumulates the round-off errors of intermediate operation. This is done soundly by computing the products $a_0 b_i$ (and $b_0 a_i$) using upward and downward rounding and accumulating the difference in x. The final addition in $a_0 b_i + b_0 a_i$ is computed using (4).

Assuming that $m \leq k$ symbols are shared between two variables, affine addition and multiplication require $4m+3$ and $10k+4m+3$ floating-point operations, respectively. Note that if we represent a computation as a directed acyclic graph (DAG) of affine operations, each creating a new error symbol, the resulting affine expression in one node only contains the new symbols from its ancestors nodes. Thus, each affine variable only contains a subset of all occurring error symbols.

The number of symbols increases by one with every operation performed. Thus, the ith affine operation requires $O(i)$ operations. If a program originally performs $g(n) = O(f(n))$ arithmetic operations, its complexity, when using AA, becomes $\sum_{i=1}^{g(n)} O(i) = O(\sum_{i=1}^{g(n)} i) = O(f^2(n))$. Thus, AA squares the arithmetic complexity of the original program.

To alleviate the increase in arithmetic complexity, current affine arithmetic libraries [29], [30] limit the number of error symbols in affine variables by fusing two or more symbols into one, thus trading possible correlation for efficiency. Two symbols are soundly fused to a new error symbol as

$$a_i \epsilon_i + a_j \epsilon_j = (|a_i| + |a_j|)\epsilon_{k+1}. \tag{6}$$

When and how to fuse error symbols is also a key element in our work.

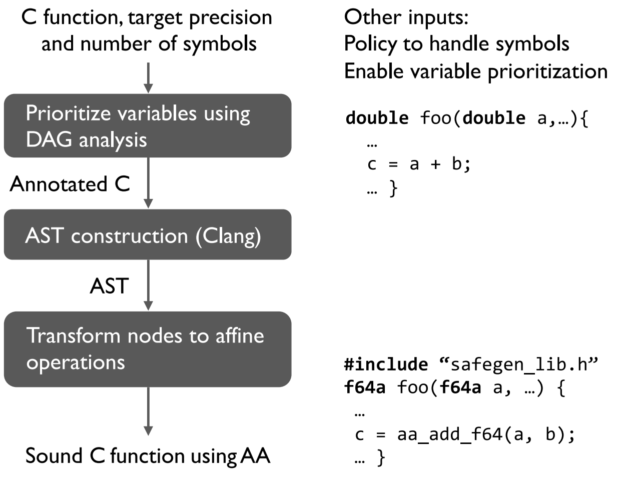
C. IGen Compiler

IGen [27], available at [31], is a source-to-source compiler that translates a C program performing floating-point computations to a sound C program using IA. The compiler uses the Clang LibTooling infrastructure [32] to perform its transformations. IGen can generate SIMD-optimized implementations of IA using either single or double precision floating-point for the endpoints, as well as the higher double-double precision [33]. In addition, the compiler can improve accuracy by using an algorithmic transformation for the reduction pattern in linear-algebra-type programs. In this work, we build on the Clang-based infrastructure of IGen to create our own source-to-source compiler SafeGen that targets AA.

III. SafeGen Overview

SafeGen translates a C program performing floating-point computations to a C program that performs the same computation soundly using AA. Soundness means that during the computation and in the output, the (ranges specified by the) affine expressions are guaranteed to contain the exact results of the original program as if computed using real arithmetic. The sizes of the ranges yield error guarantees.

Fig. 1 shows the high-level architecture of SafeGen. The input is a C function (possibly using SIMD intrinsics) performing floating-point computations, target precisions for the central value (a_0 in (1)) and the coefficients a_i of the affine variables (default is double precision), and the maximal number k of symbols that can be stored in each affine variable. The output is a C function performing the same computation soundly using


C function, target precision and number of symbols
Other inputs:
Policy to handle symbols
Enable variable prioritization
Prioritize variables using DAG analysis
Annotated C
AST construction (Clang)
AST
Transform nodes to affine operations
Sound C function using AA

```
double foo(double a,…){
  …
  c = a + b;
  … }
```

```
#include "safegen_lib.h"
f64a foo(f64a a, …) {
 …
 c = aa_add_f64(a, b);
 … }
```

Fig. 1. High-level architecture of SafeGen.

affine arithmetic, optionally optimized with SIMD intrinsics. Additional input parameters can be set to select how symbols are stored in the internal data structure (*symbol placement policy*) and to configure which symbols to fuse after every affine operation (*symbol fusion policy*) to control the (high) cost of AA. These policies are a key contribution to boost precision and are explained in detail in Section IV.

In the first compilation step, SafeGen optionally performs a novel form of static analysis to identify those affine variables whose error symbols are most likely to cancel out. The output of this stage is the input function annotated with this information. SafeGen uses this information to prioritize the symbols of these variables during the computation, effectively preventing their error symbols from fusion (as in (6)) and ultimately improving accuracy.

In the following steps, SafeGen uses the Clang LibTooling [32] infrastructure to construct the abstract syntax tree (AST) of the input program. Every node performing a floating-point computation is then translated to its corresponding affine operation. In combination with the transformations performed, we provide an efficient library implementing these operations together with the different policies for the management of symbols.

Section IV explains the design and implementation of the compiler in detail. The flexible policy-based approach for the selection of error symbols is explained in Section V. Finally, in Section VI we discuss the static analysis to find the variables to prioritize.

IV. Affine Arithmetic in SafeGen

In this section we describe the SafeGen compiler in detail. We explain the supported operations, data types and the compilation process.

A. Affine Arithmetic Library

As part of SafeGen, we implemented our own library supporting affine arithmetic operations in various precisions and different policies for the selection of error symbols.

Affine types. We support double precision and double-double [33] precision affine data types named `f64a` and `dda`, respectively. The central value of an affine expression is implemented with the respective precision, whereas for both standard double precision is used for the symbol coefficients. Internally, these types store the symbols in an array whose size is fixed and thus limits the number of error symbols. In addition, we also support single precision affine types.

Affine operations. SafeGen supports all basic arithmetic operations including addition, multiplication, division, along with comparison operations, casts and some elementary functions. To guarantee soundness, the round-off error of every intermediate operation is computed to create a new error symbol. The number of symbols is kept bounded through fusion as explained in Section V.

NaN and infinity. NaN and infinity are part of the floating-point standard [34] and can appear during an affine operation. To guarantee soundness, we use the following conventions. If an affine variable contains (as one of the coefficients a_i) a NaN, its value can be anything, including a NaN. If an affine variable contains infinity but no NaNs, its value can be any floating-point number except a NaN. Finally, a positive infinity in the central value but not in the coefficients means that the value is bigger than the maximally representable floating-point number.

B. Source-to-Source Transformations

We use the Clang LibTooling [32] infrastructure for the source-to-source transformations. SafeGen first generates the AST of the input program using Clang. Then, it translates every node in the AST to its corresponding affine representations. In particular, there are two types of nodes that are transformed, namely, declarations (`Decl`) and expressions (`Expr`). A `Decl` node declares variables, functions, fields, etc., and it is transformed by replacing the underlying floating-point type in the declaration to a supported affine type. On the other hand, `Expr` nodes represent arithmetic operations, function calls, constants, etc. This type of node is transformed by replacing the operations by function calls to our affine library. Elementary functions in the input are also transformed accordingly. For example, `sqrt(x)` is transformed to `aa_sqrt_f64(x)`.

Fig. 2 shows an example of a program transformed by SafeGen. As can be seen, the type of the declaration in line 2 of the input (top) is replaced by an appropriate affine type in line 3 of the output (bottom). Further, lines 4–6 show the transformation of the expression `c = a * b + 0.1` to multiple affine operations.

Handling constants. Constants in a program may not be exactly representable as a floating-point value. Thus, we convert constants in a conservative way to an affine expression. When a constant x is encountered, SafeGen assumes that is accurate with at most 1 ulp(x) of error. ulp(x) is the unit in the last place of x which is the gap between its two adjacent floating-point numbers. The floating-point value closest to its value is used for the central value of the new affine variable, and a new fresh symbol is created with magnitude ulp(x). As an example, line 5 in Fig. 2 (bottom) shows the transformation

Example input function:

```
1: double foo(double a, double b) {
2:   double c;          // Decl
3:   c = a * b + 0.1; // Expr
4:   return c;
5: }
```

Output function:

```
1: #include "safegen_lib.h"
2: f64a foo(f64a a, f64a b) {
3:     f64a c, t1, t2;
4:     t1 = aa_mul_f64(a, b);
5:     t2 = aa_set_f64(0.100000000000000002,
                        1.3877787807814457e-17);
6:     c  = aa_add_f64(t1, t2);
7:     return c;
8: }
```

Fig. 2. Transformation of function `foo` to use affine operations.

of the constant `0.1`. We assume that constants that are exactly representable as integers, e.g. `0.0`, are exact and do not create an additional error symbol. In addition, SafeGen also supports the constant folding optimization soundly.

Support of SIMD intrinsics. SafeGen also handles SIMD intrinsics in the input function by replacing them with AA operations. We provide hand-optimized implementations of AA operations for the most common SIMD intrinsics. For others we use the SIMD-to-C compiler provided with IGen [27] as a preprocessing step to generate C code for the intrinsics.

V. Policies to Handle Error Symbols

As explained in Section II, the cost of AA is high due to the explosion in the number of error symbols. To alleviate this problem, we impose a limit on the number k of error symbols in the affine variables in (1), which creates a trade-off between accuracy and performance. Once an affine operation surpasses the limit, some symbols are fused and possible correlation downstream is lost. The challenge therefore is the proper selection of symbols for fusion to reduce this loss. In this section, we propose different *symbol fusion policies* and also *symbol placement policies* that determine the order in which symbols are stored, which impacts the runtime of AA, and the efficiency of SIMD vectorization.

A. Symbols Placement Policies

For affine operations such as addition (3) or multiplication (5), it is necessary to identify common error symbols in the input. To do so efficiently, we propose two different symbol placement policies that we describe next.

Sorted placement policy. In the sorted placement policy, we keep the symbols in each affine variable sorted by their identifiers. When performing an affine operation, we merge the symbol arrays of the inputs into a sorted array. Duplicates are then adjacent and can be combined. At this point, the array length may exceed the imposed limit on the number of symbols, which requires an additional fusion of symbols (see Section V-B below).

Direct-mapped placement policy. With the sorted placement policy common symbols in the inputs can be identified

TABLE I
SYMBOL FUSION POLICIES.

Fusion policy	Description
Random (RP)	Symbols to fuse are randomly selected.
Oldest symbol (OP)	The oldest created symbols are fused.
Smallest value (SP)	Symbols with the smallest absolute value are fused.
Mean threshold (MP)	Symbols below the mean are fused.

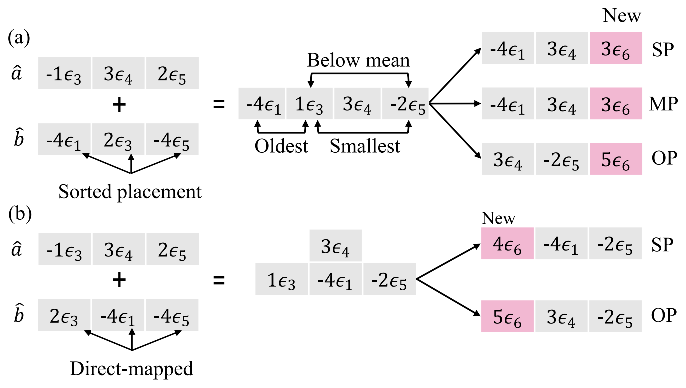


Fig. 3. An artificial example of adding the symbols of two affine variables with $k = 3$ symbols and different fusion policies. Note that usually the coefficients are small floating-point values and not integers; thus also no new round-off error symbols occur here. (a) uses sorted placement policy and (b) uses direct-mapped placement policy. The initial variables are shown at the left, the intermediate result of the addition is in the middle without applying the fusion policy. The final results for SP, MP and OP fusion policies are shown on the right.

efficiently, but the arbitrary placement makes vectorization difficult. Thus, we also consider a placement in unique positions, inspired by the direct-mapped placement policy in caches. Namely, when a symbol s is created, we derive the position in which it is stored based on its identifier by calculating $s \mod k$, where k is the array size. The fusion policies in Section V-B are then designed to avoid conflicts. For example, new symbols from fusion as in (6) obtain identifiers to occupy free slots.

A symbol is now always found in the same position of the array in all variables. This simplifies the implementation for identifying and adding common symbols coefficients of two affine variables to an element-wise masked addition, which can also be done efficiently using SIMD vectors. The downside of this approach is that in the case of two conflicting symbols in the inputs, one always has to be removed through fusion.

B. Symbol Fusion Policies

Since we limit the number k of symbols in affine variables, we propose various policies for selecting symbols to fuse (see (6)) after every affine operation. Assuming that we end with n (which is $\leq 2k+1$) symbols after an operation, $n-k+1$ are fused into a new symbol and $k-1$ are kept. Our policies are summarized in Table I. The random policy (RP) selects randomly the symbols to be fused and serves as baseline.

Oldest symbol fusion policy (OP). This policy fuses the least recently created symbols first. It is implemented by always assigning higher identifiers to new symbols and combines well with the sorted placement policy.

Smallest value fusion policy (SP). Low magnitude symbols are less likely to impact accuracy if they cancel out. Thus SP fuses these symbols first. This policy incurs no overhead when using the direct-mapped placement but incurs cost with sorted placement.

Mean threshold fusion policy (MP). This policy operates like SP but uses the mean of the absolute values of all symbol coefficients as the threshold for fusion, which makes it more efficient than SP when using sorted placement. If not enough symbols are found, OP is used for the rest. For direct-mapped placement, MP is equivalent to SP.

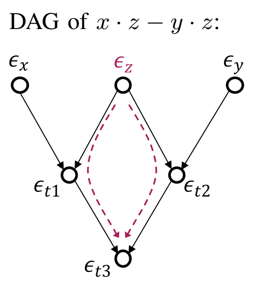
Example. Fig. 3 shows an example of the fusion policies when adding two affine variables with $k = 3$ symbols. (a) uses sorted placement and (b) uses direct-mapped. Different policies usually yield different outcomes. Note that in the direct-mapped placement, conflicts (here ϵ_1 and ϵ_4) have to be resolved according to policy. E.g., for SP, ϵ_4 has to be fused even though overall is not among the two smallest.

Arithmetic cost. The cost of performing affine operations varies depending on the selected fusion policies. In particular, addition and multiplication using the SP policy with direct-mapped placement requires $3k + 2m + 3$ and $13k + 2m + 3$ floating-point operations (including comparisons), respectively, where m is the number of symbols shared by the operands. Our vectorized implementation (for 4 divides k) uses $1.75k$ and $4.25k$ arithmetic intrinsics, respectively, and additional $1.25k$ blend operations.

VI. STATIC ANALYSIS TO PRIORITIZE SYMBOLS

The strength of AA over IA lies in the possible (usually partial) cancellations between joint error symbols. The opportunity for this to happen depends on the structure of the computation DAG associated with the program. In this section, we present a novel static analysis approach that identifies variables whose symbols are likely to cancel out and thus should be protected from fusion to boost accuracy.

As a simple example, consider the computation $x \cdot z - y \cdot z$. The affine variable z is the only one that is used twice and therefore its error symbols can cancel out when meeting in the subtraction operation. This becomes evident in the computation DAG shown in Fig. 4. Each node is tagged with the new error symbol it creates; the input nodes have one symbol each. The symbol ϵ_z propagates through the graph and cancellation occurs in the last node. Fig. 4 thus shows the property that we exploit in our analysis, namely that cancellation in AA can occur if and only if two nodes s and t (here z and t_3) are connected by two paths. In addition, for cancellation to happen, the error symbol ϵ_s (here ϵ_z) has to be protected from fusion along both paths to t. The challenge is therefore to keep symbols with this property while obeying the capacity k. The fusion policies in Section V-B cannot guarantee this since they lack the global knowledge of the DAG.



Example with $k = 2$:

$x = 1 + \epsilon_x$
$y = 1 + \epsilon_y$
$z = 1 + \epsilon_z$

$t_1 = x \cdot z = 1 + \epsilon_z + 2\epsilon_{t1}$
$t_2 = y \cdot z = 1 + \epsilon_z + 2\epsilon_{t2}$

$t_3 = t_1 - t_2 = 2\epsilon_{t1} + 2\epsilon_{t3}$

Fig. 4. DAG of computation $x \cdot z - y \cdot z$ where ϵ_z cancels out in the last node. The dotted lines indicate the propagation of symbol ϵ_z through the DAG till cancellation. On the right, there is an example where this occurs when the limit on the number of symbols is $k = 2$.

In the following, we first formalize the problem of finding symbols to prioritize. Then we explain how to model and solve it as an integer linear program (ILP). Finally, we explain the source-to-source transformation process to prioritize the symbols.

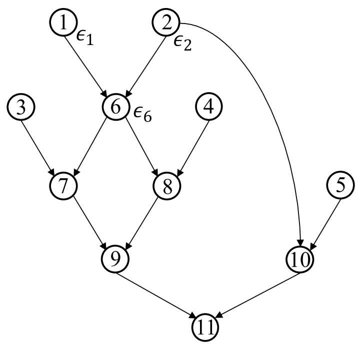
A. Max Reuse Problem

We model a program as a DAG, in which each node represents an AA operation (except for the source nodes that are input variables) and edges represent data dependencies between operations. We assume that each node creates an error symbol that can potentially propagate through the DAG. In addition, each node can propagate $k - 1$ other symbols from ancestor nodes. For example, in Fig. 4, symbol ϵ_z propagates through node t_1 which means that the symbol is not removed by the fusion policy after the computation. In contrast, ϵ_x cannot also propagate through node t_1 since $k = 2$. When a symbol propagates through a node we say that it is *prioritized* in this node. When a symbol arrives to a node from two different parents, we say that there is *reuse* and cancellation may happen. Informally, our static analysis determines the symbols that should be prioritized by each node in order to maximize reuse. In the following we formalize this problem. We start by defining the needed concepts.

Definition 1 (Reuse connection). *Given a directed acyclic graph $G = (V, E)$, a node $s \in V$ is reused at node $t \in V$ if there are two paths p_1, p_2 from s to two distinct parents of t. We call the set $p_1 \cup p_2 \backslash \{s\}$ a reuse connection of (s, t).*

The reuse connection of a pair of nodes (s, t) indicates the nodes in which the symbol ϵ_s must be prioritized to then potentially cancel at node t. As an example, Fig. 5 shows a DAG where nodes $1, 2$, and 6 yield reuse at node 9. Their respective reuse connections are shown in the top table in the figure. In addition, node 2 also yields reuse at node 11 with two possible reuse connections. In general, there are mn reuse connections between two nodes s, t, where m and n are the number of paths from s to the two parents of t, respectively.

Definition 2 (Priority assignment). *A priority assignment π is a map $V \to 2^V$, $s \mapsto A$ with $A \subseteq V$. It assigns to every node s a set of nodes in which the symbol ϵ_s should be prioritized (if it occurs), i.e., protected from fusion.*



(s,t)	Reuse connections
$(1,9)$	$\{6,7,8\}$
$(2,9)$	$\{6,7,8\}$
$(2,11)$	$\{6,7,9,10\}, \{6,8,9,10\}$
$(6,9)$	$\{7,8\}$

s	$\pi_1(s)$	$\pi_2(s)$
1	$\{\}$	$\{6,7,8\}$
2	$\{6,7,8,9,10\}$	$\{6,8,9,10\}$
6	$\{7,8\}$	$\{7,8\}$

Total reuse profit:
$Q_{\pi 1} = (2,9), (2,11), (6,9)$
$Q_{\pi 2} = (1,9), (2,9), (6,9)$
$\rho_{\text{tot}}(\pi_1) = \rho(2) + \rho(2) + \rho(6) = 5$
$\rho_{\text{tot}}(\pi_2) = \rho(1) + \rho(2) + \rho(6) = 5$

Fig. 5. Example of a DAG with its reuse connections (top table) and two possible priority assignments π_1 and π_2 (middle table). The total reuse profit $\rho_{\text{tot}}(\pi)$ of π_1 and π_2 is at the bottom.

Fig. 5 shows two (of many) possible priority assignments for the DAG, namely π_1 and π_2. Since each node can prioritize (not fuse) at most $k - 1$ symbols, a priority assignment may not be feasible if it exceeds this threshold in a node. For example, assignment π_2 in the figure is unfeasible for $k = 3$ because node 8 has to prioritize $3 > k - 1$ symbols, namely 1, 2, and 6.

Reuse profit. The goal of our static analysis is to find a feasible priority assignment that maximizes possible cancellation of symbols, i.e., that maximizes reuse. High magnitude symbols are more likely to improve accuracy if they cancel out and thus it is best to prioritize those. To account for this in the analysis, we introduce the notion of reuse profit as a heuristic to estimate the benefit of reusing a symbol. This heuristic is based on the observation that, in AA, new symbols are likely to have higher magnitudes since they are created by fusing symbols (see (6) and Section V-B) from their ancestors.

Definition 3. *The reuse profit $\rho(s)$ of a node $s \in V$ is the number of its ancestors including s.*

Based on this heuristic notion of reuse profit, we can now estimate the total benefit of using a priority assignment π by summing up profits of all reuses that takes place when prioritizing based on π. Formally:

Definition 4. *The total reuse profit of a priority assignment π is*

$$\rho_{\text{tot}}(\pi) = \sum_{(s,t) \in Q_\pi} \rho(s), \quad \textit{where} \tag{7}$$

$$Q_\pi = \{(s,t) : \pi(s) \supseteq \textit{a reuse connection of } (s,t)\}. \tag{8}$$

Intuitively, if a pair of nodes (s, t) is in Q_π, it means that s is prioritized in a reuse connection from s to t; thus, a reuse of s is guaranteed to happen and total profit increases. Fig. 5 shows the total reuse profit of π_1 and π_2, which is five for both.

Max reuse problem. We now formalize the problem of maximizing possible cancellation in a DAG, as finding a priority assignment π that maximizes the total reuse profit $\rho_{\text{tot}}(\pi)$ under the constraint that each node can prioritize at most $k-1$ symbols, i.e., formally,

$$\text{for all } v \in V: \; |P_v| \leq k-1, \quad \text{where } P_v = \{s : v \in \pi(s)\}.$$

In the example in Fig. 5, and with $k = 2$, π_1 is a feasible assignment of priorities and it is in fact an optimal solution with $\rho_{\text{tot}}(\pi) = 5$.

B. ILP Formulation

To solve the max reuse problem we formulate it as an integer linear program.

Variables. The priority assignment $\pi(s) \subseteq V$ for a node $s \in V$ is encoded as a Boolean vector $\mathbf{p}_s$ of size $|V|$: the nonzero elements of $\mathbf{p}_s$ correspond to the nodes that prioritize s.

The set Q_π of node pairs in (8) is encoded as Boolean vectors $\mathbf{q}_1, \ldots, \mathbf{q}_{|\mathbf{V}|}$ of size $|V|$. A non-zero element at location t in $\mathbf{q}_s$ means that $(s,t) \in Q_\pi$.

Finally, we introduce the Boolean matrix $\mathbf{R}_s$ of size $|V|^2$. The t-th column encodes a reuse connection of (s,t). If there is no such reuse connection then the column is filled with zeros.

ILP program. The max reuse problem can be formulated as an ILP as follows:

$$\begin{aligned} \text{maximize} \quad & \sum_{s \in V} [\rho(s), \rho(s), \ldots, \rho(s)] \mathbf{q}_s \\ \text{subject to} \quad & \sum_{s \in V} \mathbf{p}_s \leq [k-1, k-1, \ldots, k-1]^{\mathrm{T}}, \\ & \mathbf{p}_s = \mathbf{R}_s \circ \mathbf{q}_s \quad \text{for all } s \in V, \end{aligned}$$

where the binary operator $\circ$ indicates a Boolean matrix-vector multiplication, i.e., over the Galois field GF(2).[2]

Intuitively, the objective function that is maximized computes the total reuse profit, and the first constraint guarantees that at most $k-1$ symbols are prioritized per node. The last constraint expresses that the priority vector $\mathbf{p}_s$ is formed as a combination of reuse connections in the columns of $\mathbf{R}_s$ selected by decision variable $\mathbf{q}_s$. This means, that we only prioritize nodes that are in a reuse connection since other nodes do not contribute to the total profit.

Example. For the DAG in Fig. 5, and considering $k = 3$, the matrices $\mathbf{R}_1, \mathbf{R}_4$ have only one non-zero column, and $\mathbf{R}_2$ has two. For readability, we omit columns filled with zeros and the source nodes:

$$\begin{aligned} \mathbf{R_1} &= \begin{bmatrix} \cdots & 1 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}, \\ \mathbf{R_2} &= \begin{bmatrix} \cdots & 1 & 1 & 1 & 0 & 0 & 0 \\ \cdots & 1 & 1 & 0 & 1 & 1 & 0 \end{bmatrix}^{\mathrm{T}}, \\ \mathbf{R_4} &= \begin{bmatrix} \cdots & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}. \end{aligned}$$

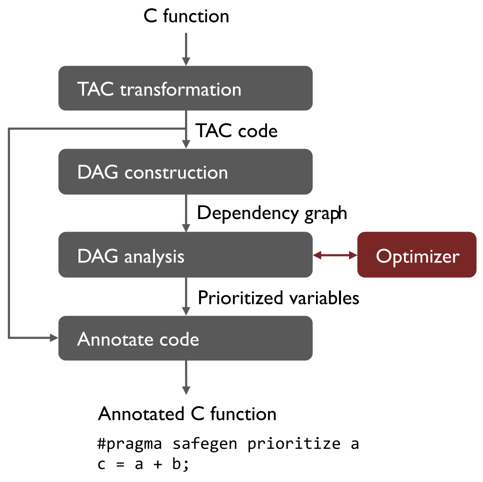
The reuse connections in each column of $\mathbf{R}_s$ are the same that are shown in the topmost table of Fig. 5. Feeding these inputs to an optimizer, e.g., Gurobi [35], yields the following solution for $\mathbf{P}_s$ with a maximum profit $\rho_{\text{tot}} = 5$:

$$\begin{aligned} \mathbf{p_1} &= \begin{bmatrix} \cdots & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}, \\ \mathbf{p_2} &= \begin{bmatrix} \cdots & 1 & 1 & 1 & 1 & 1 & 0 \end{bmatrix}^{\mathrm{T}}, \\ \mathbf{p_4} &= \begin{bmatrix} \cdots & 0 & 1 & 1 & 0 & 0 & 0 \end{bmatrix}^{\mathrm{T}}, \end{aligned}$$

which means that the maximum profit is obtained using the priority assignment $\pi(2) = \{6,7,8,9,10\}$ and $\pi(4) = \{7,8\}$, which corresponds to π_1 in Fig. 5.

Extensions. The current ILP formulation can be extended by assigning to each node a different capacity of symbols that can be prioritized instead of our globally fixed $k-1$. In addition, the model can also be extended to consider two or more reuse connections between two nodes. This can be done by adding additional constraints and variables to consider this case. Finally, input variables with multiple symbols (instead of only one) can be modeled by adding additional source nodes (one per symbol) in the DAG.

[2]Boolean operations can be modeled in ILP by introducing auxiliary variables.



```
#pragma safegen prioritize a
c = a + b;
```

Fig. 6. The architecture of the preprocessing step in SafeGen to prioritize symbols of relevant variables.

```
double foo(double* x, double* y, double* z, int n) {
  for (int i = 0; i < n; i++) {
    #pragma safegen prioritize z[i]
    double t1 = x[i] * z[i];
    double t2 = y[i] * z[i];
    #pragma safegen prioritize_none
    z[i] = t1 - t2;
  }
}
```

Fig. 7. Example of annotated output after prioritizing variables.

C. Code Transformations to Prioritize Variables

Fig. 6 shows the architecture of our preprocessing step to automatically prioritize the symbols of relevant variables. The input is a C program, and the output is the same program annotated with the variables to prioritize in each operation. For example, Fig. 7 shows an example of an annotated output.

Three-address code transformation. The first step is to transform floating-point expressions to three-address code

(TAC) form. In this format, each floating-point operation is computed in one line and temporal variables are introduced to hold their intermediate results. This format allows to prioritize each individual computation with different variables if needed.

DAG construction. We use the data dependency graph analysis in LLVM to construct the DAG of the computation. We simplify the graph in a way that each node represents a floating-point computation, except for the source nodes which are the input variables. The edges are data dependencies between the nodes. Note that we do not keep loop-carried (circular) dependencies since the analysis works on DAGs. Extending the analysis to reason about loop-carried dependencies is a possible further improvement but symbol management would need to become parametric. During DAG construction, we enable debug information and extract the location information of each node in the source file (line and column in file), which is used later to annotate the code.

DAG analysis. After generating the DAG of the computation, we apply our static analysis to find a priority assignment as described above. In principle, we can use this information to instruct an affine operation to avoid fusing symbols in its priority assignment. However, note that the identifiers of such symbols are not known at static time. Thus, this approach requires gathering the identifiers of prioritized symbols at runtime and passing them to the affine operation where they are prioritized. Gathering these symbols may introduce overheads since they are created in different nodes (and thus stored in different variables). To minimize this extra overhead, we prioritize the symbols contained in only one variable for each affine operation (i.e., for each node in a DAG) instead of symbols spread across multiple variables. We use a heuristic to select the variable to prioritize in a node s: from the priority assignment, we inspect the prioritized symbols of s (i.e., $\pi(s)$) and select the one with the highest reuse profit. We choose the variable of the node that generates this symbol as the one to prioritize in s. At the end of this process, this step returns the variables to prioritize for each node in the DAG.

Annotating and transforming the code. As a final step, we identify the position of each node in the source code and insert custom pragmas to instruct SafeGen on the variables to prioritize in each computation. To this end, we search for the nodes at the AST level using the location information extracted earlier. The TAC format facilitates this process since each node appears in a different line in the source code. At this point, the process in Fig. 6 is finished. The resulting annotated code is then processed by the following stages of SafeGen, where each pragma is replaced by a function that prioritizes the symbols of the given variable.

VII. EVALUATION

In this section, we evaluate the accuracy and performance of SafeGen on a set of benchmarks when using the proposed policies as well as the static analysis to prioritize variables. In addition, we compare against prior work on libraries and the IA-based compiler IGen for sound floating-point.

TABLE II
BENCHMARKS.

Label	Description	Base implementation
henon	Henon map	Manually implemented
sor	Jacobi successive over-relaxation	SciMark [36]
luf	LU matrix factorization	SciMark [36]
fgm	Fast gradiant method	FiOrdOs [37]

The key metric we consider is the trade-off between accuracy and performance. Or more precisely, how many bits can be certified for which slowdown compared to the original code. Full AA without limit on the error symbols (Section II-B) will always yield the highest accuracy but at prohibitive cost.

Benchmarks. It is known that AA can only improve over IA for programs where cancellation of symbols can happen (see Section II-B). Thus, we focus on such programs for the evaluation. We use the benchmarks described in Table II, which are known to yield pessimistic results with IA due to the dependency problem. All are implemented in double precision. The Henon map is an iterative algorithm defined as $x_{i+1} = 1 - ax_i^2 + y_i$ and $y_{i+1} = bx_i$. We use $a = 1.05$ and $b = 0.3$ in the experiments as done in [38]. The *sor* and *luf* benchmarks are taken from the benchmark suite SciMark [36] for numerical computing and they were used in [30] to evaluate AA: *sor* is a method for solving a system of linear equations, and *luf* implements the LU matrix factorization. Finally, *fgm* is an efficient implementation of the fast gradient method autogenerated by FiOrdOs [37], which is commonly used as a subroutine in Model Predictive Control, where soundness is important. For all benchmarks, we use SafeGen to generate multiple sound implementations varying the number of symbols, policies, and precisions. The generation of each implementation took less than a second for all considered benchmarks.

Experimental setup. We execute our benchmarks on an Intel Xeon E-2176M CPU with Coffee Lake microarchitecture running at 2.7 GHz, under Ubuntu 20.04 with Linux kernel v4.15. All tests are compiled using gcc 9.3.0 with flags -O3 -march=host -frounding-math. We measure performance by repeating every measurement 30 times on random inputs drawn uniformly in the range $[0, 1]$ and taking the median of the runtimes. The inputs are affine expressions with random central values x_0 and one symbol of length 1 ulp(x_0), where ulp(x_0) is the space between x_0 and the following floating-point number. Intel Turbo Boost is disabled. All tests are run with warm cache.

Accuracy metric. The exact result x of a computation (as if computed using real arithmetic) is guaranteed to be inside the range of the resulting affine expression $\widehat{a}$ (soundness property). We measure error as the base-2 logarithm of the number of floating-point values between the exact result x and any floating-point value inside its range, which is upper bounded by:

$$\text{err}(\widehat{a}) = \log_2 |\{x \in \mathbb{F} : a_0 - r(\widehat{a}) \leq x \leq a_0 + r(\widehat{a})\}| ,$$

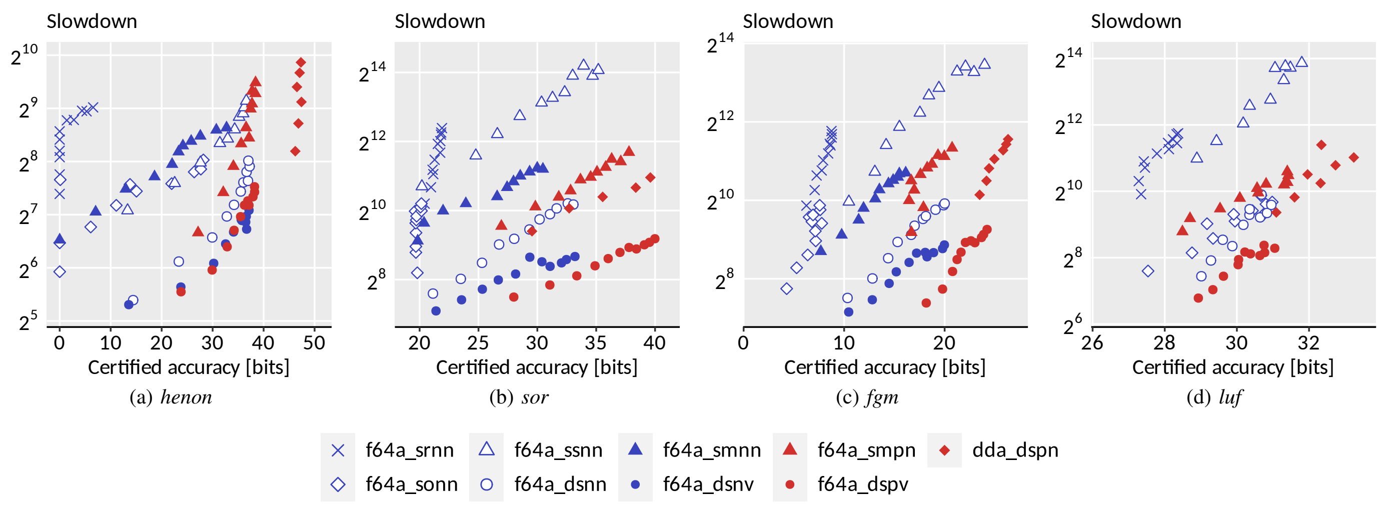


Fig. 8. Certified accuracy and runtime of various configurations of SafeGen when increasing the maximal number of symbols k. Note that (b) and (d) do not start at zero on the x-axis for better display.

where $\mathbb{F}$ is the set of floating-point values and $r(\widehat{a})$ is the radius as defined in (2). This is the same metric used in [27], [39], [40]. In addition, we define accuracy as the number of certified bits in the result calculated as:

$$\operatorname{acc}(\widehat{a}) = p - \operatorname{err}(\widehat{a}),$$

where p is the number of mantissa bits in a given precision, e.g., $p = 53$ in double precision. Intuitively, the accuracy of an affine expression $\widehat{a}$ resulting from a computation, is the minimum number of most-significant bits shared by the mantissa of the exact result with any floating-point value contained in $\widehat{a}$ (assuming same exponent). In our experiments, when a result consists of multiple values, we only consider the one with the lowest accuracy, i.e., we consider worst-case errors. We take the average across all runs. Note that the metric of error that we use relates to the common notion of relative error as explained next. Let y be any floating-point value inside $\widehat{a}$ and assume that x (the exact result) and y share the same exponent in floating-point representation. The relative error is upper-bounded by

$$\left|\frac{x-y}{x}\right| < 2^{\operatorname{err}(\widehat{a})-p+1} = 2^{-\operatorname{acc}(\widehat{a})+1}.$$

A. Accuracy and Performance of SafeGen

Plot navigation. Fig. 8 shows the results for certified accuracy (acc) and runtime of various configurations of SafeGen. The runtime is shown as slowdown factor compared to the original unsound program. Each point shape in a plot represents a different configuration of SafeGen varying the symbol policies, precision, and prioritization. There are many combinations, but several are inferior in performance as explained in Section V, so we show only the most important. We use input matrices of size 10×10 and 20×20 for *sor* and *luf* respectively. Note that two plots do not start at zero on the x-axis for better display.

TABLE III
ACCURACY AND RUNTIME OF DIFFERENT CONFIGURATIONS WITH $k = 40$. ALL CONFIGURATIONS ARE NON-VECTORIZED AND WITHOUT PRIORITY.

	Certified accuracy [bits]				Speedup compared to ss			
Bench.	ss	sm	so	ds	ss	sm	so	ds
henon	36.4	27.6	26.4	36.6	1	1.4	2.2	2.2
sor	33.9	29.3	20.1	31.6	1	5.9	13.0	13.2
fgm	22.1	15.3	6.91	19.1	1	5.5	11.2	10.5
luf	31.3	31.4	30.8	30.4	1	9.8	16.0	18.1

We use the notation f64a-*wxyz* to indicate a given configuration in double precision, where $w \in$ *s,d* specifies the placement policy used (sorted or direct-mapped) and $x \in$ *s,m,o,r* specifies the fusion policy used (smallest, mean, oldest, or random). Finally, $y \in$ *p,n* indicates whether the configuration prioritizes variables or not using our static analysis, and $z \in$ *v,n* whether the generated code is SIMD-vectorized or not. For example, f64a-dspv means that the configuration uses direct-mapped placement policy, shortest fusion policy, prioritization of variables is enabled and the output code is SIMD-vectorized. In addition, the configuration dda-dspn uses double-double precision for the center points in the affine expressions.

Configurations with prioritization are shown in solid red, and their counterparts without priority in solid blue. We use white for the remaining combinations. For each configuration we generate code varying the maximum number of symbols k in the affine variables using $k = 8, 12, \ldots, 48$. For clarity, we ignore the last five points of dda-dspn which are outside the shown plot area. Fig. 8 shows that the certified accuracy and runtime vary significantly between configurations. The best constitute the Pareto-front towards the bottom-right corner.

Placement and fusion policies. We first disregard prioritization (red markers). f64a-srnn (random fusion) has the lowest accuracy and shows the importance of a suitable method to fuse symbols. In many cases, f64-ssnn achieves the best accuracy

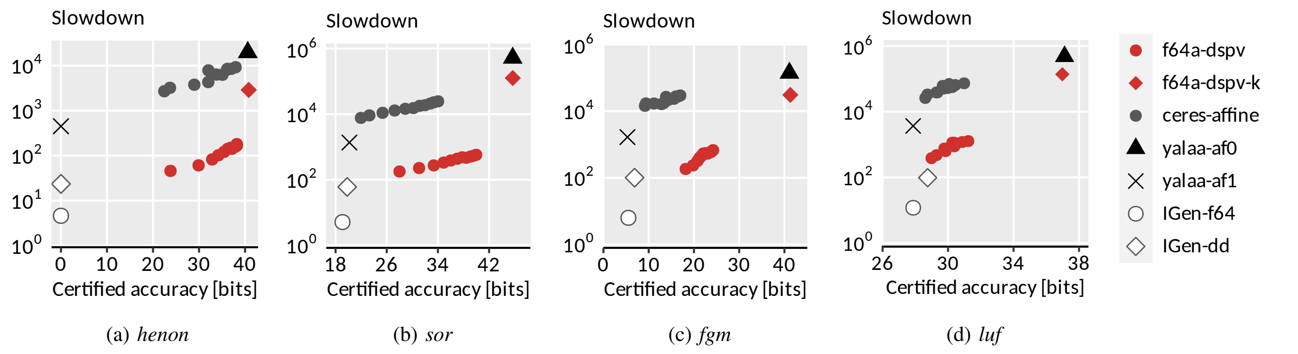


Fig. 9. Certified accuracy and slowdown of SafeGen vs. sound libraries. Note that (b) and (d) do not start at zero on the x-axis for better display.

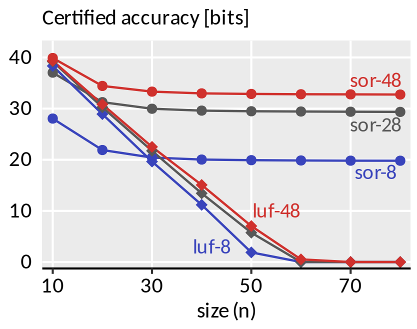


Fig. 10. Certified accuracy of f64a-dspv in *sor* and *luf* benchmarks when increasing the size of the $n \times n$ input matrix. There are three series for each benchmark: using $k = 8$ (blue), $k = 24$ (gray) and $k = 8$ (red).

for a fixed number k of symbols but is slow. Table III shows $k = 40$ and the impact of direct-mapped placement (f64-dsnn): an order of magnitude speed-up at only slight loss of accuracy. Generally speaking, dsxx is the superior combination overall.

Vectorization. We achieve additional speedup by generating SIMD-vectorized code, whose efficiency is due to the direct-mapped placement: f64-dsnv is between 1.2 and 3 times faster than f64-dsnn while achieving the same accuracy.

Effect of variable prioritization. Now we consider our variable prioritization based on static analysis (Section VI) and see that the corresponding red markers make up almost the entire Pareto-optimal front in all cases. f64a-smpn is best among all sorted-placement policies. f64a-dspv is almost always Pareto-optimal and achieves usually 4.5–8 more bits of certified accuracy than its counterpart f64a-dsnv for the same k at only 20–30% performance overhead. Only for *luf* the analysis did not find a feasible prioritization and thus f64a-dspv = f64a-dsnv. Finally, dda-dspn can further boost precision but at about another four times slowdown.

Increasing the input size. Fig. 10 shows the certified accuracy of f64a-dspv in the *sor* and *luf* benchmarks for different sizes of the $n \times n$ input matrix. As can be seen, *sor* scales well by maintaining roughly a constant accuracy when $n > 30$. On the other hand, the accuracy in *luf* decreases till no bit can be certified when $n \geq 60$. In general, the depth D of the computation DAG matters for scaling. For *luf*, the computational depth is $D = O(n)$ whereas *sor* has $D = O(1)$.

B. Comparison with Prior Work

We now compare the accuracy and runtime of SafeGen against prior work on sound floating-point.

Libraries. We compare SafeGen against the two AA libraries Yalaa v0.92 [29] and Ceres [30]. Yalaa is implemented in C++ and offers the affine data types yalaa-af0 and yalaa-af1. The former implements full AA (no limit k), and the latter fixes k to the number of input variables and does not create new symbols. Ceres is a Scala library for certifying numerical results, which limits the number of symbols in variables and uses a symbol fusion policy when this limit is exceeded. Finally, we also compare SafeGen with the sound IA code generated by IGen [27] in both double and double-double precision (IGen-f64 and IGen-dd).

Experimental setup. We use the same experimental setup as before for SafeGen, IGen, and Yalaa. For Ceres, we use the OpenJDK 64-Bit Server JVM supporting Java 1.11, measuring runtime with ScaleMeter [41]. To obtain fair results in the JVM, we perform 100 warm-up runs on all tests to trigger the JIT compiler. Fig. 9 shows the results. For SafeGen and Ceres, we plot different points, again varying $k = 8, 12, \ldots, 48$. For Yalaa, we can only show yalaa-af0 and yalaa-af1 (shown in black). In addition, we show the results of IGen when using double and double-double precision (shown in white). Finally, we introduce SafeGen (f64a-dspv-k) with a large enough k such that no fusion of symbols occurs, thus simulating full AA. The value of this large k is 800, 12K, 6K and 2.5K symbols for *henon*, *sor*, *fgm* and *luf* respectively. The results will be discussed next.

Comparison with affine libraries. SafeGen outperforms the affine libraries and is in the Pareto-front of all benchmarks. In particular, f64a-dspv is 30–70 times faster than ceres-affine when using the same number of symbols k, while also being up to 6 and 9 bits more accurate in *sor* and *fgm*, respectively. Since yalaa-af0 does not limit the number of symbols, it is always the most accurate but also the most expensive by far. Since f64a-dspv-k simulates full AA, it achieves the same accuracy as yalaa-af0 while being 3–6 times faster. Finally, yalaa-af1 does not perform well since it does not create new error symbols.

Comparison with IA. IA is in essence AA with $k = 1$. Thus, it is inherently strictly inferior to AA when using the same floating-point precision. We compare our approach with IA code generated by IGen. For *henon*, IA loses all bits of accuracy even using double-double, while f64a-dspv keeps 23 bits of precision when using only $k = 8$ symbols. For *fgm* IGen only certifies 7 bits while f64a-dspv keeps 18 bits. For *sor* and *luf* the gain is less for $k = 8$ but improves with larger k. This improvement in accuracy over IA comes at the cost of overhead. f64a-dspv with $k = 8$ is $10\times$–$36\times$ slower than IGen-f64a and $2\times$–$3.8\times$ slower than IGen-dd. Compared to the original (unsound) code, f64a-dspv is $48\times$–$185\times$ slower. Note that the cost is rather high, but AA gives proven guarantees while still preserving accuracy in situations where IA fails.

VIII. Related Work

We review the existing work on worst-case error analysis, libraries used for sound floating-point, and techniques to improve accuracy and efficiency of affine arithmetic.

Automatic round-off error analysis. Most of the tools to automatically estimate round-off errors are based on static code analysis. These tools normally use abstract interpretation such as interval or affine arithmetic to derive ranges of variables as well as error bounds [14]–[16], [18]. In other tools, round-off error analysis is approached as an optimization problem. Real2Float [19] is based on semidefinite programming, whereas FPTaylor uses symbolic Taylor expansions and global optimization. Satire [21] uses bound optimization and improves on the scalability of previous approaches. All of these consider all possible inputs, and thus usually obtain very conservative bounds, especially when dealing with conditional branches and loops [16], [42]. In addition, most only support straight-line programs with a few dozens of operations. Satire improves scalability but is unable to handle loops and conditionals.

Other tools such as FPDebug [43] and CGRS [44] perform a form of dynamic analysis to find inputs that cause high floating-point errors. The former uses shadow variables in higher precision and the latter uses a heuristic search method. However, these tools do not provide guarantees on the errors bounds.

Libraries for sound floating-point. Most existing libraries for sound floating-point use IA [23]–[25] and [27] provides a compiler do so automatically. Although IA is efficient it is also very pessimistic due to the dependency problem [22]. AA overcomes this problem to some extent, but only few libraries exist. Ceres [30] implements two special affine data types AffineFloat and SmartFloat in Scala. The former is used to enclose the true result of the computation whereas the latter to determine upper bounds on the round-off error. In addition, it uses a fusion policy to reduce the number of symbols after a predefined threshold. Affine libraries implemented in C++ include Libaffa [45], aaflib [46] and Yalaa [29]. These libraries implement full AA (no k) and are thus very expensive. Only Yalaa gives some flexibility by optionally dedicating a special error symbol to accumulate all new error terms but is inferior to SafeGen-generated code as shown.

Currently, no AA library gives flexibility to define variables with different capacities k in their number of symbols. This can be beneficial for applications that present low reuse of symbols in some parts of the computation (where AA with small k or even IA may suffice) and high reuse in other parts. Assigning a different limit on the number of symbols for each variable may thus improve the overall performance while preserving accuracy and is a possible direction for future work.

Affine arithmetic. In addition to the estimation of round-off errors, AA is used in a range of applications to deal with uncertainties. For example, the work in [47] uses static error analysis based on AA to determine a suitable precision for the implementation of DSP coders. Similarly, [48] uses AA to optimize the bit-widths of fixed-point and floating-point designs. In circuit design, it is used to determine worst-case circuit tolerances [49], [50]. Most of the work on improving the accuracy of AA focus on either improving the model, e.g., by preserving not only linear correlations but also higher order symbols [51], or reducing the overapproximation of non-affine operations such as multiplication [52], [53] and division [54]. All these methods make AA more expensive. To avoid the explosion on the number of errors symbols, others suggest fusion strategies for the symbols [30], [38]. In particular, the strategy in [38] uses information of all active affine variables to estimate the symbols that have less impact on the accuracy based on a heuristic; thus, it may become expensive in practice.

Arbitrary precision libraries. Floating-point computations can benefit from arbitrary precision libraries [55]–[57] for applications that require high precision. Although useful, these libraries do not provide any guarantee on the accuracy of the result and thus are orthogonal to our work. Some interval arithmetic libraries [26], [58] have been developed based on these libraries to increase precision. The overhead of using an arbitrary precision library such as MPFR [55] with 100 decimal digits is between two and three orders of magnitude (see Table III in [55]), which is comparable to the overhead introduced by SafeGen.

IX. Conclusions

Certified floating-point accuracy is a hard problem but can be crucial for safety-critical applications. Most of the prior work has focused on either static analysis or IA. Static analysis is both difficult (and thus to date was mostly focused on simple program expressions) and conservative since it considers ranges of inputs. Rewriting a program using IA yields relatively small performance overheads but also pessimistic bounds since intervals can only grow, ignoring correlations.

Our main contribution is to make using the powerful AA both automatic and more practical by significantly reducing the performance overhead using a dedicated compiler. Two main ideas underlie our approach. First, a direct placement approach for error symbols combined with a suitable strategy for fusing symbols, which in turn also enables efficient SIMD vectorization. Second, a novel form of static analysis to identify error symbols that should be protected from fusion. Thus our automatic approach improves one to two orders of magnitude over manually using prior libraries and offers a wide range of Pareto-optimal trade-offs between certified accuracy and performance.

Artifact Appendix

A. Abstract

Our artifact provides the source code of SafeGen, benchmarks to evaluate its performance and accuracy, and scripts for reproducing main experiments. The artifact is in the form of a virtual machine running Ubuntu 18.04 which provides all required dependencies.

More specifically, our artifact consists of:

1) Source code of SafeGen compiler and library.
2) Source code of our benchmarks.
3) Scripts to set up and automate running the benchmarks saving the results in CSV files.
4) Scripts to generate graphs from the CSV files.

The artifact also contains the source code of the LLVM Project 11.0 with custom modifications.

B. Artifact Check-List (Meta-Information)

- **Program:** SafeGen compiler with benchmarks.
- **Compilation:** We have included a script that builds SafeGen and associated benchmarks using GCC 9.4.
- **Binary:** A modified version of Clang 11.0 is precompiled.
- **Run-time environment:** An Ubuntu-based virtual machine with all necessary software dependencies.
- **Hardware:** x86 machine supporting AVX2.
- **Execution:** We provide scripts to set up, run and plot the benchmarks in this paper. A more detailed description of how to use them is included in README.
- **Output:** Running the scripts yields CSV files containing runtime and accuracy numbers of the benchmarks. In addition, the pareto plots in Fig. 8 are generated.
- **How much disk space required (approximately)?:** 20 GB to support the virtual machine.
- **How much time is needed to prepare workflow?:** Immediately available after importing the virtual machine in VirtualBox.
- **How much time is needed to complete experiments?:** Approximately 1 hour to set up, compile and run all benchmarks.
- **Publicly available?:** Yes.
- **Code license:** BSD 3-Clause License

C. Description

How delivered. As a virtual machine available at: `https://doi.org/10.5281/zenodo.5711307`.

Hardware dependencies. x86 machine with AVX2 support. The results in this paper were obtained using an Intel Xeon E-2176M CPU.

Software dependencies. All software dependencies have been pre-installed in the provided virtual machine. We tested the artifact in VirtualBox 6.1.0. Details on the dependencies pre-installed in the virtual machine can be found in the `README` distributed with the artifact.

D. Installation

Import and access the virtual machine in VirtualBox[3]. The login credentials are the following:

username: cgo2022
password: safegen

Since all dependencies have been pre-installed, the system should be ready once accessing the virtual machine.

E. Experiment Workflow

Once in the virtual machine, open the terminal and navigate to the `benchmarks` directory:

```
$ cd artifact/benchmarks
```

There is a script named `run_benchmarks.py` which builds and runs the benchmarks:

```
$ python3 run_benchmarks.py
```

F. Evaluation and Expected Result

After the experiments finish running, the generated CSV files with the results are saved in `artifact/benchmarks/results` directory. There is one folder for each benchmark (e.g. `henon`, `sor`, `mpc` and `luf`). The script also generates a pareto plot for each benchmark. These plots will be saved as PDF files in `results/_plots`.

G. Notes

We recommend disabling Intel Turbo Boost and Hyper Threading technologies in the host machine to avoid the effects of frequency scaling and resource sharing on the measurements. These technologies can be disabled in the BIOS settings of the machines that have BIOS firmware.

H. Methodology

Information regarding submission, reviewing and badging methodology can be found at the following sites:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

[3]More information on importing virtual machine in VirtualBox can be found at `https://docs.oracle.com/cd/E26217_01/E26796/html/qs-import-vm.html`

References

[1] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "Towards an industrial use of FLUCTUAT on safety-critical avionics software," in *Formal Methods for Industrial Critical Systems (FMICS)*, 2009, pp. 53–69.

[2] F. Franchetti, T. M. Low, S. Mitsch, J. P. Mendoza, L. Gui, A. Phaosawasdi, D. Padua, S. Kar, J. M. F. Moura, M. Franusich, J. Johnson, A. Platzer, and M. M. Veloso, "High-Assurance SPIRAL: End-to-end guarantees for robot and car control," *IEEE Control Systems Magazine*, vol. 37, no. 2, pp. 82–103, 2017.

[3] T. A. Johansen, "Toward Dependable Embedded Model Predictive Control," *IEEE Systems Journal*, vol. 11, no. 2, pp. 1208–1219, 2017.

[4] I. McInerney, E. C. Kerrigan, and G. A. Constantinides, "Modeling Round-off Error in the Fast Gradient Method for Predictive Control," in *Proceedings IEEE Conference on Decision and Control (CDC)*, 2019, pp. 4331–4336.

[5] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings Symposium on Principles of Programming Languages (POPL)*, 1978, pp. 84–96.

[6] D. Cattaruzza, A. Abate, P. Schrammel, and D. Kroening, "Sound numerical computations in abstract acceleration," in *Proceedings Numerical Software Verification (NSV)*, 2017, pp. 38–60.

[7] L. Chen, A. Miné, and P. Cousot, "A sound floating-point polyhedra abstract domain," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2008, pp. 3–18.

[8] D. Monniaux, "The pitfalls of verifying floating-point computations," *ACM Transactions Programming Languages and Systems (TOPLAS)*, vol. 30, no. 3, 2008.

[9] G. Singh, T. Gehr, M. Mirman, M. Püschel, and M. Vechev, "Fast and effective robustness certification," in *Proceedings Proceedings of the 32nd International Conference on Neural Information Processing Systems (NIPS)*, 2018, pp. 10 825–10 836.

[10] G. Singh, T. Gehr, M. Püschel, and M. Vechev, "An abstract domain for certifying neural networks," *Proceedings of the ACM on Programming Languages (PACMPL)*, vol. 3, 2019.

[11] F. Serre, C. Müller, G. Singh, M. Püschel, and M. Vechev, "Scaling polyhedral neural network verification on GPUs," in *Proceedings Proceedings Machine Learning and Systems (MLSys)*, 2021.

[12] K. Jia and M. Rinard, "Exploiting verified neural networks via floating point numerical error," *CoRR*, vol. abs/2003.03021, 2020.

[13] D. Zombori, B. Bánhelyi, T. Csendes, I. Megyeri, and M. Jelasity, "Fooling a complete neural network verifier," in *Proceedings International Conference on Learning Representations (ICLR)*, 2021. [Online]. Available: https://openreview.net/forum?id=4IwieFS44l

[14] E. Goubault and S. Putot, "Static analysis of finite precision computations," in *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2011, pp. 232–247.

[15] M. Daumas and G. Melquiond, "Certification of bounds on expressions involving rounded operators," *ACM Transactions on Mathematical Software (TOMS)*, vol. 37, no. 1, 2010.

[16] E. Darulova and V. Kuncak, "Towards a compiler for reals," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 39, no. 2, 2017.

[17] T. Ramananandro, P. Mountcastle, B. Meister, and R. Lethin, "A unified coq framework for verifying c programs with floating-point computations," in *Certified Programs and Proofs (CPP)*, 2016, pp. 15–26.

[18] L. Titolo, M. A. Feliú, M. Moscato, and C. A. Muñoz, "An abstract interpretation framework for the round-off error analysis of floating-point programs," in *Proceedings Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2018, pp. 516–537.

[19] V. Magron, G. Constantinides, and A. Donaldson, "Certified roundoff error bounds using semidefinite programming," *ACM Transactions on Mathematical Software (TOMS)*, vol. 43, no. 4, 2017.

[20] A. Solovyev, M. S. Baranowski, I. Briggs, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 41, no. 1, 2018.

[21] A. Das, I. Briggs, G. Gopalakrishnan, S. Krishnamoorthy, and P. Panchekha, "Scalable yet rigorous floating-point error analysis," in *Proceedings International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2020, pp. 1–14.

[22] R. E. Moore, "Interval analysis," *Prentice-Hall*, 1966.

[23] H. Brönnimann, G. Melquiond, and S. Pion, "The design of the Boost interval arithmetic library," *Theoretical Computer Science*, vol. 351, no. 1, pp. 111–118, 2006.

[24] M. Lerch, G. Tischler, J. W. V. Gudenberg, W. Hofschuster, and W. Krämer, "FILIB++, a Fast Interval Library Supporting Containment Computations," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 2, pp. 299–324, 2006.

[25] F. Goualard, "Gaol 4.2.0: Not just another interval arithmetic library," https://sourceforge.net/projects/gaol, 2015.

[26] N. Revol and F. Rouillier, "Motivations for an arbitrary precision interval arithmetic and the mpfi library," *Reliable Computing*, vol. 11, no. 4, pp. 275–290, 2005.

[27] J. Rivera, F. Franchetti, and M. Püschel, "An Interval Compiler for Sound Floating-Point Computations," in *Proceedings International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 52–64.

[28] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, 2004.

[29] S. Kiel, "Yalaa: Yet another library for affine arithmetic," *Reliable Computing*, vol. 16, pp. 114–129, 2012.

[30] E. Darulova and V. Kuncak, "Trustworthy numerical computation in scala," in *Proceedings ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, 2011, pp. 325–344.

[31] J. Rivera, "IGen: A Compiler for Sound Floating-Point." [Online]. Available: https://github.com/joaoriverd/IGen

[32] Clang. (2020) Clang libtooling. Available at https://clang.llvm.org/docs/LibTooling.html, version 11.0.0.

[33] Y. Hida, S. Li, and D. Bailey, "Library for double-double and quad-double arithmetic," 2008, https://web.mit.edu/tabbott/Public/quaddouble-debian/qd-2.3.4-old/docs/qd.pdf.

[34] "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2008*, pp. 1–70, 2008.

[35] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021. [Online]. Available: https://www.gurobi.com

[36] R. Pozo and B. R. Miller, "Java SciMark 2.0," 2004. [Online]. Available: http://math.nist.gov/scimark2/about.html

[37] F. Ullmann, "FiOrdOs: A Matlab Toolbox for C-Code Generation for First Order Methods," Master's thesis, ETH Zurich, 2011.

[38] M. Kashiwagi, "An algorithm to reduce the number of dummy variables in affine arithmetic," *Scientific Computing, Computer Arithmetic and Verified Numerical Computations (SCAN)*, 2012.

[39] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock, "Automatically improving accuracy for floating point expressions," in *Conference on Programming Language Design and Implementation (PLDI)*, 2015, pp. 1–11.

[40] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic optimization of floating-point programs with tunable precision," in *Proceedings Conference on Programming Language Design and Implementation (PLDI)*, 2014, pp. 53–64.

[41] A. Prokopec, "ScalaMeter," 2012. [Online]. Available: https://scalameter.github.io

[42] E. Goubault and S. Putot, "Robustness analysis of finite precision implementations," in *Asian Symposium on Programming Languages and Systems (APLAS)*, 2013, pp. 50–57.

[43] F. Benz, A. Hildebrandt, and S. Hack, "A Dynamic Program Analysis to Find Floating-Point Accuracy Problems," in *Proceedings Conference on Programming Language Design and Implementation (PLDI)*, 2012, pp. 453–462.

[44] W.-F. Chiang, G. Gopalakrishnan, Z. Rakamaric, and A. Solovyev, "Efficient Search for Inputs Causing High Floating-Point Errors," *SIGPLAN Not.*, vol. 49, no. 8, pp. 43–52, 2014.

[45] "Libaffa - C++ Affine Arithmetic Library for GNU/Linux," 2004. [Online]. Available: https://www.nongnu.org/libaffa/

[46] "aaflib - An Affine Arithmetic C++ Library," 2010. [Online]. Available: http://aaflib.sourceforge.net/

[47] C. F. Fang, R. A. Rutenbar, M. Püschel, and T. Chen, "Toward Efficient Static Analysis of Finite-Precision Effects in DSP Applications via Affine Arithmetic Modeling," in *Proceedings Annual Design Automation Conference (DAC)*. Association for Computing Machinery, 2003, pp. 496–501.

[48] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer, "Automatic Accuracy-Guaranteed Bit-Width Optimization for Fixed and Floating-Point Systems," in *Proceedings International Conference on Field Programmable Logic and Applications (FPL)*, 2007, pp. 617–620.

[49] T. Ding, R. Trinchero, P. Manfredi, I. S. Stievano, and F. G. Canavero, "How Affine Arithmetic Helps Beat Uncertainties in Electrical Systems," *IEEE Circuits and Systems Magazine*, vol. 15, no. 4, pp. 70–79, 2015.

[50] N. Femia and G. Spagnuolo, "True worst-case circuit tolerance analysis using genetic algorithms and affine arithmetic," *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, vol. 47, no. 9, pp. 1285–1296, 2000.

[51] F. Messine and A. Touhami, "A General Reliable Quadratic Form: An Extension of Affine Arithmetic," *Reliable Computing*, vol. 12, no. 3, pp. 171–192, 2006.

[52] I. Skalna and M. Hladík, "A new algorithm for Chebyshev minimum-error multiplication of reduced affine forms," *Numerical Algorithms*, vol. 76, no. 4, pp. 1131–1152, 2017.

[53] L. Zhang, Y. Zhang, and W. Zhou, "Tradeoff between Approximation Accuracy and Complexity for Range Analysis using Affine Arithmetic," *Journal of Signal Processing Systems*, vol. 61, 2010.

[54] S. Miyajima and M. Kashiwagi, "A dividing method utilizing the best multiplication in affine arithmetic," *IEICE Electronics Express*, vol. 1, no. 7, pp. 176–181, 2004.

[55] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding," *ACM Transaction on Mathematical Software*, vol. 33, no. 2, pp. 13–es, 2007.

[56] F. Bellard, "LibBF Library," https://bellard.org/libbf/, 2020, [Online; accessed 2-Dec-2021].

[57] B. Haible, "CLN - Class Library for Numbers," https://bellard.org/libbf/, [Online; accessed 2-Dec-2021].

[58] F. Johansson, "Arb: efficient arbitrary-precision midpoint-radius interval arithmetic," *IEEE Transactions on Computers*, vol. 66, pp. 1281–1292, 2017.