A Scalable Approach for Automated Precision Analysis

David Boland and George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
London, UK
{david.boland03, g.constantinides}@imperial.ac.uk

ABSTRACT

The freedom over the choice of numerical precision is one of the key factors that can only be exploited throughout the datapath of an FPGA accelerator, providing the ability to trade the accuracy of the final computational result with the silicon area, power, operating frequency, and latency. However, in order to tune the precision used throughout hardware accelerators automatically, a tool is required to verify that the hardware will meet an error or range specification for a given precision. Existing tools to perform this task typically suffer either from a lack of tightness of bounds or require a large execution time when applied to large scale algorithms; in this work, we propose an approach that can both scale to larger examples and obtain tighter bounds, within a smaller execution time, than the existing methods. The approach we describe also provides a user with the ability to trade the quality of bounds with execution time of the procedure, making it suitable within a word-length optimization framework for both small and large-scale algorithms.

We demonstrate the use of our approach on instances of iterative algorithms to solve a system of linear equations. We show that because our approach can track how the relative error decreases with increasing precision, unlike the existing methods, we can use it to create smaller hardware with guaranteed numerical properties. This results in a saving of 25% of the area in comparison to optimizing the precision using competing analytical techniques, whilst requiring a smaller execution time than the these methods, and saving almost 80% of area in comparison to adopting IEEE double precision arithmetic.

Categories and Subject Descriptors

G.1.0 [**General**]: Error analysis

Keywords

Range analysis, Precision analysis, Word-length Optimisation

1. INTRODUCTION

When designing custom accelerators for an algorithm, the majority of effort is placed into the architecture design, with the aim of maximizing the performance obtained from a fixed silicon or power budget. However, if one were to also to attempt to optimize the precision used throughout the hardware design to meet an algorithm designer's output specification, it is possible to save a significant amount of silicon area; this in turn could affect the performance achievable. Such a specification may take the form of ensuring the result lies within some desired output range or meeting a threshold on the maximum error introduced by the use of finite precision arithmetic (floating point or fixed point). While performance improvements are also available for general purpose processors and GPUs by moving from double to single precision, provided a proof can be created that ensures this move is valid, these devices are largely restricted to these two IEEE standards, unlike FPGAs which have the freedom to implement any precision. Unfortunately, it is hard to choose a number system which can guarantee any range or error specification can be met. As a result, simulation is sometimes used to justify that a customized precision is valid [1, 2], but in this case the resulting implementation comes with no guarantees, or a standard precision is selected to make a 'fair' comparison against a software implementation [3, 4], ignoring the potential performance improvements. To create a custom accelerator that benefits from this freedom with guaranteed numerical properties, we require a tool to verify such an algorithm specification is met. While no tool exists which can provide such verification for an arbitrary algorithm over a set of input ranges and precision in a tractable time, there remains an interest in developing analytical tools that can guarantee a certain precision is sufficient to satisfy the specification.

There are two important metrics for any such tool: quality of bounds and scalability. A tool that can find tight bounds will generally be able to guarantee that hardware will satisfy the chosen output criterion with a lower precision than a tool which calculates wide bounds. We note that any approach that does not calculate a bound, such as simulation, cannot be used to make hardware with any such guarantees. Scalability is important because rounding errors arise from every operation and the tool must be able to keep track of all the potential errors that can arise throughout an algorithm and be able to calculate bounds, taking all these errors into account, in a reasonable time to be of any use. Furthermore, a scalable and fast tool is important given that it is likely to be used repeatedly within some compilation and synthesis framework to find the best precision for a custom hardware design.

Existing work in this field has typically focused on either one of these goals, meaning the methods that can calculate tight bounds are restricted to trivial computational kernels [5–7], or the restricted LTI domain [8], and the methods which can scale to larger problems do not find tight bounds and consequently limit the potential hardware optimizations [9, 10]. This work describes a tool to calculate bounds for any algorithm consisting of algebraic operations or elementary functions and can be translated into straight-line single static assignment (SSA) form by performing a static analysis of input code. This means our approach can support algorithms consisting of loops with known bounds at compile-time, by loop unrolling, and conditional statements by analyzing branches separately. The bounds our approach finds approach the quality of the time consuming methods in a significantly shorter execution time, and our method is capable of scaling to larger examples than all but the simplest existing method. Furthermore, unlike existing approaches, our approach also allows a user a finer level of control over the trade-off between execution time and quality of bounds.

We examine the use of our method on two iterative algorithms to solve a system of linear equations: successive-over-relaxation and MINRES [11]. We demonstrate how our approach obtains better bounds than existing analytical techniques, and this provides the ability to tune the precision used in a hardware acceleration of an algorithm, which we show to translate into a significant reduction in silicon area. Furthermore, we highlight that our approach is much more scalable than most of the existing approaches, enabling the analysis of much larger algorithms than these examples.

This paper first describes background into the main existing approaches to calculate bounds in Section 2 and how to model floating point round-off error in Section 3. We follow this with an analysis of how this model of error affects the scalability of the methods to find bounds in Section 4, before we describe how our new method addresses problems in the existing methods in order to obtain a control over the execution time in Section 5. We follow this with a description of our algorithm to compute tight bounds in Section 6, after which we demonstrate the benefits of our algorithm relative to alternate methods in Section 7 and draw conclusions in Section 8.

2. BACKGROUND

While there is a large amount of literature in the field of word-length optimization focused on developing tools to optimize the potential area, frequency and power benefits to satisfy a design specification [12], the number of basic analysis techniques is limited. In this section, we elaborate on these methods: simulation, interval arithmetic, Taylor forms, and Handelman representations, so as to compare these with our new suggested approach in later sections.

The most straightforward way to estimate an error is through simulation. The aim of any simulation-based approach is to find the inputs that will cause the extreme ranges of the data set. Unfortunately, the size of the search space for the inputs will generally be too large to explore exhaustively, meaning that even advanced simulation methods, for example statistical profiling [13], can only estimate the error because corner cases can be missed.

To calculate true bounds for general algorithms, the most well known analytical approach is interval arithmetic (IA) [14]. Interval arithmetic represents every value as lying within some interval: $[x_1, x_2]$, where x_1 and x_2 are the lower and upper bounds respectively. The intervals are then propagated through the computation according to basic rules, given in (1) (provided the range of a divisor does not include zero), which calculate at each stage the new worst case bound.

$$\begin{aligned}[x_1,x_2]\odot[y_1,y_2] =&[\min(x_1\odot y_1, x_2\odot y_2, x_1\odot y_2, x_2\odot y_1), \\ &\max(x_1\odot y_1, x_2\odot y_2, x_1\odot y_2, x_2\odot y_1)], \\ &\odot\in\{+,-,\bullet,\div\}.\end{aligned} \tag{1}$$

However, interval arithmetic suffers from the so-called dependency problem, where if the same variable is used twice, information is lost. A trivial example is the following: for a variable x which lies in the interval $[0,1]$, perform the operation $x-x$. The interval should be $[0,0]$, but the result using interval arithmetic would be $[-1,1]$. Several simple examples can demonstrate how this problem may cause bounds that are significantly wider than the tightest bounds [14]. As a result of these problems, there is an active community of researchers in *robust computing* who have developed ways to mitigate this problem [15].

One simple method to reduce the effect of the dependency problem is to split intervals into the union of much smaller intervals (2), and evaluate each of these independently, because dependencies between smaller intervals have a reduced effect of widening bounds (3). While effective, the number of intervals that must be evaluated grows as of $\Theta(n_o n_s^{n_{sv}})$, where n_o is the number of operations, n_s is the maximum number of interval splits of the n_{sv} variables that are split. This means this technique is not computationally scalable. Even though some methods have been proposed to choose these splits more wisely, such as the GAPPA tool which uses a set of in-built and user-defined 'hints' [16], or the work by Nicolici et al. which uses Satisfiability-Modulo Theories (SMT) [5], the quality of bounds these approaches can obtain are heavily limited by the run-time. As a result of the slow run-time, the authors of the latter paper have attempted to improve scalability using vector approximations [17], and by adding additional constraints informing the solver to ignore certain regions [18], but both sacrifice the tightness of bounds and because they use interval splitting, they will always suffer if there are many input variables, as we will see in Section 7.

$$[x_{lower}, x_{upper}] = \bigcup_{i=1}^{n}[x_{ilower}, x_{iupper}]. \tag{2}$$

$$f([x_{lower}, x_{upper}]) \supseteq \bigcup_{i=1}^{n} f([x_{ilower}, x_{iupper}]). \tag{3}$$

Alternatively, more recently a set of methods that can be loosely grouped together under the name of Taylor forms, analyzed in detail [19], have gained popularity. These use a polynomial representation of the error terms with the intuition that this allows cancellation of dependencies; in the case of the earlier example, use of these approaches would result in $x-x=[0,0]$ as desired. Unfortunately, a polynomial with second order terms or higher contains dependencies within the polynomial, and finding optimal bounds for a multivariate polynomial has been shown to be NP-hard [20].

The most well-known of these Taylor forms, affine arithmetic (AA) [21], avoids this problem by restricting polynomials to first order, ensuring the polynomial contains no dependencies, meaning applying interval arithmetic to the final polynomial can find the ideal bounds. However, to ensure the polynomial only contains first order terms and still bounds the potential range, it must approximate bounds on any higher order terms which are created, using a new variable. Unfortunately, any difference between the true range of the higher order terms and their approximation will result in wider bounds, whilst the dependency information between the higher order and lower order terms is lost. In addition, the added variables can affect the scalability, as will be seen in Section 7.

To minimize both these effects, the more general method by Berz et al., named Taylor methods with Interval Remainder bounds (TwIR) [22] represents range using the form (T_ρ, I_ρ), where T_ρ is a polynomial consisting of all the terms that are less than or equal to an order, ρ, chosen by a user, and I_ρ is an interval which bounds the remaining higher order terms. Using a single interval I_ρ avoids continually introducing new variables to bound higher order terms whenever they are created, as performed by affine arithmetic. Furthermore, the choice of maximum order gives a user some form of control over the trade-off between execution time and quality of bounds, because by retaining higher orders, the higher order dependencies can cancel. Unfortunately, using the single interval I_ρ

means any operations involving I_ρ suffer from the same dependency problem as interval arithmetic, and in addition, finding the final bounds of the polynomial T_ρ still relies on interval arithmetic, and this is a problem once again because dependencies will exist between the high and lower order terms of the polynomial.

Recently, a novel approach has been suggested which is capable of reducing the effect of widening of bounds due to dependencies in a multivariate polynomial by bounding the polynomial using Handelman representations [6]. This approach obtained superior bounds to those achievable by interval arithmetic, and also showed significant advantages in the case of division, by using a rational expression instead of a polynomial. Unfortunately, the approach was limited to small problems because it made no effort to control the size of the polynomial, as will be discussed in Section 4. Furthermore, it was limited to algorithms consisting of the basic algebraic operations $\{+, -, \bullet, \div\}$, unlike Taylor forms which have the added advantage of being applicable to more complex functions by applying polynomial approximation techniques [23].

In this work, we aim to develop a new approach which can find bounds approaching those of the method using Handelman representations, whilst scaling to much larger problems. We also demonstrate how to allow a user to have a strong control over the trade-off between execution time and the tightness of bounds, and show how our new method can be used to create superior hardware designs.

3. FLOATING POINT MODEL OF ERROR

In this work, we have elected to use the multiplicative model of floating point error used throughout numerical analysis literature. This represents the closest radix-2 floating-point approximation $\hat{x}$ to any real value x as (4) [24], where η represents the number of mantissa bits used and δ represents the small unknown roundoff error. It is similarly possible to specify that the radix-2 floating-point result of any scalar operation $(\odot)$ is bounded as in (5), provided the exponent is sufficiently large to span the range of the result. This allows us to create a polynomial representing the potential range of a variable, as shown for a simple example in Table 1. This is the only general model that is useful for worst-case floating-point error modelling, with the exception of a bit-blasting of the floating-point logic. More sophisticated models, such as detecting if one operand is a power of two and setting the error to zero in this case will have minimal impact and severely complicate modelling [25]. We note, however, that provided the chosen model, or a similar model for fixed-point error, can be expressed using polynomials or rational functions, the algorithms described in this work are still applicable.

$$\hat{x} = x(1+\delta) \quad (|\delta| \le \Delta, \text{where } \Delta = 2^{-\eta}). \tag{4}$$

$$\widehat{x \odot y} = (x \odot y)(1+\delta). \tag{5}$$

Table 1: Construction of polynomials

x, y are inputs, Δ is the error bound determined by the precision, so that $\lvert\delta_i\rvert \le \Delta$	
Input Code	Polynomial Representation of Variable Value
$a = x \bullet y;$	$a = xy(1+\delta_1)$
$b = a \bullet a;$	$b = (xy(1+\delta_1))^2(1+\delta_2)$
$c = b - a;$	$c = [(xy(1+\delta_1))^2(1+\delta_2) - xy(1+\delta_1)](1+\delta_3)$

The value of this representation is that it allows us to apply the symbolic analysis techniques described in the Section 2 to calculate bounds on the range or relative error of any variable in the input code. In the following section, we analyze the limitations of these techniques when using this model of error, before describing how we overcome them in Section 5.

4. SCALABILITY OF EXISTING APPROACHES

In this section, we analyze the worst case execution time of these approaches in terms of the number of operations in the code, denoted n_o, and the number of input variables n. Throughout this paper, we define a term as a product of the variables raised to some integer powers, *e.g.* $\delta_1\delta_2\delta_3^2$, and a monomial as a term multiplied by some real coefficient, *e.g.* $10\delta_1\delta_2\delta_3^2$.

Using interval arithmetic, an interval evaluation is performed to find an initial bound on the result of every floating point operation in the code, after which an extra interval evaluation is used to find the bounds taking into account the floating point model of error, where the latter is computed by replacing each of the variables δ_i with an interval. Consequently, the worst case execution time is proportional to the number of operations, or of $O(n_o)$, or as we have mentioned in Section 2, in the case of applying interval splitting, it will then scale as of $O(n_o n_s^{n_{sv}})$

At the other extreme lies the approach using Handelman representations. In this approach, the polynomials representing the variable value, as in Table 1, are first expressed as a sum of monomials in which each term appears at most once. The number of monomials in the polynomials quickly become large, as an example, the floating-point error model for an algorithm consisting of a series of floating-point multiplications (Figure 1), will have a polynomial representation as in (6); the number of monomials in this polynomial, when expanded into canonical form, is exponential in n_o.

```
$y = 1$
for $i = 1; i \le n_o; i++$ do
    $y = y \bullet x[i]$
end for
```

Figure 1: Code to calculate the product of vector elements.

$$x_1 x_2 \ldots x_{n_o}(1+\delta_1)(1+\delta_2)\ldots(1+\delta_{n_o}) \tag{6}$$

As we indicated in Section 2, Taylor forms control this growth in the polynomial representing the variable value by only retaining monomials up to a given order. However, the lack of similar control over the number of variables ensures that the size of this polynomial remains unbounded. This is a problem when using the floating point model of error of Section 3, because a variable (δ_i) bounding the finite precision error will be added after every operation, and hence the total number of variables for an intermediate polynomial bounding the range of a variable in an algorithm, when including the n input variables, is of $O(n_o + n)$. This means that for a Taylor form limited to a maximum order ρ, the number of monomials in any polynomial representing the range of an intermediate variable can grow as of $O\binom{n_o+n+\rho}{\rho}$, so bounding the higher order terms will require of $O\binom{n_o+n+\rho}{\rho}$ interval evaluations. Altogether, the execution time to do this for n_o operations grows as $O(n_o \times \binom{n_o+n+\rho}{\rho})$.

This analysis makes it clear that only interval arithmetic has an execution time that scales well with the number of floating point operations. However, as shown in [6, 21, 22], interval arithmetic is unable to find tight bounds due to dependencies between variables. The use of interval splitting in conjunction with any of these approaches allows some trade-off between quality of bounds and execution time, but this scales poorly in the number of variables. Our aim is to create an approach where the execution time of grows in proportion with the code size $O(n_o)$ and provides a user a very flexible level of control over the trade-off between execution time per floating point operation and can still obtain bounds approaching the tightness of the approach described in [6]. We discuss the main method to obtain a control over the execution time in Section 5, and our overall approach which uses this heuristic to calculate bounds on the range of any variable in an algorithm in Section 6.

5. CONTROLLING EXECUTION TIME

The basic concept we employ to obtain control over the execution time needed to bound the result of an operation is to directly control the number of monomials in every polynomial to a user chosen value, N, and hence the worst case execution time to create any intermediate polynomial will be some function of N. Since the worst case execution time to create any intermediate polynomial becomes constant, the overall execution time of our algorithm grows as $O(n_o)$, and the choice of N provides the user the ability to trade potential tightness of bounds with execution time. We note that this is a much finer level of control than Taylor forms.

To create the intermediate polynomial of only N monomials that still bound the correct result, we apply the algorithm described in Figure 2. This algorithm retains the monomials that have the greatest potential contribution to the final bounds, as calculated by computing the bounds of each monomial using interval arithmetic, and then representing the worst case bounds of the remaining monomials, again computed using interval arithmetic, using a new polynomial that consists of a constant C_i a single monomial ζ_i. The constant C_i is chosen to center the monomial ζ_i over the desired range, for this has previously been shown to obtain the best error properties [26]. The rationale for choosing monomials with the greatest potential contribution is that many monomials within a polynomial represent a small contribution towards the final bounds of the function, and hence if the dependency information of these monomials is lost, it has little impact on the final result. Table 3 demonstrates the potential contributions to the final bounds for all the individual monomials when computing bounds for a simple problem. We note that unlike Taylor forms, our approach may retain higher order monomials at the expense of lower order monomials, for example, in Table 3, the second order monomial x_1y_1 has a higher contribution than the first order monomial δ_1. However, because some input variables may have much wider ranges than other input variables, and often wider ranges than variables bounding finite precision errors, this approach is logical as it is most important to retain the dependency information for the variables with the widest bounds whereas the dependency information for small perturbations can be sacrificed in favor of a reduced execution time.

```
($\hat{p}$, $k$) = **Simplify Polynomial** ($p$, $N$, $k$)
1: $\hat{p}$ = $N$ monomials from $p$ with the largest magnitude of potential contribution to final bounds, calculated by IA
2: $C_k + \zeta_k$ =new polynomial bounding potential contribution of other monomials in $p$
3: $\hat{p}$ = $\hat{p} + C_k + \zeta_k$
4: $k$ = $k + 1$
```

Figure 2: Algorithm to control the size of the polynomial.

6. TRADING QUALITY FOR EXECUTION TIME

While the algorithm given in Figure 2 would be sufficient to control the execution time if integrated into either affine arithmetic or Taylor series with interval remainder bounds, a combined approach would still suffer from dependencies within a polynomial. This was addressed by the use of Handelman representations [6], which also had added benefits by retaining correlation between a numerator and denominator polynomial in the case of division. As such, in this section, we describe how we combine this algorithm with the technique to bound rational functions using Handelman representations to create a scalable framework to find tight bounds for the range or relative error for any variable in an algorithm.

Table 3: Potential contribution of each monomial in $(1+x_1)(1+y_1)(1+\delta_1)$.

Compute $a = x \bullet y$, where $x = [0.8; 1.2]$, $y = [0.9; 1.1]$ in 6 bit floating point
let $|x_1| \leq 0.2, |y_1| \leq 0.1, |\delta_i| \leq 2^{-6} \Rightarrow x \in 10(1+x_1), y \in 10(1+y_1)$
$a = 10(1+x_1)10(1+y_1)(1+\delta_1)$
$= (100 + 100x_1 + 100y_1 + 100\delta_1 + 100x_1y_1 + 100x_1\delta_1$
$+100y_1\delta_1 + 100x_1y_1\delta_1)$

Monomial	Potential Contribution	Monomial	Potential Contribution
100	100	$100x_1$	± 20
$100\delta_1$	±0.09765625	$100x_1\delta_1$	±0.01953125
$100y_1$	±10	$100x_1y_1$	± -2
$100y_1\delta_1$	±0.009765625	$100x_1y_1\delta_1$	±0.001953125

6.1 Representing the range of a variable

One of the problems with using the polynomial simplification algorithm described in Figure 2 is that when we replace all of the monomials with small contributions to the final bounds with a new monomial, we lose information on whether those monomials with small contributions were a function of only the input variables, or a function of both input variables and finite precision errors. This is an issue when computing bounds on the relative error. To compute the relative error, if we have a polynomial p representing the range in infinite precision, and a polynomial $\hat{p}$ representing the range in the presence of finite precision errors, the bound on the relative error is found by maximizing the function $|\frac{p-\hat{p}}{p}|$. However, if we were to control the size of the polynomials p and $\hat{p}$ using the algorithm described in Figure 2, we would lose any correlation between the added monomials bounding small contributions in p and $\hat{p}$.

To demonstrate how this can become a problem, we use a simple example shown in Table 2(a). In this example, we attempt find bounds on the relative error of the computation $(x \bullet y) \bullet z$, where $x \in [0.8; 1.2], y \in [0.9; 1.1], z \in [9.9; 10.1]$ using a 6-bit precision, where we limit the maximum number of monomials in a polynomial to be 6. If we were to compute the relative error of this operation, according to Table 2(a), we must compute bounds of the function $|\dfrac{z_1 + 2.048\zeta_2 - 10\delta_1 - 25.3203\zeta_3}{10 + 10x_1 + 10y_1 + z_1 + 10x_1y_1 + 2.048\zeta_2}|$. The problem with this is that there is correlation between the monomials z_1, ζ_2 and ζ_3 which is lost due to the simplification algorithm. This leads to much wider bounds on the relative error.

In order to avoid this problem, we separate a polynomial $\hat{p}$ into the sum of two polynomials $p + p_\epsilon$, where the polynomial p consists of monomials that are only a function of the input variables, and the polynomial p_ϵ store the additional monomials resulting from the introduction of finite precision errors. We note now that even if we apply the polynomial simplification algorithm, the polynomial p will bound the result in infinite precision, and by keeping these polynomials separate, we can now compute bounds on the relative error by finding bounds of the rational function $|\frac{p_\epsilon}{p}|$; This allows us to find much tighter bounds, as shown in Table 2(b).

This technique is an effective method to describe polynomials, however, to take advantage of correlation between numerator and denominator polynomials using the Handelman representations approach, we bound the range of any intermediate variable in the code using a rational function of the form $\frac{n+n_\epsilon}{d+d_\epsilon}$, where n and d are the numerator and denominator polynomials that contribute to the bounds in infinite precision, and n_ϵ and d_ϵ store the additional monomials resulting from the introduction of finite precision errors.

6.2 Bounding the range of variables in finite precision arithmetic for a user algorithm

In order to compute bounds on the range or relative error for any variable within an algorithm, we first compile the target al-

Table 2: Controlling polynomial size

(a) Using the algorithm defined in Figure 2 to control polynomial size with $N = 6$.

Calculate the relative error of the computation $(x \bullet y) \bullet z$, where $x \in [0.8; 1.2]$, $y \in [0.9; 1.1]$, $z \in [9.9; 10.1]$ in floating point with a 6-bit mantissa. let $\lvert x_1\rvert \leq 0.2, \lvert y_1\rvert \leq 0.1, \lvert z_1\rvert \leq 0.1 \quad \Rightarrow \quad x=(1+x_1), y=(1+y_1), z=(10+z_1)$. Also let $\forall_i, \lvert\delta_i\rvert \leq 2^{-6}, \lvert\zeta_i\rvert \leq 2^{-6}$			
Create polynomials to bound the range of every intermediate variable			
Code	Polynomial bounding variable range	Polynomial in canonical form	Simplified polynomial
$a = x \bullet y$	$a = (1+x_1)(1+y_1)$	$1 + x_1 + y_1 + x_1y_1$	$1 + x_1 + y_1 + x_1y_1$
	$\hat{a} = (1+x_1)(1+y_1)(1+\delta_1)$	$1 + x_1 + y_1 + x_1y_1 + \delta_1 + x_1\delta_1 + y_1\delta_1 + x_1y_1\delta_1$	$1 + x_1 + y_1 + x_1y_1 + \delta_1$ $+0.32\zeta_1$
$b = a \bullet z$	$b = (1 + x_1 + y_1 + x_1y_1)(10$ $+z_1)$	$10 + 10x_1 + 10y_1 + 10x_1y_1 + z_1 + x_1z_1 + y_1z_1 + x_1y_1z_1$	$10 + 10x_1 + 10y_1 + z_1$ $+10x_1y_1 + 2.048\zeta_2$
	$\hat{b} = (1 + x_1 + y_1 + x_1y_1 + \delta_1$ $+0.32\zeta_1)(10 + z_1)(1 + \delta_2)$	$10 + 10x_1 + 10y_1 + 10x_1y_1 + 10\delta_1 + 3.2\zeta_1 + z_1 + x_1z_1 + y_1z_1$ $+x_1y_1z_1 + z_1\delta_1 + 0.32z_1\zeta_1 + 10\delta_2 + 10x_1\delta_2 + 10y_1\delta_2$ $+10x_1y_1\delta_2 + 10\delta_1\delta_2 + 3.2\zeta_1\delta_2 + z_1\delta_2 + x_1z_1\delta_2 + y_1z_1\delta_2$ $+x_1y_1z_1\delta_2 + z_1\delta_1\delta_2 + 0.32z_1\zeta\delta_2$	$10 + 10x_1 + 10y_1 + 10\delta_1$ $+10x_1y_1 + 25.3203\zeta_3$
Find relative error of every intermediate variable			
Variable	Rational function bounding relative error		Bound on relative error using IA
$\lvert\frac{a-\hat{a}}{a}\rvert$	$\lvert\frac{\delta_1 + 0.32\zeta_1}{1 + x_1 + y_1 + x_1y_1}\rvert$		$\lvert 0.0303\rvert$
$\lvert\frac{b-\hat{b}}{b}\rvert$	$\lvert\frac{z_1 + 2.048\zeta_2 - 10\delta_1 - 25.3203\zeta_3}{10 + 10x_1 + 10y_1 + z_1 + 10x_1y_1 + 2.048\zeta_2}\rvert$		$\lvert 0.1026\rvert$

(b) Using the algorithm defined in Figure 2 with $N = 3$ to control separate polynomials for range in infinite precision and the additional monomials resulting from finite precision errors.

Create polynomials to bound the range of every intermediate variable			
Code	Polynomial bounding range of variable	Simplified polynomial, p, bounding range in infinite precision	Simplified polynomial, p_ϵ, bounding finite precision errors
$a = x \bullet y$	$(1+x_1)(1+y_1)(1+\delta_1)$	$1 + x_1 + 7.68\zeta_1$	$\delta_1 + x_1\delta_1 + 0.12\zeta_2$
$b = a \bullet z$	$(1 + x_1 + 7.68\zeta_1 + \delta_1 + x_1\delta_1 + 0.12\zeta_2)(10 + z_1)(1 + \delta_2)$	$10 + 10x_1 + 83.9680\zeta_3$	$1.2\zeta_2 + \delta_2 + 15.6723\zeta_4$
Find relative error of every intermediate variable			
Variable	Rational function bounding relative error		Bound on relative error using IA
$\lvert\frac{a_\epsilon}{a}\rvert$	$\lvert\frac{\delta_1 + x_1\delta_1 + 1.12\zeta_2}{1 + x_1 + 7.68\zeta_1}\rvert$		$\lvert 0.0244\rvert$
$\lvert\frac{b_\epsilon}{b}\rvert$	$\lvert\frac{1.2\zeta_2 + \delta_2 + 15.6723\zeta_4}{10 + 10x_1 + 83.9680\zeta_3}\rvert$		$\lvert 0.0363\rvert$

gorithm into a 2-input static single assignment (SSA) intermediate representation consisting of vector operations, with the aid of techniques such as loop unrolling. For our tests, we performed this by hand, but for more complex examples we could make use of front end compilation tools such as GCC. Throughout our algorithms, we prefer to operate on vectors so as to take advantage of the fact that every element in a vector will typically share the same denominator and hence our algorithms are designed to retain this correlation so as to improve the tightness of bounds. We then proceed to calculate bounds on this intermediate representation using a set of simple algorithms summarized in Figures 3 and 4. In the rest of this section, we explain the rationale behind each of these algorithms.

6.2.1 *Bound variable in code*

In the main algorithm, we first create a set V containing all the input variables in the algorithm, stored as vectors wherever this is applicable. Our algorithm proceeds by sequentially examining each operation in the intermediate representation and creates new rational functions which bound the range of every output element from this operation. The operations we support are scalar multiplication, scalar division, scalar addition and subtraction, vector addition and subtraction, dot products, and any other function to which a polynomial approximation can be computed. The reason we prefer to perform vector operations is because after creating each new rational function, the number of monomials in the polynomials n and d are controlled according to a user choice N_1, and the number of monomials in the polynomials n_ϵ and d_ϵ are controlled according to a user choice N_2, where the choice of N_1 and N_2 is left to a user to trade the execution time against the potential quality of bounds. If we were to perform each operation on scalars instead of performing a single vector operation, then if the number of monomials in the polynomials d and d_ϵ created by the operation are greater than N_1 or N_2, then the denominator polynomials would be simplified with the use of a different variable ζ_k for each element in the vector, meaning that none of the denominators for the vector would be the same. By performing a single vector operation, we only need to simplify the denominator polynomial for the entire vector once, retaining correlation for all the vector elements and improving the overall bounds. We note that because each numerator polynomial for a vector will in general be different, each of these are simplified individually, and in order to capture round-off uncertainty, we first apply the model error described in Section 3 to the polynomial n_ϵ. Finally, once we have created a rational function to bound the range of the desired output variable, we calculate bounds using Handelman representations [6] to try to improve the bounds by taking into account any dependencies in the rational function.

6.2.2 *Compute rational function*

Figure 4 describes how we create a rational function representing the value of every intermediate variable in the SSA version of the code. In general, a rational function representing a result of the basic algebraic operations ($\odot \in \{+, -, \bullet, \div\}$) applied to two other rational functions can easily be computed symbolically. For example, equation (7) shows how to perform multiplication of two rational functions $v_a = \frac{n_a + n_{a\epsilon}}{d_a + d_{a\epsilon}}$ and $v_b = \frac{n_b + n_{b\epsilon}}{d_b + d_{b\epsilon}}$; in this equation, we have used brackets to separate the polynomials which consist of both input variables and finite precision errors. However, for addition or subtraction when the denominator polynomials are different for the two operands, we apply a different approach. The reason for this exception is that this operation can result in a very large numerator polynomial which has lots of correlation with the denominator polynomial, as shown in equation (8). However, when we subsequently simplify the numerator and denominator polynomials, according to the algorithm in Figure 3, we lose correlation between these polynomials, and because the number of monomials

$$\frac{n_a + n_{a\epsilon}}{d_a + d_{a\epsilon}} \times \frac{n_b + n_{b\epsilon}}{d_b + d_{b\epsilon}} = \frac{n_a n_b + (n_a n_{b\epsilon} + n_b n_{a\epsilon} + n_{a\epsilon} n_{b\epsilon})}{d_a d_b + (d_a d_{b\epsilon} + d_b d_{a\epsilon} + d_{a\epsilon} d_{b\epsilon})} \tag{7}$$

$$\frac{n_a + n_{a\epsilon}}{d_a + d_{a\epsilon}} + \frac{n_b + n_{b\epsilon}}{d_b + d_{b\epsilon}} = \frac{n_a d_b + n_b d_a + (n_a d_{b\epsilon} + n_b d_{a\epsilon} + n_{a\epsilon} d_{b\epsilon} + n_{b\epsilon} d_{a\epsilon} + n_{a\epsilon} d_{b\epsilon} + n_{b\epsilon} d_{a\epsilon})}{d_a d_b + (d_a d_{b\epsilon} + d_b d_{a\epsilon} + d_{a\epsilon} d_{b\epsilon})} \tag{8}$$

```
Bound variable in code ($N_1$, $N_2$, code).
// $N_1$, $N_2$ are user chosen variables to control the maximum polynomial sizes
// We denote vectors of rational functions bounding variables with $\boldsymbol{v}$ where
the $i^{th}$ rational function of this vector is indexed $v^i$. The total number of
elements in a vector is given by $|v|$. As a scalar variable is a vector consisting
of only one element so we omit the superscript $i$.
// Number of variables $\delta_j$, $\zeta_k$ are determined at run time.
 1: Create set $V$ of all input variables as vectors of the form:
      $\boldsymbol{v_a} = \left[ \frac{n_a^1 + n_{a\epsilon}^1}{d_a^1 + d_{a\epsilon}^1}, \ldots, \frac{n_a^{|\boldsymbol{v_a}|} + n_{a\epsilon}^{|\boldsymbol{v_a}|}}{d_a^{|\boldsymbol{v_a}|} + d_{a\epsilon}^{|\boldsymbol{v_a}|}} \right]$.
 2: $(j, k) = (1, 1)$.
 3: for every operation $\boldsymbol{v_a} \odot \boldsymbol{v_b}$ in intermediate representation do
 4:   $(\boldsymbol{v_\star}, j)$ = Compute rational function $(\boldsymbol{v_a}, \odot, \boldsymbol{v_b}, N_1, N_2, j)$
 5:   for $i = 1$ to $|\boldsymbol{v_\star}|$ do
 6:     $(n_\star^i, n_{\star\epsilon}^i, k)$ = Simplify Polynomials($n_\star^i$, $n_\star^i \delta_j + n_{\star\epsilon}^i (1 + \delta_j)$
                                   , $N_1$, $N_2$, $k$)
 7:     $j = j + 1$
 8:   end for
 9:   $(d_\star, d_{\star\epsilon}, k)$ = Simplify Polynomials($d_\star$, $d_{\star\epsilon}$, $N_1$, $N_2$, $k$)
10:   $j = j + 1$
11:   Add $\boldsymbol{v_\star}$ to $V$
12: end for
13: Bound desired variable $\boldsymbol{v}$ in $V$ using Handelman representations

$(\hat{p}, \hat{p}_\epsilon, k)$ = Simplify Polynomials $(p, p_\epsilon, N_1, N_2, k)$
 1: $(\hat{p}, k)$ = Simplify Polynomial $(p, N_1, k)$
 2: $(\hat{p}_\epsilon, k)$ = Simplify Polynomial $(p_\epsilon, N_2, k)$
```

Figure 3: Overall algorithm to find bounds on the range or relative error of a variable from a user input code.

in the numerator is substantially larger than in the denominator, this will result in wider bounds.

As a result, in Figure 4, we instead compute a rational function to bound the result by first normalising the denominators of the two input rational functions to 1 by multiplying their numerator polynomials by a polynomial approximation of the reciprocal of their denominator polynomials. To do this, we could use any of the well known techniques in approximation theory such as Taylor approximations, Chebyshev approximations, or the Remez algorithm [23]. Though we note that this can lead to wider bounds due to the errors in the approximation, experimentally these errors have in general been found to be smaller.

To implement other elementary functions, such as ($\odot \in \{\surd$, $\sin()$, $\exp()\}$), we apply polynomial or rational function approximations, as for the reciprocal. In general, the choice of approximation will trade trade quality of bounds with execution time, and due to the wealth of research in this area, this is left to a user to choose the optimum approximation. However, we note that this flexibility is unavailable when using AA and TwIR, for these require the polynomial approximation to be of a specific form. Furthermore, because if we approximate a function over a smaller range, the approximation will generally have less worst case error, we use Handelman representations to find bounds on the range of a variable before performing any polynomial approximation.

7. RESULTS

We have created two examples, shown in Figures 5 and 6 to help demonstrate the benefits of our proposed approach in terms of scalability and quality of bounds. In these figures, we present the original pseudo code alongside a breakdown of this pseudo code into vector operations, because in the original code, no order is specified, but the order of operations affects the accumulation of errors,

```
$(\boldsymbol{v_\star}, j)$ = Compute rational function $(\boldsymbol{v_a}, \odot, \boldsymbol{v_b}, N_1, N_2, j)$
 1: if $(\odot == \bullet)$ then
 2:   $(d_\star, d_{\star\epsilon}) = (d_a d_b, d_a d_{b\epsilon} + d_{a\epsilon} d_b + d_{a\epsilon} d_{b\epsilon})$
 3:   if $(|\boldsymbol{v_b}| == 1)$ then
 4:     for $i = 1$ to $|\boldsymbol{v_a}|$ do
 5:       $(n_\star^i, n_{\star\epsilon}^i) = (\hat{n} + n_a^i n_b, \hat{n}_\epsilon + n_a^i n_{b\epsilon} + n_{a\epsilon}^i n_b + n_{a\epsilon}^i n_{b\epsilon})$
 6:     end for
 7:   else
 8:     $(n_\star, n_{\star\epsilon}) = (0, 0)$
 9:     for $i = 1$ to $|\boldsymbol{v_a}|$ do
10:       $(n_\star, n_{\star\epsilon}) = (n_\star + n_a^i n_b^i, n_{\star\epsilon} + n_a^i n_{b\epsilon}^i + n_{a\epsilon}^i n_b^i + n_{a\epsilon}^i n_{b\epsilon}^i$
                 $+ \delta_j (n_a^i n_b^i + n_a^i n_{b\epsilon}^i + n_{a\epsilon}^i n_b^i + n_{a\epsilon}^i n_{b\epsilon}))$
11:       $n_{\star\epsilon} = \hat{n}\delta_j + \hat{n}_\epsilon (1 + \delta_{j+1})$
12:       $j = j + 2$
13:     end for
14:   end if
15: else if $((\odot == +)$ or $(\odot == -))$ then
16:   if $(d_a, d_{a\epsilon}) == (d_b, d_{b\epsilon})$ then
17:     for $i = 1$ to $|\boldsymbol{v_a}|$ do
18:       if $(|\boldsymbol{v_b}| == 1)$ then
19:         $(\hat{n}^i, \hat{n}_\epsilon^i) = (n_a^i \odot n_b, n_{a\epsilon}^i \odot n_{b\epsilon})$
20:       else
21:         $(\hat{n}^i, \hat{n}_\epsilon^i) = (n_a^i \odot n_b^i, n_{a\epsilon}^i \odot n_{b\epsilon}^i)$
22:       end if
23:     end for
24:     $(d_\star, d_{\star\epsilon}) = (d_a, d_{a\epsilon})$
25:   else
26:     $\boldsymbol{v_1} = \frac{d_a + d_{a\epsilon}}{1}$, $\boldsymbol{v_2} = \frac{d_b + d_{b\epsilon}}{1}$
27:     $\boldsymbol{v_1}$ = Compute Polynomial Approximation($\boldsymbol{v_1}$,$\lambda x.x^{-1}$)
28:     $\boldsymbol{v_2}$ = Compute Polynomial Approximation($\boldsymbol{v_2}$,$\lambda x.x^{-1}$)
29:     $(\boldsymbol{v_1}, j)$ = Compute rational function $(\boldsymbol{v_a}, \bullet, \boldsymbol{v_1}, N_1, N_2, j)$
30:     $(\boldsymbol{v_2}, j)$ = Compute rational function $(\boldsymbol{v_b}, \bullet, \boldsymbol{v_2}, N_1, N_2, j)$
31:     $(\boldsymbol{v_\star}, j)$ = Compute rational function $(\boldsymbol{v_1}, \odot, \boldsymbol{v_2}, N_1, N_2, j)$
32:   end if
33: else if $(\odot == \div)$ then
34:   $\boldsymbol{v_b} = \frac{d_b + d_{b\epsilon}}{n_b + n_{b\epsilon}}$
35:   $(\boldsymbol{v_\star}, j)$ = Compute rational function $(\boldsymbol{v_a}, \bullet, \boldsymbol{v_b}, N_1, N_2, j)$
36: else
37:   $\boldsymbol{v_\star}$ = Compute Polynomial Approximation($\boldsymbol{v_a}$, $\odot$)
38: end if
```

Figure 4: Algorithm to create rational functions bounding intermediate variables.

and hence this information is required to calculate any bound on the range or relative error. Using these examples, we compare against all the main competing methods that are capable of bounding error: IA, TwIR, AA, and Handelman representations. Furthermore, we also examine the impact of using interval splitting on the diagonals of the matrix A and the elements of the vector b when finding bounds using IA, as splitting these intervals will have the greatest impact on the final bounds. We do not perform this for the second example because it is unclear which variables would be best to split, whilst splitting all variables, as we shall see in Section 7.1, would require too large an execution time. These examples are large compared to similar publications in the field, consisting of approximately 20-30 input variables and 400-500 floating point operations, each of which add a new variable bounding the floating point roundoff error; in contrast, the examples of [5, 6] consist of up to 10 input variables and 30 floating point operations, with the former paper not taking finite precision errors into account.

In the first test, because there is little error in the first order approximation of the reciprocation of $a_{i,i}$, and all the non-affine operations are multiplications of polynomials or rational functions by input variables $a_{i,j}$, the majority of the information regarding the

$$A = \begin{pmatrix} 100 & -10 & -15 & -4 & 16 \\ -10 & 105 & -13 & 4 & 14 \\ -14 & -13 & 90 & 19 & -11 \\ 12 & 4 & 14 & 110 & 15 \\ 16 & 14 & -10 & 8 & 95 \end{pmatrix} \pm 1\%, \; b = \begin{pmatrix} 200 \\ -120 \\ -160 \\ 180 \\ -100 \end{pmatrix} \pm 1,$$

```
// The $i^{th}$ element of a vector $v$ is indexed $v^i$ as before
// The $j^{th}$ row of a matrix $A$ is indexed $A(j)$
1: for $k = 1; k \leq 8; k++$ do
2:     for $j = 1; j \leq 5; j++$ do
3:         $x^j = (1-w)x^j + \frac{w}{A(j)^j}(b_j - \sum_{i=1,i\neq j}^{5} A(j)^i x^i)$
4:     end for
5: end for
```

```
1: // Initialisations
2: for $i = 1; i \leq 5; i++$ do
3:   $wDIVa\_i = w/A(i)^i$
4:   for $j = 1; j \leq 5; j++$ do
5:     $ASUBdiagA\_i^j = A(j)^i$
6:   end for
7:   $ASUBdiagA\_i^i = 0$
8: end for
9: $U\_SUBw = 1 - w$
10: // Algorithm
11: for $k = 1; k \leq 8; k++$ do
12:   for $j = 1; j \leq 5; j++$ do
13:     $ax = ASUBdiagA^j \bullet x$
14:     $bSUBax = b^j - ax$
15:     $rhs = wDIVa^j \bullet bSUBax$
16:     $lhs = U\_SUBw \bullet x^j$
17:     $x^j = lhs - rhs$
18:   end for
19: end for
```

Figure 5: 5x5 Successive over relaxation benchmark: inputs, original code and code expressed using vector operations.

final range of the x values is in the first order terms. This implies IA, AA and TwIR should be able to calculate tight bounds, so in this test, we wish to demonstrate that our approach will perform well even where the existing methods ought to perform well. In contrast, the second test involves products of polynomials or rational functions bounding intermediate variables, and the division and square root of a multivariate polynomial, so we use this test to show that our approach can perform well in a more complex algorithm.

When performing polynomial approximations, for a fair comparison we use the same polynomial approximation method for affine arithmetic and our approach, with the exception of using Handelman representations to find the input range for this approximation, as mentioned in Section 6.2.2.

7.1 Test 1: Successive over relaxation

Scalability: Figures 7(a) and 7(b) demonstrate how the execution time on an Intel Xeon E5345 of each of the methods grows with the number of operations when computing the range or relative error for intermediate variables over the course of the successive over relation algorithm.

For the range analysis case, seen in Figure 7(a), IA, 1st order TwIR, AA and our approach initially scale well with the number of operations, whereas TwIR of orders greater than 1 and Handelman representations scale poorly due to exponential growth, as mentioned in Section 4. However, it is also clear that only IA and our approach have an execution time proportional to the number of operations, as expected from our analysis in Section 4, and this means that as the number of operations gets large, our approach can run faster than AA and TwIR. As we have previously mentioned, this is because our approach directly controls the size of the polynomials bounding the range of every intermediate variable, unlike Taylor forms where these polynomials are proportional to the number of operations because a variable bounding the roundoff error is added after every operation. Furthermore, as expected from our analysis in Section 4, AA scales worse than 1st order TwIR because it gains an extra variable for every operation as a result of bounding the error of the higher order terms created by the multiplicative model of error. Finally, this graph shows that when applying IA splitting,

$$A = \begin{pmatrix} 50 & -60 & 70 & -80 \\ -60 & -60 & 40 & 70 \\ 70 & 40 & 40 & -30 \\ -80 & 70 & -30 & -40 \end{pmatrix} \pm 0.25, \; x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \; b = \begin{pmatrix} 80 \\ 60 \\ 40 \\ 70 \end{pmatrix} \pm 0.25$$

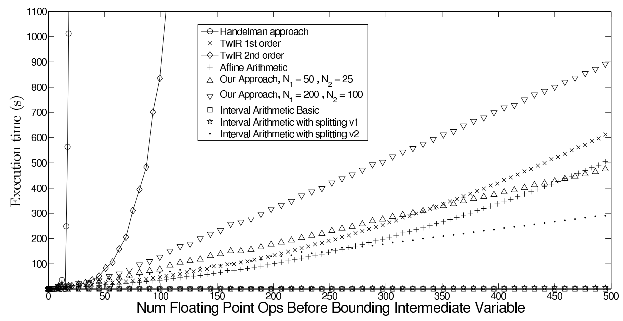
```
1: $v_1 = b - Ax_0;$
2: $\beta_1, \eta = ||v_1||_2$
3: $v_0, w_0, w_{-1} = [0000]'$
4: $\sigma_0, \sigma_1 = 0 \; ; \gamma_0, \gamma_1 = 1$
5: for $i = 1; i \leq 3; i++$ do
6:     $v_i = \frac{v_i}{\beta_i}$
7:     $\alpha = v_i^T A v_i$
8:     $v_{i+1} = Av_i - \alpha v_i - \beta v_{i-1}$
9:     $\beta_{i+1} = ||v_{i+1}||_2$
10:    $\delta = \gamma_i \alpha_i - \gamma_{i-1} \sigma_i \beta_i$
11:    $\rho_1 = \sqrt{\delta^2 + \beta_{i+1}^2}$
12:    $\rho_2 = \sigma_i \alpha_i + \gamma_{i-1} \gamma_i \beta_i$
13:    $\rho_3 = \sigma_{i-1} \beta_i$
14:    $\gamma_{i+1} = \frac{\delta}{\rho_1}$
15:    $\sigma_{i+1} = \frac{\beta}{\rho_1}$
16:    $w_i = \frac{v_i - \rho_3 w_{i-2} - \rho_2 w_{i-1}}{\rho_1}$
17:    $x_i = x_{i-1} + \gamma_{i+1} \eta w_i$
18:    $\eta = -\sigma_{i+1} \eta$
19: end for
```

```
1: // Initialisations
2: for $i = 1; i \leq 4; i++$ do
3:   $Ax^i = A(i) \bullet x_0$
4:   $v_0^i = 0$
5:   $w_0^i = 0$
6:   $w_{-1}^i = 0$
7: end for
8: $v_1 = b - Ax;$
9: $tmp\_\beta = v_1^T \bullet v_1$
10: $\beta_1 = \sqrt{tmp\_\beta}$
11: $\sigma_0, \sigma_1 = 0 \; ; \gamma_0, \gamma_1 = 1$
12: // Algorithm
13: for $i = 1; i \leq 3; i++$ do
14:   $v_i = \frac{v_i}{\beta_i}$
15:   for $j = 1; j \leq 4; j++$ do
16:     $Av^j = A(j) \bullet v_i$
17:   end for
18:   $\alpha = v_i \bullet Av$
19:   $\alpha V = \alpha \bullet v_i$
20:   $\beta V_{-1} = \beta_i \bullet v_{i-1}$
21:   $tmp\_v_{i+1} = \alpha V - \beta V_{-1}$
22:   $v_{i+1} = Av - tmp\_v_{i+1}$
23:   $tmp\_\beta = v_1 \bullet v_1$
24:   $\beta_{i+1} = \sqrt{tmp\_\beta}$
25:   $tmp1\_\delta = \gamma_i \bullet \alpha_i$
26:   $tmp2\_\delta = \gamma_{i-1} \bullet \sigma_i$
27:   $tmp2\_\delta = tmp2\_\delta \bullet \beta_i$
28:   $\delta = tmp1\_\delta - tmp2\_\delta$
29:   $tmp\_\rho_1 = \delta \bullet \delta$
30:   $tmp\_\rho_1 = tmp1\_\rho_1 + tmp\_\beta$
31:   $\rho_1 = \sqrt{tmp\_\rho_1}$
32:   $tmp1\_\rho_2 = \sigma_i \bullet \alpha_i$
33:   $tmp2\_\rho_2 = \gamma_{i-1} \bullet \gamma_i$
34:   $tmp2\_\rho_2 = tmp2\_\rho_2 \bullet \beta_i$
35:   $\rho_2 = tmp1\_\rho_2 + tmp2\_\rho_2$
36:   $\rho_3 = \sigma_{i-1} \bullet \beta_i$
37:   $\gamma_{i+1} = \frac{\delta}{\rho_1}$
38:   $\sigma_{i+1} = \frac{\beta_i}{\rho_1}$
39:   $tmp1\_w = \rho_3 \bullet w_{i-2}$
40:   $tmp2\_w = \rho_2 \bullet w_{i-1}$
41:   $tmp2\_w = tmp1\_w - tmp2\_w$
42:   $tmp1\_w = v_i - tmp2\_w$
43:   $w_i = \frac{tmp1\_w}{\rho_1}$
44:   $tmp\_x = \eta \bullet w_i$
45:   $tmp\_x = \gamma_{i+1} \bullet tmp\_x$
46:   $x_i = x_{i-1} + tmp\_x$
47:   $\eta = -\sigma_{i+1} \bullet \eta$
48: end for
```

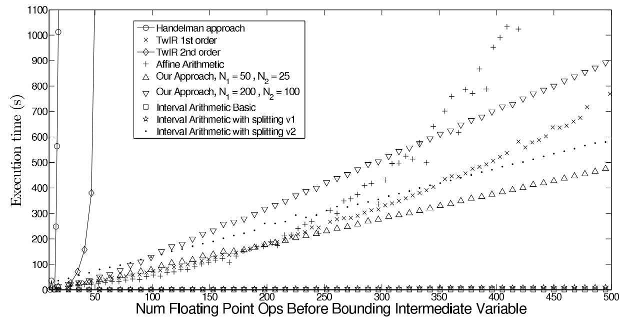
Figure 6: MINRES algorithm benchmark: inputs, original code and code expressed using vector operations.

while the execution time is still proportional to the number of operations, there is a large difference between the execution time depending on the number of splits. We have calculated bounds using IA without splitting, IA where the chosen variables are split into two regions (IA with splitting v1), and IA where the chosen variables are split into three regions (IA with splitting v2); there is a significant difference in execution time between these approaches, indeed this difference can be several orders of magnitude, as seen in Table 4. This is because IA with splitting v1 requires 2^{10} separate interval evaluations, IA with splitting v2 requires 3^{10} interval evaluations. Clearly, performing any further splits is not scalable, and we note that this is with an intelligent selection of 10 variables to split; a naïve approach of splitting every variable would scale far worse and is unlikely to find much tighter bounds.

When bounding the relative error of intermediate variables, as seen in Figure 7(b), the execution time for AA and TwIR grows much faster whereas it remains similar for our approach. The cause of this is that to compute the relative error using AA or TwIR, one must first generate two polynomials, a polynomial p representing the range of the desired variable in the absence of finite precision errors, and a polynomial $\hat{p}$ representing the range in the presence of these errors, then compute bounds of the function $|\frac{p-\hat{p}}{p}|$. To bound this using AA or TwIR, one must first compute a polynomial approximation of $\tilde{p} = 1/p$ then bound the result of the computation $(p - \hat{p}) \times \tilde{p}$, and because $p, \hat{p}$ and $\tilde{p}$ are large polynomials where



(a) Execution time of various methods to bound the range of the intermediate variable after a given number of operations within a 5x5 Successive Over Relaxation.



(b) Execution time of various methods to bound the relative error of the intermediate variable after a given number of operations within a 5x5 Successive Over Relaxation.

Figure 7: Range and relative error of various methods to bound error applied to a 5x5 successive over relaxation.

Table 4: Average bound on range for x vector of various methods to calculate bounds applied to a successive over relaxation of a 5x5 matrix for precision of 20 bits.

Method	Range	Execution time (s)
IA	0.2452	2.279100e-02
IA with Splitting v1	0.2014	10
IA with Splitting v2	0.1866	580
1st Order TwIR	0.2492	612
Affine Arithmetic	0.1369	555
Our Approach $N_1 = 50, N_2 = 25$	0.1366	430
Our Approach $N_1 = 200, N_2 = 50$	0.1330	767

the number of monomials in these polynomials are proportional to the number of operations, the result will be a very large polynomial with many monomials that must be bounded.

Quality of bounds: Table 4 shows the computed bounds on ranges for the first example for the methods that could calculate bounds in a tractable time. It is clear that our approach can compute tighter bounds than the existing approaches, in a reduced execution time, illustrating our claim that our approach can perform well where the existing methods perform well. However, for relative error analysis, shown in Figure 8, only our approach is able to track how relative error decreases with increasing precision. This is because monomials representing floating point error are a function of roundoff variables and input variables, meaning they are second order or greater, and hence first order methods such as AA and 1st order TwIR can only approximate these errors, whereas by retaining 'most significant terms', our approach can retain these higher order terms. In addition, our approach calculates bounds on the worst case relative error that are close to the estimates found by random simulation, implying our bounds are tight.

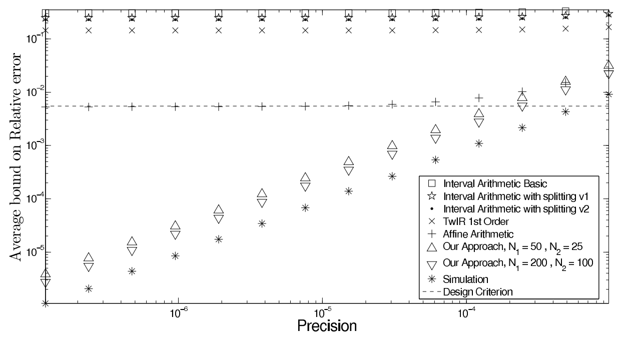


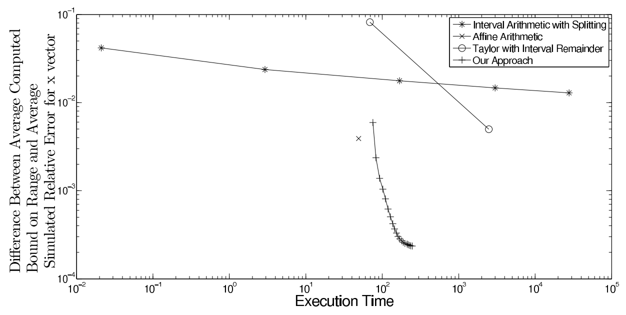
Figure 8: Average bound on relative error for x vector of various methods to calculate bounds applied to a successive over relaxation of a 5x5 matrix.

Table 5: Slice use and max frequency of 5x5 successive over relaxation required according to analytical tools to guarantee the relative error is less than 5.5×10^{-3}, or using IEEE standards.

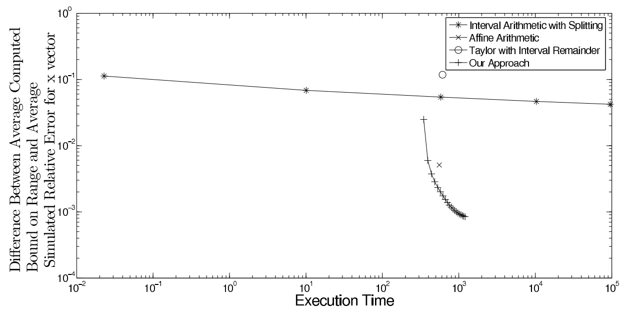
Method	Exponent (# bits)	Mantissa (# bits)	Slice Regs	Slice LUTs	Frequency (MHz)
Simulation	8	11	3562	3012	330
Our Approach	8	13	4261	3647	330
Affine Arithmetic	8	18	6606	5368	300
IA/TwIR	∞	∞	∞	∞	N/A
IEEE Single Precision	8	24	8407	6815	280
IEEE Double Precision	11	53	27200	22066	251

In order to demonstrate how this can be used to improve a hardware design, we created a hardware implementation where every floating point operation in line (3) of Figure 5 has a dedicated floating point operator, and the dot product is performed using a row of parallel multiplication units and an adder reduction tree, as in [27]. Table 5 shows the resources required, post place and route on a Virtex 5 LX 330T, to create such a hardware implementation using the minimum precision necessary to ensure the relative error lies below a threshold of 5.5×10^{-3} according to the various methods. Using our approach, we can create a hardware design with 25% less silicon area than by using AA. The other methods cannot prove bounds, meaning that one could either attempt a simulation based approach, which in these examples would use less hardware at the cost of sacrificing guarantees that it will satisfy the desired bounds, or implement it using IEEE double precision floating point units, but our approach could save up almost 80% of the silicon area in comparison. Our proofs could also be used to show that IEEE single precision would be sufficient to meet the design criterion, for use on other hardware platforms, but this precision would be much greater than is necessary. Furthermore, because our approach can track how relative error decreases with increasing precision we could use it to tune hardware to satisfy a tighter bound on relative error, to which even AA would be unable to find a proof.

Execution time vs quality of bounds performance trade-off: Figure 9 compares the ability of our approach to trade quality of bounds with scalability with the existing approaches. In this figure, we have chosen to compare bounds on the range, because as we have shown earlier, only our approach is capable of tracking how relative error decreases with precision. All the values in this figure are based on a mantissa of 20 bits. Furthermore, in order to compare more approaches, in Figure 9(a), we have restricted the successive over relaxation example to only two iterations, allowing us also to see the quality of bounds and execution time trade-off for increasing the order of TwIR to 2nd order, while in Figure 9(b) we plot the same after the conclusion of the example. For our approach, we have varied N_1 in 10 equal size increments from 20 to



(a) After two iterations.



(b) After seven iterations.

Figure 9: Trade off between bound on range and execution time of various implementations of our approach to find the average bound on the range of x vector of a successive over relaxation of a 5x5 matrix for various approaches.

400 and set $N_2 = \frac{1}{2}N_1$. Finally, to obtain a measure of the quality of the bounds, we plotted the difference between the calculated and simulated bounds. This graph demonstrates how our method provides a much finer grain level of control in comparison to existing methods: interval splitting has a large growth in execution time for every extra split of the desired variables, AA has no control over this trade-off, and TwIR has a large difference in execution time between 1st and 2nd order. In addition, this graph shows once again that our approach is capable of computing the tightest bounds of all these methods, and because these bounds approach those found by simulation, it implies the bounds we compute are tight. While we note that in the case of Figure 9(a), due to the additional overheads of our algorithm in creating rational functions representing the value for intermediate variables, AA can run quicker than our approach, as the number of operations becomes larger as in Figure 9(b), because AA cannot trade execution time for quality of bounds, our approach has the flexibility to run quicker than AA at a cost of bounds, or for longer than AA to obtain tighter bounds.

7.2 Test 2: MINRES

Scalability: Figure 10 demonstrates how the execution time of each of the methods grow with the number of operations when computing the range for intermediate variables over the course of the MINRES algorithm. We note that 2nd order TwIR once again scales poorly with the number of operations, while IA and 1st order TwIR fail to compute bounds after 228 operations because they are unable to prove the input to the $\sqrt{}$ function is non-negative. For our approach and AA, in comparison to Figures 7(a) and 7(b), in this experiment the execution time grows much faster with the number of operations. This is because, as mentioned when introducing these examples, this algorithm contains the multiplication of two polynomials or rational functions which may be large, as

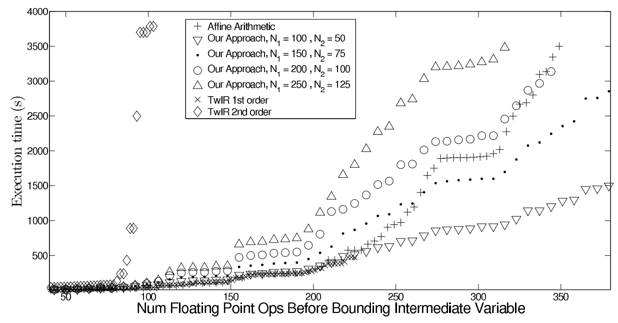


Figure 10: Execution time vs number of operations using various methods to bound range applied to a MINRES algorithm of a 4x4 matrix.

Table 6: Slice use and max frequency of MINRES implementation required according to analytical tools to guarantee the relative error is less than 1×10^{-3}, or using IEEE standards.

Method	Exponent (# bits)	Mantissa (# bits)	Slice Regs	Slice LUTs	Frequency (MHz)
Our Approach, N_1=150, N_2=75	8	21	25221	20793	225
Our Approach, N_1=100, N_2=50	8	22	26216	21519	225
AA/IA/TwIR	∞	∞	∞	∞	N/A
IEEE Single Precision	8	24	30121	24797	150
IEEE Double Precision	11	53	74883	89063	120

opposed to the multiplication of a rational functions by a single input variable. However, we comment that as the maximum number of monomials in the product of two polynomials is given by the product of the number of monomials in each polynomial and the size of the AA polynomial for intermediate variables can be much greater than $N_1 + N_2$. This ensures AA suffers much more than our approach, and is the reason our approach becomes faster than AA after much fewer operations than in Figures 7(a) and 7(b). We also comment that this graph grows in steps, unlike Figures 7(a) and 7(b), this is because operations which are products of two polynomials create many more monomials and take longer than operations which are sums of two polynomials, and hence the execution time for sums is below the worst case execution time for any operation.

Quality of bounds: When attempting to calculate bounds on the relative error that would enable one to create an optimized hardware design, once again, only our approach was capable of computing bounds for this example that track how relative error decreases with increasing precision. This was similarly because errors arising from the use of finite precision arithmetic will be second order or greater, meaning first order methods can only approximate these errors. Table 6 shows the resource use for a parallel implementation of the MINRES algorithm, as described in [28], that would be required to satisfy a bound on relative error of less than 1×10^{-3} using the various methods to calculate bounds, including our approach with different run-times, as well as using IEEE single and double precision. This table again demonstrates significant savings can be made in comparison with IEEE double precision. Furthermore, by using an increasing run-time, we obtain tighter bounds that result in a smaller hardware implementation, highlighting the trade-off we aim to achieve. Finally, we note that in this instance our software has the potential to obtain a proof that IEEE single precision is sufficient to satisfy the desired precision, and this illustrates how our tool could also be used to obtain performance improvements on other hardware platforms.

Altogether, we have shown that our approach can compute bounds that are much tighter than competing approaches in a smaller execution time, and has the ability to scale to much larger examples because its execution time scales worst case linearly with the number of operations. Furthermore, by retaining higher order information, our technique can also track the effect of finite precision errors and compute tight bounds on the relative error introduced by the use of finite precision arithmetic, and this enables us to design hardware that meets a given relative error specification with less silicon area; in the case of the successive over relaxation example, saving over 80% of the slices in comparison to IEEE 754 double precision, and in the case of the MINRES example, saving over 65% of the slices in comparison to IEEE 754 double precision.

8. CONCLUSION

This paper has presented a new algorithm to calculate bounds on finite precision errors. We have demonstrated that this algorithm is more scalable than the existing approaches, meaning that it has the potential to be applied to examples that even the most advanced methods previously could not realistically handle, and can also calculate tighter bounds than the existing methods. In addition, we have shown that our algorithm has significantly greater control over the trade-off between execution time and quality of bounds making it useful for both small and large examples within an optimization framework. Finally, we have shown that because our tool can not only find tight bounds on the range, but also track how relative error decreases with increasing precision, we can use it to create hardware designs with guaranteed error properties that obtain significant silicon area savings over hardware implementations that adhere to IEEE standard single or double precision.

9. REFERENCES

[1] G. Chow, K. Kwok, W. Luk, and P. Leong, “Mixed precision processing in reconfigurable systems,” in *Proc. IEEE. Symp. on Field-Programmable Custom Computing Machines*, 2011, pp. 17–24.

[2] A. R. Lopes and G. A. Constantinides, “A fused hybrid floating-point and fixed-point dot-product for FPGAs.” in *Proc. Int. Symp. on Applied Reconfigurable Recomputing*, 2010, pp. 157–168.

[3] M. deLorimier and A. DeHon, “Floating-point sparse matrix-vector multiply for FPGAs,” in *Proc. Int. Symp. on Field-Programmable Gate Arrays*, 2005, pp. 75–85.

[4] L. Zhuo, G. R. Morris, and V. K. Prasanna, “High-performance reduction circuits using deeply pipelined operators on FPGAs,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377–1392, 2007.

[5] A. Kinsman and N. Nicolici, “Bit-width allocation for hardware accelerators for scientific computing using SAT-modulo theory,” *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 29, pp. 405–413, 2010.

[6] D. Boland and G. Constantinides, “Bounding variable values and round-off effects using handelman representations,” *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 30, no. 11, pp. 1691 –1704, 2011.

[7] Y. Pang, K. Radecka, and Z. Zilic, “Optimization of imprecise circuits represented by Taylor series and real-valued polynomials,” *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 29, pp. 1177–1190, August 2010.

[8] G. Constantinides, P. Cheung, and W. Luk, “Optimum wordlength allocation,” *Proc. Int. Symp. Field-Programmable Custom Computing Machines*, pp. 219–228, 2002.

[9] D.-U. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk, and G. Constantinides, “Accuracy-guaranteed bit-width optimization,” *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 25, no. 10, pp. 1990–2000, 2006.

[10] M. L. Chang and S. Hauck, “Automated least-significant bit datapath optimization for FPGAs,” *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, pp. 59–67, 2004.

[11] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia: SIAM, 1994.

[12] G. Constantinides, A. Kinsman, and N. Nicolici, “Numerical data representations for FPGA-based scientific computing,” *Design & Test of Computers*, vol. 28, no. 4, pp. 8–17, 2011.

[13] Z. Zhao and M. Leeser, “Precision modeling and bit-width optimization of floating-point applications,” in *High Performance Embedded Computing*, 2003, pp. 141–142.

[14] R. E. Moore, *Interval Analysis*. Englewood Cliff, NJ: Prentice-Hall, 1966.

[15] B. Einarsson, *Handbook on Accuracy and Reliability in Scientific Computation*. Soc for Industrial & Applied Math, 2005, ch. 10, pp. 195 – 240.

[16] F. de Dinechin, C. Q. Lauter, and G. Melquiond, “Assisted verification of elementary functions using gappa,” in *Proc. Symp. Applied computing*, 2006, pp. 1318–1322.

[17] A. Kinsman and N. Nicolici, “Computational bit-width allocation for operations in vector calculus,” in *IEEE Int. Conf. on Computer Design*, oct. 2009, pp. 433 –438.

[18] ——, “Robust design methods for hardware accelerators for iterative algorithms in scientific computing,” in *Proc. Design Automation Conference*, 2010, pp. 254–257.

[19] A. Neumaier, “Taylor forms - use and limits,” *Reliable Computing*, vol. 9, pp. 43–79, 2003.

[20] N. Courtois, A. Klimov, J. Patarin, and A. Shamir, “Efficient algorithms for solving overdefined systems of multivariate polynomial equations,” in *Proc. Int. Conf. on Theory and application of cryptographic techniques*, 2000, pp. 392–407.

[21] L. H. de Figueiredo and J. Stolfi, *Self-Validated Numerical Methods and Applications*. Rio de Janeiro: IMPA/CNPq, 1997.

[22] K. Makino and M. Berz, “Taylor models and other validated functional inclusion methods,” *International Journal of Pure and Applied Mathematics*, vol. 4, pp. 379–456, 2003.

[23] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*. Birkhauser, 2005.

[24] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA, USA: Soc for Industrial & Applied Math, 2002.

[25] W. S. Brown, “A simple but realistic model of floating-point computation,” *ACM Trans. Math. Softw.*, vol. 7, pp. 445–480, December 1981.

[26] H. Ratschek, “Centered forms,” *SIAM Journal on Numerical Analysis*, vol. 17, no. 5, pp. pp. 656–662, 1980.

[27] D. Boland and G. Constantinides, “Optimising memory bandwidth use and performance for matrix-vector multiplication in iterative methods,” *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, pp. 22:1–22:14, 2011.

[28] ——, “An FPGA-based implementation of the MINRES algorithm,” in *Proc. Int. Conf. Field Programmable Logic and Applications*, Sept. 2008, pp. 379–384.