A SCALABLE PRECISION ANALYSIS FRAMEWORK

David Boland and George Constantinides, *Senior Member, IEEE*

***Abstract*—In embedded computing, typically some form of silicon area or power budget restricts the potential performance achievable. For algorithms with limited dynamic range, custom hardware accelerators manage to extract significant additional performance for such a budget via mapping operations in the algorithm to fixed-point. However, for complex applications requiring floating-point computation, the potential performance improvement over software is reduced. Nonetheless, custom hardware can still customise the precision of floating-point operators, unlike software which is restricted to IEEE standard single or double precision, to increase the overall performance at the cost of increasing the error observed in the final computational result. Unfortunately, because it is difficult to determine if this error increase is tolerable, this task is rarely performed.**

We present a new analytical technique to calculate bounds on the range or relative error of output variables, enabling custom hardware accelerators to be tolerant of floating point errors by design. In contrast to existing tools that perform this task, our approach scales to larger examples and obtains tighter bounds, within a smaller execution time. Furthermore, it allows a user to trade the quality of bounds with execution time of the procedure, making it suitable for both small and large-scale algorithms.

I. Introduction

The use of custom hardware to accelerate an application has been extensively studied in the reconfigurable computing community, and there is a wide range of literature illustrating that significant performance gains over a software implementation can be obtained [1], [2]. Due to the increasing functionality of modern reconfigurable hardware, these custom hardware accelerators are increasingly being designed using IEEE-754 standard single or double precision so as to report performance gains as a speed up over software whilst operating on the same data types [3], [4]. However, given that floating point addition is not associative [5], and that it is likely that parallelism will change the order of operations, it is impractical to insist that both implementations obtain the exact same result. Ideally, it would be desirable to create a proof that both designs will satisfy an alternative metric, such as 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), to ensure the hardware accelerator will be appropriate at a system level.

If a tool existed that could generate such proofs, it would allow much greater freedom when creating a hardware accelerator to maximise the performance subject to some fixed silicon area or power budget. This is because it would enable one to tune the precision used throughout an algorithm, which has an impact on both of these factors, as well as the memory use, latency and maximum frequency of the design. Such a tool could also help obtain performance improvements for general purpose processors and GPUs by creating a proof that ensures moving from double to single precision is valid, for such a move results in significantly improved performance [6], [7]. However, while these devices are largely restricted to these two IEEE standards, because custom hardware has the freedom to implement any precision, there is much greater potential to exploit in this latter case. As such, in this paper, we create a tool that can be used within word-length optimization frameworks to maximize the performance of an accelerator, whilst providing guaranteed numerical properties, such that finite precision errors are tolerated by design. This paper elaborates on previous work by the authors [8] with a more detailed description of the background literature, including a study of word-length optimization tools to show how they could potentially benefit from the tool we describe. It also contains an extended results section providing more detailed discussion of our approach in comparison to existing approaches and new benchmarks illustrating its use for word-length optimization in the field of DSP.

There are three important metrics for any such tool: whether it calculates a bound, the quality of the bound and the scalability of the tool. Several individual applications have explored the potential performance improvements obtainable by optimizing the precision throughout a design [9], [10], but for large-scale datapaths these studies are largely based upon simulation. Since simulation can only estimate the error over a set of input ranges for a given precision, it is preferable for a tool to calculate a bound that guarantees the accelerator will satisfy the design criterion. 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. Scalability is important because rounding errors arise from every operation. This means the tool must keep track of all the potential errors that can arise throughout an algorithm and calculate bounds, taking all these errors into account, in a reasonable time to be of practical use. Furthermore, a scalable and fast tool is important because 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 that calculates a valid bound has typically focused on either tightness of bounds or scalability of the tool. As such, there are methods that can calculate tight bounds, but are restricted to trivial computational kernels [11]–

[13], or the restricted LTI domain in DSP [14], and methods that scale to larger problems, but do not find tight bounds and consequently limit the potential hardware optimizations [15], [16]. In this work, we describe a tool to calculate bounds for any algorithm that consists 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 bounds that are known at run-time by loop unrolling, and conditionals statements by analyzing branches separately. The bounds our approach finds approach the quality of the highly 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 much greater level of control over the trade-off of execution time against quality of bounds.

We examine the use of our method on two iterative algorithms to solve a system of linear equations: successive-over-relaxation [17], and MINRES [18], and also on an FFT [19] and a normalised least mean square adaptive filter. Finding the solution to a system of linear equations is a common subcomputation in scientific computing [20], control [21] and communication systems, for example within techniques such as channel equalization, spectral estimation and MIMO communications [22]. The FFT is a widely used technique in DSP, for example in OFDM demodulators, while the normalised least mean square adaptive filter example allows us to explore the benefits of word-length optimization on a hardware accelerator for a specific application, in this case, echo cancellation. We demonstrate that because our approach obtains better bounds than existing analytical techniques, in a much shorter execution time, we can use it to tune the precision used in a custom hardware accelerator for these algorithms, and this translates into a significant reduction in silicon area.

This paper first describes background into the main existing approaches to calculate bounds in Section II and how to model floating point round-off error in Section III. We follow this with an analysis of how this model of error can affect the scalability of the methods to calculate bounds in Section IV, before we describe the principles behind our new method that enable us to gain control over the execution time in Section V. We subsequently discuss our algorithm to compute tight bounds in Section VI, and demonstrate its advantages over existing methods in Section VII before drawing conclusions in Section VIII.

II. Background

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 [23]. 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. In addition, we describe their use within various word-length optimization tools to illustrate the depth of research aiming to tap into this potential performance, and also to show how the performance of a word-length optimization framework is dependent upon the ability of the core analysis technique.

The most straightforward way to estimate an error is through simulation. The aim of a simulation-based approach is to find the inputs that will cause the extreme ranges of the data set. Tools using simulation based approaches include the work by Sung *et al.* [24]–[26], Cantin *et al.* [27], [28], and Roy et al. [29], which trade system area with precision. However, even though the performance benefits are substantial, in the word-length optimization tools a large amount of effort is expended on minimizing the simulation time to obtain a good estimate of the bounds. This not only illustrates that simulation is time consuming, but more importantly that the hardware created cannot be guaranteed to satisfy the desired design constraints because the size of the search space for the inputs will generally be too large to explore exhaustively. This means simulation can only estimate the error because corner cases can be missed. This is true even for advanced simulation methods, such as statistical profiling [30].

To calculate true bounds for general algorithms, the most well known analytical approach is interval arithmetic (IA) [31]. 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.

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

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 [31]. Despite this problem, because it is easy to implement and can find bounds within a very short execution time, some tools have been developed that use interval arithmetic. The Précis project [16], [32] is one such word-length optimization tool which combines interval analysis with area models of simple components such as adders and multipliers within a simulated annealing based approach for automatic optimization. Other approaches include the FRIDGE project [33], [34], Cmar *et al.* [35] and Gaffar *et al.* [36], which combine interval analysis with simulation to identify cases in a datapath that have wide bounds due to dependencies so as to perform word-length optimization with more sophisticated examples; for example the latter work involves floating point computations. However, any such combined approach once again loses any guaranteed numerical properties because it is not possible to exhaustively

simulate over a set of inputs.

To enable interval analysis to become more widespread, there is an active community of researchers in *robust computing* who have developed ways to mitigate the dependency problem [37]. 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). However, while effective, the number of intervals that must be evaluated grows as $\Theta(n_o n_s^{n_{sv}})$, 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 that such an approach is not computationally scalable, and 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' [38], or the work by Nicolici *et al.* which uses Satisfiability-Modulo Theories (SMT) [11], the quality of bounds these approaches can obtain are heavily limited by the run-time.

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

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

Alternatively, more recently a set of methods that can be loosely grouped together under the name of Taylor forms, analyzed in detail [39], have been developed which use a polynomial representation of the error terms. The intuition behind this method being 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 [40].

The most well-known of these Taylor forms, affine arithmetic (AA) [41], 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, and 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 VII.

To minimize both these effects, the more general method by Berz et al., named Taylor methods with Interval Remainder bounds (TwIR) [42] 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.

The Minibit and later Minibit+ [15], [43], [44] tools are examples of the use of affine arithmetic, combined with interval arithmetic, within a word-length optimization framework, again using simulated annealing based refinement alongside area cost models. While this tool demonstrates that tight bounds can be obtained, in this work, the examples are very small in comparison to our benchmarks we describe in Section VII, for example matrix multiplication of 2x2 matrices and an 8x8 discrete cosine transform, and it is unclear whether this approach will perform well for larger examples. Affine arithmetic has also been used by Caffarena *et al.* [45] to model errors in the DSP domain within a word-length optimization framework. However, this work focuses on DSP systems with the aim of obtaining good average case SQNR using perturbation theory to the effect that higher order terms created by operations such as multiplication of two polynomials are neglected to maintain an affine model. This work acknowledges that such an assumption can result in errors, and that this error is amplified by feedback, using the example of an 2nd order LMS filter. This will mean the results cannot be used in systems where it is critical there are no errors, and once again, the examples are very simple in comparison to those we describe in Section VII.

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 [12]. This work demonstrated that using this approach it is possible to obtain 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 IV. 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 [46].

Altogether, we have discussed how the existing analytical techniques either suffer from a lack of tightness of bounds or a lack of scalability when targeted to large examples, and this limitation is reflected in any word-length optimization tools that uses these techniques. We note that the work by Constantinides *et al.* [14], [47] and Menard *et al.* [48] have avoided these problems by restricting the problem domain to linear time invariant systems, these are systems that consist only of additions, subtractions and constant coefficient multi-

plications. In this problem domain, it is possible to analytically calculate the optimum number of bits using LTI theory, and then optimise the individual word-lengths to minimise the area and satisfy a design criterion, such as signal to noise ratio, using heuristics or integer linear programming. However, any algorithm that involves general multiplication is not linear time invariant, so we aim to create a tool that can calculate such bounds for a more general algorithm, as we describe in Section VI-B.

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, which may be of use within a word-length optimization framework, and show how our new method can be used to create superior hardware designs.

III. 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), where η represents the number of mantissa bits used and δ represents the small unknown roundoff error [5]. 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, allowing us to create a polynomial to represent the potential range of a variable, as shown for a simple example in Table I. 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. Some additional possibilities could be to detect that one operand is a power of two and set the error to zero in this case, but these will have minimal impact and severely complicate modelling [49]. We note, however, that provided these models, or other models for fixed-point error, can be expressed using polynomials or rational functions, the algorithms described in this work are still applicable.

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

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

Table I
Construction of polynomials

x, y are inputs Δ is the error bound determined by the precision, so that $\lvert\delta_i\rvert \leq \Delta$	
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 II 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 V.

IV. 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 section and the remainder of 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 is defined 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 II, in the case of applying interval splitting, it will then scale as $O(n_o n_s^{n_{sv}})$.

At the other extreme lies the approach using Handelman representations. In this approach, the polynomials representing the variable value, as in Table I, are first expressed in a canonical form, 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 ≤ n_o; i++ do
    y = y • x[i]
end for
```

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

$$x_1x_2....x_{n_o}(1+\delta_1)(1+\delta_2)...(1+\delta_{n_o}) \tag{6}$$

As we indicated in Section II, Taylor forms control this growth by restricting the canonical form of every intermediate variable 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 described in Section III, because a variable (δ_i) bounding the finite precision error will be added after every operation, and this means that when including the n input variables, the total number of variables for an intermediate polynomial bounding the range of a variable in an algorithm 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 bounding the range of an intermediate variable can grow as $O\binom{n_o+n+\rho}{\rho}$, and hence 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 for existing work, only interval arithmetic has an execution time that scales well

with the number of floating point operations. However, as shown in [12], [41], [42], 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 particularly poorly in the number of variables. Our aim is to create an approach where the execution time grows in proportion with the code size, of $O(n_o)$, provides a user a very flexible level of control over the execution time per floating point operation and can still obtain bounds approaching the tightness of the approach described in [12]. We discuss the main method to obtain a control over the execution time in Section V, and our overall approach which uses this heuristic to calculate bounds on the range of variables in an algorithm in Section VI.

V. 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. This bounds the worst case execution time to create any intermediate polynomial to be some function of N. As a result, because 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 level of control is a much finer level of control than that provided by Taylor forms.

To create the intermediate polynomial of only N monomials that still bounds 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. It then represents 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 rationale for choosing monomials that have 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 II 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 II, 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. In contrast, the dependency information for small perturbations can be sacrificed in favor of a reduced execution time. In the final stage, we represent the range of all unwanted

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

Compute $a = x \bullet y$, where $x = [8; 12], y = [9; 11]$ 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	$100x_1$	$\pm$ 20
$100\delta_1$	$\pm$0.09765625	$100x_1\delta_1$	$\pm$0.01953125
$100y_1$	$\pm$10	$100x_1y_1$	$\pm$ -2
$100y_1\delta_1$	$\pm$0.009765625	$100x_1y_1\delta_1$	$\pm$0.001953125

monomials using a monomial ζ_i along with a constant C_i that centers the monomial ζ_i over the desired range, for this has previously been shown to obtain the best error properties [50].

$(\hat{p}, k)$ = **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.

VI. 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 technique using Handelman representations [12], 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 variables in an algorithm.

A. 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, then the bound on the relative error is computed by maximizing the function $|\frac{p-\hat{p}}{p}|$. However, if we 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 III(a). In this example, we

Table III
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 $\|x_1\| \leq 0.2, \|y_1\| \leq 0.1, \|z_1\| \leq 0.1 \quad \Rightarrow \quad x = (1 + x_1), y = (1 + y_1), z = (10 + z_1)$. Also let $\forall_i, \|\delta_i\| \leq 2^{-6}, \|\zeta_i\| \leq 2^{-6}$			
Create polynomials to bound the range of every intermediate variable			
Code	Polynomial bounding variable range	Polynomial in canonical form	Simplified polynomial
$a = x \bullet y$	$a = (1 + x_1)(1 + y_1)$	$1 + x_1 + y_1 + x_1 y_1$	$1 + x_1 + y_1 + x_1 y_1$
	$\hat{a} = (1 + x_1)(1 + y_1)(1 + \delta_1)$	$1 + x_1 + y_1 + x_1 y_1 + \delta_1 + x_1\delta_1 + y_1\delta_1 + x_1 y_1\delta_1$	$1 + x_1 + y_1 + x_1 y_1 + \delta_1 + 0.32\zeta_1$
$b = a \bullet z$	$b = (1 + x_1 + y_1 + x_1 y_1)(10 + z_1)$	$10 + 10x_1 + 10y_1 + 10x_1 y_1 + z_1 + x_1 z_1 + y_1 z_1 + x_1 y_1 z_1$	$10 + 10x_1 + 10y_1 + z_1 + 10x_1 y_1 + 2.048\zeta_2$
	$\hat{b} = (1 + x_1 + y_1 + x_1 y_1 + \delta_1 + 0.32\zeta_1)(10 + z_1)(1 + \delta_2)$	$10 + 10x_1 + 10y_1 + 10x_1 y_1 + 10\delta_1 + 3.2\zeta_1 + z_1 + x_1 z_1 + y_1 z_1 + x_1 y_1 z_1 + z_1\delta_1 + 0.32 z_1\zeta_1 + 10\delta_2 + 10x_1\delta_2 + 10y_1\delta_2 + 10x_1 y_1\delta_2 + 10\delta_1\delta_2 + 3.2\zeta_1\delta_2 + z_1\delta_2 + x_1 z_1\delta_2 + y_1 z_1\delta_2 + x_1 y_1 z_1\delta_2 + z_1\delta_1\delta_2 + 0.32 z_1\zeta\delta_2$	$10 + 10x_1 + 10y_1 + 10\delta_1 + 10x_1 y_1 + 25.3203\zeta_3$
Find relative error of every intermediate variable			
Variable	Rational function bounding relative error		Bound on relative error using IA
$\|\frac{a - \hat{a}}{a}\|$	$\|\frac{\delta_1 + 0.32\zeta_1}{1 + x_1 + y_1 + x_1 y_1}\|$		0.0303
$\|\frac{b - \hat{b}}{b}\|$	$\|\frac{z_1 + 2.048\zeta_2 - 10\delta_1 - 25.3203\zeta_3}{10 + 10x_1 + 10y_1 + z_1 + 10x_1 y_1 + 2.048\zeta_2}\|$		0.1026

(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
$\|\frac{a_\epsilon}{a}\|$	$\|\frac{\delta_1 + x_1\delta_1 + 1.12\zeta_2}{1 + x_1 + 7.68\zeta_1}\|$		0.0244
$\|\frac{b_\epsilon}{b}\|$	$\|\frac{1.2\zeta_2 + \delta_2 + 15.6723\zeta_4}{10 + 10x_1 + 83.9680\zeta_3}\|$		0.0363

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 III(a), we must compute bounds on the function $|\frac{z_1 + 2.048\zeta_2 - 10\delta_1 - 25.3203\zeta_3}{10 + 10x_1 + 10y_1 + z_1 + 10x_1 y_1 + 2.048\zeta_2}|$. 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 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 III(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 intermediate variables 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.

B. 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 algorithm into a 2-input static single assignment (SSA) intermediate representation consisting of vector operations, with the aid of techniques such as loop unrolling. We note that how the loop is unrolled can affect the error seen at the output, as described by Roy et al. [29]; we have unrolled our loops in a similar fashion. Throughout our algorithms, we prefer to operate on vectors in order to take advantage of the fact that every element in a vector will often 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.

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

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 δ_j, ζ_k are determined at run time.
Create set V of all input variables as vectors of the form:
$\boldsymbol{v_a} = \left[\frac{n_a^1+n_{a\epsilon}^1}{d_a^1+d_{a\epsilon}^1}, ..., \frac{n_a^{|\boldsymbol{v_a}|}+n_{a\epsilon}^{|\boldsymbol{v_a}|}}{d_a^{|\boldsymbol{v_a}|}+d_{a\epsilon}^{|\boldsymbol{v_a}|}}\right]$.
$(j,k) = (1,1)$.
for every operation $\boldsymbol{v_a} \odot \boldsymbol{v_b}$ in intermediate representation **do**
  $(\boldsymbol{v_\star}, j)$ = Compute rational function $(\boldsymbol{v_a}, \odot, \boldsymbol{v_b}, N_1, N_2, j)$
  for $i = 1$ to $|\boldsymbol{v_\star}|$ **do**
    $(n_\star^i, n_{\star\epsilon}^i, k)$ = Simplify Polynomials$(n_\star^i, n_\star^i\delta_j + n_{\star\epsilon}^i(1+\delta_j), N_1, N_2, k)$
    $j = j + 1$
  end for
  $(d_\star, d_{\star\epsilon}, k)$ = Simplify Polynomials$(d_\star, d_{\star\epsilon}, N_1, N_2, k)$
  $j = j + 1$
  Add $\boldsymbol{v_\star}$ to V
end for
Bound desired variable $\boldsymbol{v}$ in V using Handelman representations

$(\hat{p}, \hat{p}_\epsilon, k)$ = **Simplify Polynomials** $(p, p_\epsilon, N_1, N_2, k)$
$(\hat{p}, k)$ = Simplify Polynomial (p, N_1, k)
$(\hat{p}_\epsilon, k)$ = Simplify Polynomial (p_ϵ, N_2, k)

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

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, the denominator polynomials would be simplified with the use of a different variable ζ_k for each element in the vector. As a result, none of the denominators for the vector would be the same. Instead, 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 III 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 [12] to try to improve the bounds by taking into account any dependencies in the rational function.

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

$(\boldsymbol{v_\star}, j)$ = **Compute rational function** $(\boldsymbol{v_a}, \odot, \boldsymbol{v_b}, N_1, N_2, j)$
if $(\odot == \bullet)$ **then**
  $(d_\star, d_{\star\epsilon}) = (d_a d_b, d_a d_{b\epsilon} + d_{a\epsilon} d_b + d_{a\epsilon} d_{b\epsilon})$
  if $(|\boldsymbol{v_b}| == 1)$ **then**
    for $i = 1$ to $|\boldsymbol{v_a}|$ **do**
      $(n_\star^i, n_{\star\epsilon}^i) = (\hat{n} + n_a^i n_b, \hat{n}_\epsilon + n_a^i n_{b\epsilon} + n_{a\epsilon}^i n_b + n_{a\epsilon}^i n_{b\epsilon})$
    end for
  else
    $(n_\star, n_{\star\epsilon}) = (0, 0)$
    for $i = 1$ to $|\boldsymbol{v_a}|$ **do**
      $(n_\star, n_{\star\epsilon}) = (n_\star + n_a^i n_b^i, n_{\star\epsilon} + n_a^i n_{b\epsilon}^i + n_{a\epsilon}^i n_b^i + n_{a\epsilon}^i n_{b\epsilon}^i + \delta_j(n_a^i n_b^i + n_a^i n_{b\epsilon}^i + n_{a\epsilon}^i n_b^i + n_{a\epsilon}^i n_{b\epsilon}))$
      $n_{\star\epsilon} = \hat{n}\delta_j + \hat{n}_\epsilon(1+\delta_{j+1})$
      $j = j + 2$
    end for
  end if
else if $((\odot == +)$ **or** $(\odot == -))$ **then**
  if $(d_a, d_{a\epsilon}) == (d_b, d_{b\epsilon})$ **then**
    for $i = 1$ to $|\boldsymbol{v_a}|$ **do**
      if $(|\boldsymbol{v_b}| == 1)$ **then**
        $(\hat{n}^i, \hat{n}_\epsilon^i) = (n_a^i \odot n_b, n_{a\epsilon}^i \odot n_{b\epsilon})$
      else
        $(\hat{n}^i, \hat{n}_\epsilon^i) = (n_a^i \odot n_b^i, n_{a\epsilon}^i \odot n_{b\epsilon}^i)$
      end if
    end for
    $(d_\star, d_{\star\epsilon}) = (d_a, d_{a\epsilon})$
  else
    $\boldsymbol{v_1} = \frac{d_a + d_{a\epsilon}}{1}$, $\boldsymbol{v_2} = \frac{d_b + d_{b\epsilon}}{1}$
    $\boldsymbol{v_1}$ = Compute Polynomial Approximation$(\boldsymbol{v_1}, \lambda x.x^{-1})$
    $\boldsymbol{v_2}$ = Compute Polynomial Approximation$(\boldsymbol{v_2}, \lambda x.x^{-1})$
    $(\boldsymbol{v_1}, j)$ = Compute rational function $(\boldsymbol{v_a}, \bullet, \boldsymbol{v_1}, N_1, N_2, j)$
    $(\boldsymbol{v_2}, j)$ = Compute rational function $(\boldsymbol{v_b}, \bullet, \boldsymbol{v_2}, N_1, N_2, j)$
    $(\boldsymbol{v_\star}, j)$ = Compute rational function $(\boldsymbol{v_1}, \odot, \boldsymbol{v_2}, N_1, N_2, j)$
  end if
else if $(\odot == \div)$ **then**
  $\boldsymbol{v_b} = \frac{d_b + d_{b\epsilon}}{n_b + n_{b\epsilon}}$
  $(\boldsymbol{v_\star}, j)$ = Compute rational function $(\boldsymbol{v_a}, \bullet, \boldsymbol{v_b}, N_1, N_2, j)$
else
  $\boldsymbol{v_\star}$ = Compute Polynomial Approximation$(\boldsymbol{v_a}, \odot)$
end if

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

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 that 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 containing 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 because the number of monomials in the numerator is substantially larger than in the denominator.

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 [46]. Though we note that this can lead to our approach finding wider bounds than the true bounds due to the errors in the approximation, experimentally

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

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

these errors have in general been found to be smaller than applying our algorithm without this additional approximation.

To implement other elementary functions, such as ($\odot \in \{\sqrt{}, \sin(), \exp()\}$), 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.

VII. Results

A. Benchmarks

We have created four benchmarks, shown in Figures 5, 6, 7 and 8 and to help demonstrate the benefits of our proposed approach in terms of scalability and quality of bounds. These figures present the original pseudo code alongside a breakdown of this pseudo code into vector operations. We provide this breakdown because in the original code, no order is specified, but the order of operations affects the accumulation of errors, and hence this information is required to calculate any bound on the range or relative error. Using these benchmarks, we compare against all the main competing methods that are capable of bounding error: IA, TwIR, AA, and Handelman representations. Furthermore, for the first benchmark, 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 other benchmarks because it is unclear which variables would be best to split, whilst splitting all variables, as we shall see in Section VII-B, would require too large an execution time. These benchmarks 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 [11], [12] 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, given that 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 final range of the x values

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

// The i^{th} element of a vector v is indexed v^i as before, the j^{th} row of a matrix A is indexed $A(j)$

for $k = 1; k \leq 8; k++$ **do**
  for $j = 1; j \leq 5; j++$ **do**
    $x^j = (1-w)x^j + \frac{w}{A(j)^j}(b_j - \sum_{i=1, i\neq j}^{5} A(j)^i x^i)$
  end for
end for

// Initialisations
for $i = 1; i \leq 5; i++$ **do**
  $wDIVa\_i = w/A(i)^i$
  for $j = 1; j \leq 5; j++$ **do**
    $ASUBdiagA\_i^j = A(j)^i$
  end for
  $ASUBdiagA\_i^i = 0$
end for
$U\_SUBw = 1 - w$
for $k = 1; k \leq 8; k++$ **do**
  for $j = 1; j \leq 5; j++$ **do**
    $ax = ASUBdiagA^j \bullet x$
    $bSUBax = b^j - ax$
    $rhs = wDIVa^j \bullet bSUBax$
    $lhs = U\_SUBw \bullet x^j$
    $x^j = lhs - rhs$
  end for
end for

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

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

$v_1 = b - Ax_0$;
$\beta_1, \eta = ||v_1||_2$
$v_0, w_0, w_{-1} = [0000]'$
$\sigma_0, \sigma_1 = 0\ ; \gamma_0, \gamma_1 = 1$
for $i = 1; i \leq 3; i++$ **do**
  $v_i = \frac{v_i}{\beta_i}$
  $\alpha = v_i^T A v_i$
  $v_{i+1} = Av_i - \alpha v_i - \beta v_{i-1}$
  $\beta_{i+1} = ||v_{i+1}||_2$
  $\delta = \gamma_i \alpha_i - \gamma_{i-1}\sigma_i\beta_i$
  $\rho_1 = \sqrt{\delta^2 + \beta_{i+1}^2}$
  $\rho_2 = \sigma_i\alpha_i + \gamma_{i-1}\gamma_i\beta_i$
  $\rho_3 = \sigma_{i-1}\beta_i$
  $\gamma_{i+1} = \frac{\delta}{\rho_1}$
  $\sigma_{i+1} = \frac{\beta}{\rho_1}$
  $w_i = \frac{v_i - \rho_3 w_{i-2} - \rho_2 w_{i-1}}{\rho_1}$
  $x_i = x_{i-1} + \gamma_{i+1}\eta w_i$
  $\eta = -\sigma_{i+1}\eta$
end for

// Initialisations
for $i = 1; i \leq 4; i++$ **do**
  $Ax^i = A(i) \bullet x_0$
  $v_0^i = 0$
  $w_0^i = 0$
  $w_{-1}^i = 0$
end for
$v_1 = b - Ax$;
$tmp\_\beta = v_1^T \bullet v_1$
$\beta_1 = \sqrt{tmp\_\beta}$
$\sigma_0, \sigma_1 = 0\ ; \gamma_0, \gamma_1 = 1$
// Algorithm
for $i = 1; i \leq 3; i++$ **do**
  $v_i = \frac{v_i}{\beta_i}$
  for $j = 1; j \leq 4; j++$ **do**
    $Av^j = A(j) \bullet v_i$
  end for
  $\alpha = v_i \bullet Av$
  $\alpha V = \alpha \bullet v_i$
  $\beta V_{-1} = \beta_i \bullet v_{i-1}$
  $tmp\_v_{i+1} = \alpha V - \beta V_{-1}$
  $v_{i+1} = Av - tmp\_v_{i+1}$
  $tmp\_\beta = v_1 \bullet v_1$
  $\beta_{i+1} = \sqrt{tmp\_\beta}$
  $tmp1\_\delta = \gamma_i \bullet \alpha_i$
  $tmp2\_\delta = \gamma_{i-1} \bullet \sigma_i$
  $tmp2\_\delta = tmp2\_\delta \bullet \beta_i$
  $\delta = tmp1\_\delta - tmp2\_\delta$
  $tmp\_\rho_1 = \delta \bullet \delta$
  $tmp\_\rho_1 = tmp1\_\rho_1 + tmp\_\beta$
  $\rho_1 = \sqrt{tmp\_\rho_1}$
  $tmp1\_\rho_2 = \sigma_i \bullet \alpha_i$
  $tmp2\_\rho_2 = \gamma_{i-1} \bullet \gamma_i$
  $tmp2\_\rho_2 = tmp2\_\rho_2 \bullet \beta_i$
  $\rho_2 = tmp1\_\rho_2 + tmp2\_\rho_2$
  $\rho_3 = \sigma_{i-1} \bullet \beta_i$
  $\gamma_{i+1} = \frac{\delta}{\rho_1}$
  $\sigma_{i+1} = \frac{\beta_i}{\rho_1}$
  $tmp1\_w = \rho_3 \bullet w_{i-2}$
  $tmp2\_w = \rho_2 \bullet w_{i-1}$
  $tmp2\_w = tmp1\_w - tmp2\_w$
  $tmp1\_w = v_i - tmp2\_w$
  $w_i = \frac{tmp1\_w}{\rho_1}$
  $tmp\_x = \eta \bullet w_i$
  $tmp\_x = \gamma_{i+1} \bullet tmp\_x$
  $x_i = x_{i-1} + tmp\_x$
  $\eta = -\sigma_{i+1} \bullet \eta$
end for

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

```
// Input signal x is 64 samples of a sinusoid with magnitude of 1.
// FFT is radix-2 butterfly circuit
// x_re is real part of x.
// x_im is imaginary part of x.

// Algorithm
for j = 1; j ≤ 6; j++ do
  for k = 1; k ≤ 64; k++ do
    for l = 1; l ≤ 2^(j-1); l++ do
      xExp_re1 = x_re(2^(j-1)(2k-1)+l) • real(e^(-i2π(l-1)2^(6-j)/N));
      xExp_re2 = x_im(2^(j-1)(2k-1)+l) • (e^(-i2π(l-1)2^(6-j)/N);
      xExp_im1 = x_im(2^(j-1)(2k-1)+l) • e^(-i2π(l-1)2^(6-j)/N);
      xExp_im2 = x_re(2^(j-1)(2k-1)+l) • e^(-i2π(l-1)2^(6-j)/N);
      xExp_re(2^(j-1)(2k-1)+l) = xExp_re1 - xExp_re2;
      xExp_im2(2^(j-1)(2k-1)+l) = xExp_im1 + xExp_im2;
    end for
    for l = 1; l ≤ 2^(j-1); l++ do
      x_re(2^j(k-1)+l) = xExp_re(2^j(k-1)+l) + xExp_re(2^(j-1)(2k-1)+l);
      x_im(2^j(k-1)+l) = xExp_im(2^j(k-1)+l) + xExp_im(2^(j-1)(2k-1)+l);
      x_re(2^(j-1)(2k-1)+l) = xExp_re(2^j(k-1)+l) - xExp_re(2^(j-1)(2k-1)+l);
      x_im(2^(j-1)(2k-1)+l) = xExp_im(2^j(k-1)+l) - xExp_im(2^(j-1)(2k-1)+l);
    end for
  end for
end for
```

Figure 7. 64 Sample Radix-2 FFT using Cooley-Turkey. Code expressed using vector operations.

```
// Original signal sinusoid with a magnitude of 1.
// Received signal is original and echo with delayed by 3π/5 s with magnitude 2/5 of the original signal
// Samples performed every π/5 s
// Assume input signal measurement error is less than 0.1%
// Assume noise and measurement error is less than 1%
// Filter is of order 10

stepSize = 1;
for j=1; j ≤ noSteps; j++ do
    samp = origSign^j;
    for i = 1; k ≤ 10; k++ do
        ecSamp^i = ecSig^(j+(i-10));
    end for
    err = samp - w' * ecSamp;
    w = w + stepSize*ecSamp*err / (0.01 + ecSamp'*ecSamp);
end for
```

```
// Initialisations
stepSize = 1;
// Algorithm
for j=1; j ≤ noSteps; j++ do
  samp = origSig^j;
  for i = 1; k ≤ 10; k++ do
    ecSamp^i = ecSig^(j+(i-10));
  end for
  wEcSamp = w • ecSamp;
  err = samp - wEchoSamp;
  ecSampSq = ecSamp • ecSamp;
  denom = ecSampSq + 0.01;
  ecSampErr = ecSamp • error;
  ecSampErrStep = stepSize • ecSampErr;
  wUpdate = ecSampErrStep / denom;
  w = w + wUpdate;
end for
```

Figure 8. 10th Order NLMS Filter benchmark: inputs, original code and code expressed using vector operations.

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 latter tests involves products of polynomials or rational functions bounding intermediate variables, as well as the division and square root of a multivariate polynomial, and hence we use these benchmarks to demonstrate that our approach can also perform well in a more complex algorithms.

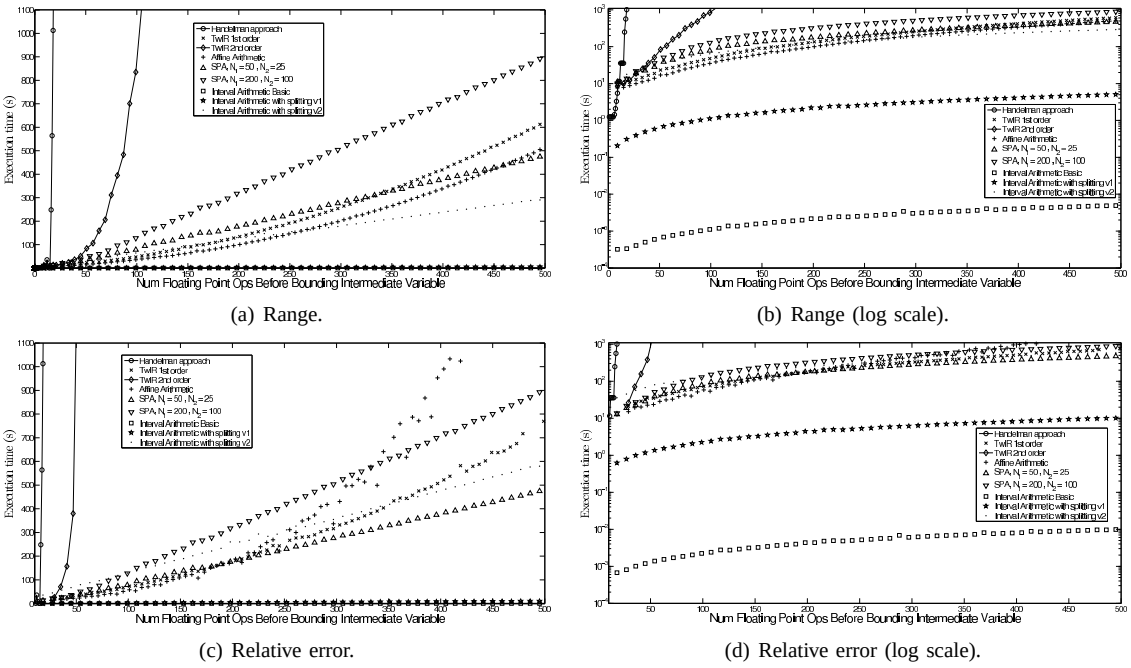
When performing polynomial approximations, for a fair comparison we use the same polynomial approximation method for affine arithmetic and our approach, with the exception of using Handelman representations to find the input range for this approximation, as mentioned in Section VI-B2. Throughout the remainder of this section, we use the name SPA (scalable precision analysis) to describe our approach.

B. Scalability

Figures 9(a), 9(b), 9(c) and 9(d) 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 benchmark.

For the range analysis case, seen in Figures 9(a) and 9(b), IA, 1st order TwIR, AA and SPA 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 IV. However, it is also clear that only IA and SPA have an execution time proportional to the number of operations, as expected from our analysis in Section IV, and this means that as the number of operations gets large, SPA can run faster than AA and TwIR. As we have previously mentioned, this is because SPA 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 IV, 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, while the execution time is still proportional to the number of operations, depending on the number of splits, there is a large difference between the execution time. 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 IV. 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 Figures 9(c) and 9(d), the execution time for AA and TwIR grows much faster whereas it remains similar for SPA. 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



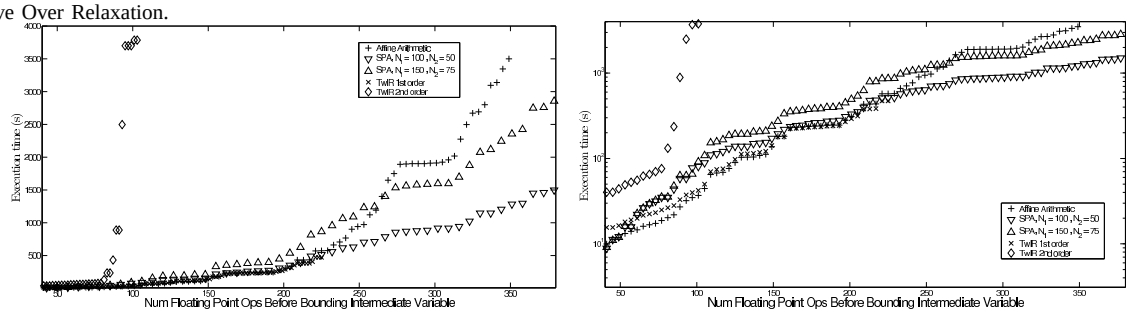
(a) Range.

(b) Range (log scale).

(c) Relative error.

(d) Relative error (log scale).

Figure 9. Execution time of various methods to bound the range and relative error of the intermediate variables after a given number of operations within a 5x5 Successive Over Relaxation.



(a) Range.

(b) Range (log scale).

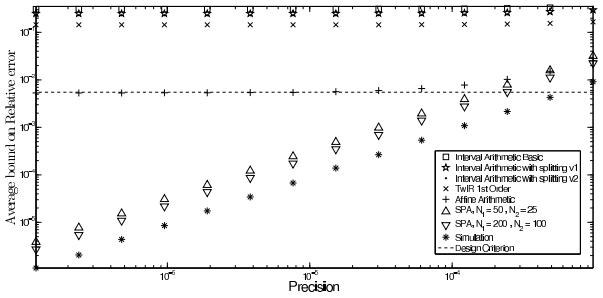
Figure 10. Execution time vs number of operations using various methods to bound range applied to a MINRES algorithm of a 4x4 matrix.

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 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.

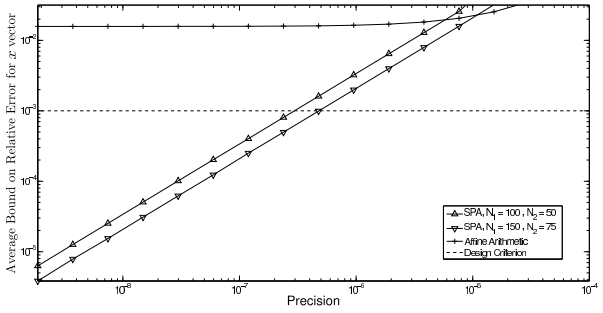
Figure 10 demonstrates how the execution time of each of the methods grow with the number of operations when computing the range for intermediate variables over the course of the MINRES algorithm. We note that 2nd order TwIR once again scales poorly with the number of operations, while IA and 1st order TwIR fail to compute bounds after 228 operations because they are unable to prove the input to the $\sqrt{}$ function is non-negative. For SPA and AA, in comparison to Figures 9(a) and 9(b), in this experiment the execution time grows much faster with the number of operations. This is because, once again, this algorithm contains the multiplication of two polynomials or rational functions which may be large, as opposed to the multiplication of a rational functions by a single input variable. As a result, because the maximum number of monomials in the product of two polynomials is given by the product of the number of monomials in each polynomial, 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 SPA, and is the reason SPA becomes faster than AA after much fewer operations than in Figures 9(a) and 9(b). We also comment that this graph grows in steps, unlike Figures 9(a) and 9(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. If we were to plot similar graphs for the FFT and NLMS benchmarks, because they also contain multiplication of two polynomials or rational functions, the graphs would be similar to Figure 10.

Table IV
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	0.023
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



(a) For Successive Over Relaxation benchmark.



(b) For MINRES benchmark

Figure 11. Average bound on relative error for x vector of various methods to calculate bounds.

C. Quality of bounds:

Table IV shows the computed bounds on ranges for the successive over relaxation benchmark for the methods that could calculate bounds in a tractable time. It is clear that SPA can compute tighter bounds than the existing approaches, in a reduced execution time, illustrating our claim that SPA can perform well where the existing methods perform well. However, for relative error analysis, shown in Figure 11(a), only SPA 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', SPA can retain these higher order terms. In addition, SPA calculates bounds on the worst case relative error that are close to the estimates found by simulation, implying our bounds are tight. In Figure 11(b), which shows the bounds on the average relative error over each of the output variables (the x-vector) for the MINRES example, we seen that once again, only SPA was capable of computing bounds for this benchmark that track how relative error decreases with increasing precision. Similar plots could be created for the other two benchmarks.

In order to demonstrate how this can be used to improve a hardware design, we created custom hardware implementations of the successive over relaxation and MINRES benchmarks. For the successive over relaxation circuit, our hardware implementation has a dedicated floating point operator for every floating point operation in line (3) of Figure 5, and the dot product is performed using a row of parallel multiplication units and an adder reduction tree, as in [4], where the A matrix is stored in RAM. A description of the parallel MINRES circuit can be found in [51].

Table V shows the resources required, post place and route on a Virtex 5 LX 330T, to create the successive over relaxation 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 SPA, we can create a hardware design with 25% less silicon area than by using AA, alternatively, our proof could justify performing this computation using single precision arithmetic in software to take advantage of the performance improvements over double precision. 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 SPA could save up almost 80% of the silicon area in comparison. Furthermore, because SPA 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. Similarly, Table VI shows the resource use for the MINRES benchmark required to satisfy a bound on relative error of less than 1×10^{-3} using the various methods to calculate bounds, 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 increased run-time, we obtain tighter bounds that result in a smaller hardware implementation, highlighting the trade-off we aim to achieve. Finally, once again, our software has the potential to obtain a proof that IEEE single precision is sufficient to satisfy the desired precision, illustrating how our tool could also be used to obtain performance improvements on other hardware platforms.

While these techniques may be beneficial when designing hardware to satisfy a specific design criteria, assuming a user had greater flexibility, an alternative use of these techniques could be to provide more information about the trade-offs between precision and error so as to enable the user to make an informed choice. Figure 12 illustrates this trade-off, showing the number of slices and DSPs required to meet a desired bound on the relative error seen at the output for the FFT benchmark, as calculated by SPA with $N_1 = 200$ and $N_2 = 100$. We have not plotted this graph with other methods, because as discussed above, only SPA can track how relative error decreases with increasing precision. The resource figures are for a single radix-2 butterfly circuit, which is general enough to perform the entire algorithm. As one would expect, to obtain a tighter bound on relative error, the internal operators require a higher level of precision and therefore consume a larger amount of area. We note that for this benchmark, the resource use for a parallel implementation with many butterfly circuits would have a similar shape, but would scale according to the chosen number of butterfly circuits. As such, this graph is particularly useful in this benchmark because it could assist a user in choosing the amount of precision and level of parallelism for a fixed hardware budget.

Whereas in the previous examples, it was the bound on the

Table V

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, AND EXECUTION TIME FOR EACH TECHNIQUE TO GENERATE PROOF CHOICE OF PRECISION WILL SATISFY DESIGN CRITERION.

Method	Precision (# bits)	Slices	Frequency (MHz)	Execution time of bounding procedure (s)
Simulation	11	1252	330	N/A
SPA, $N_1 = 200, N_2 = 100$	13	1609	330	1300
SPA, $N_1 = 50, N_2 = 25$	14	1707	330	600
AA	18	2154	300	5000
IA/TwIR	∞	∞	N/A	N/A
IEEE Single Precision	24	2681	280	N/A
IEEE Double Precision	52	7983	251	N/A

Table VI

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, AND EXECUTION TIME FOR EACH TECHNIQUE TO GENERATE PROOF CHOICE OF PRECISION WILL SATISFY DESIGN CRITERION.

Method	Precision (# bits)	Slices	Frequency (MHz)	Execution time of bounding procedure (s)
SPA $N_1 = 150, N_2 = 75$	21	8213	225	2500
SPA $N_1 = 100, N_2 = 50$	22	8371	225	3200
AA/IA/TwIR	∞	∞	N/A	N/A
IEEE Single Precision	24	7592	150	N/A
IEEE Double Precision	52	25965	120	N/A

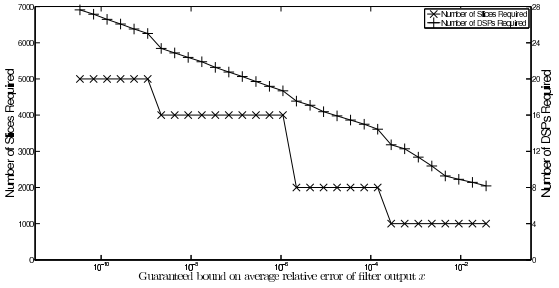


Figure 12. Resource use required to satisfy different bound on mean relative error of outputs of FFT.

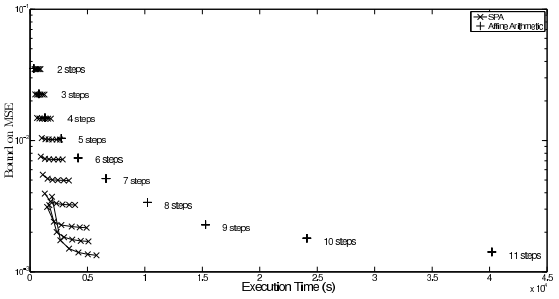


Figure 13. Execution time vs bound on mean square error for various numbers of steps using various methods to bound range applied to a 10th order NLMS filter.

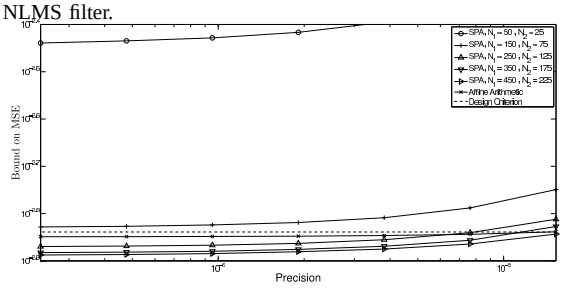


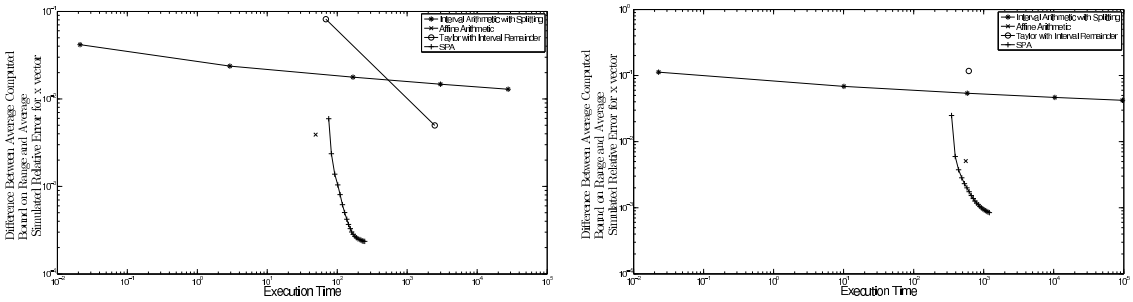
Figure 14. Precision vs bound on mean square error after 10 steps using various methods to bound range applied to a 10th order NLMS filter.

relative error that was of interest, in the case of an NLMS filter for echo cancellation, a typical aim may be to ensure that the mean square error of the filter lies below a certain threshold after a given number of steps, in the presence of input noise and finite precision errors. Figure 13 plots the bound on the mean square error after every step of the algorithm for AA and our approach against the execution time for a precision of 30 bits; for SPA, we plot results with N_1 varying from 50 to 300 in steps of 50 and N_2 set to $N_2 = \frac{1}{2}N_1$. We restricted the figure to only compare these two approaches because the other techniques were incapable of computing comparable bounds, as seen in the previous tests. Similar to the MINRES example, because there is multiplication of intermediate variables which are bounded using large polynomials, the execution time for this operation becomes very large after many floating point operations when using AA, whereas SPA can compute tighter bounds in a smaller execution time. This is reflected in Figure 13, which shows that as the number of steps increases, SPA can have an execution time that is an order of magnitude less than affine arithmetic whilst computing comparable bounds, and also grows at a much slower rate. This means that using SPA, we could examine the MSE after many further steps. This figure also once again illustrates how our technique can trade quality of bounds with execution time, most interestingly when SPA is set to $N_1 = 50, N_2 = 25$, it appears to have a higher MSE after 10 iterations than after 9; this is because without sufficient terms retained, when we group monomials we lose dependency information and eventually, many first order dependencies will be lost and result in wider bounds.

To illustrate its use for hardware design, suppose we had the aim to design an echo cancellation device using an NLMS filter that ensured the mean square error for a signal would lie below a threshold of 1.45×10^{-3} using as little silicon area as possible. Using Figure 13, with a high precision, we can ensure that after 11 steps, the NLMS filter will satisfy the desired design criterion. However, if we subsequently analyse the same algorithm with different precisions, as shown in Figure 14 where the desired design criteria is highlighted using a dotted line, we see that we can meet this design criterion with reduced precision. Once again, this could translate into significant silicon area savings, as illustrated in Table VII.

Table VII
SLICE USE AND MAX FREQUENCY OF 10TH ORDER NLMS FILTER IMPLEMENTATION REQUIRED ACCORDING TO ANALYTICAL TOOLS TO GUARANTEE THE RELATIVE ERROR IS LESS THAN 1.45×10^{-3}, OR USING IEEE STANDARDS, AND EXECUTION TIME FOR EACH TECHNIQUE TO GENERATE PROOF CHOICE OF PRECISION WILL SATISFY DESIGN CRITERION.

Method	Precision (# bits)	Slices	Frequency (MHz)	Execution time of bounding procedure (s)
SPA $N_1 = 450, N_2 = 225$	15	3028	300	11000
AA	16	3217	300	40000
SPA $N_1 = 350, N_2 = 175$	16	3217	300	6600
SPA $N_1 = 250, N_2 = 125$	17	3438	300	5000
IA/TwIR	∞	∞	N/A	N/A
IEEE Single Precision	24	5225	300	N/A
IEEE Double Precision	52	16046	120	N/A



(a) After two iterations. (b) After seven iterations.

Figure 15. Trade off between bound on range and execution time of various implementations of SPA to find the average bound on the range of x vector of a successive over relaxation of a 5x5 matrix for various approaches.

D. Execution time vs quality of bounds performance trade-off:

Figure 15 compares the ability of SPA 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 SPA 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 15(a), we restricted the successive over relaxation benchmark 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 15(b) we plot the same after the conclusion of the benchmark. For SPA, we have varied N_1 in 10 equal size increments from 20 to 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 hence it can only create a single point on each graph – and TwIR has a large difference in execution time between 1st and 2nd order. In addition, this graph shows once again that SPA 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 15(a), due to the additional overheads of our algorithm in creating rational functions representing the value for intermediate variables, AA can run quicker than SPA, as the number of operations becomes larger as in Figure 15(a), because AA cannot trade execution time for quality of bounds, SPA has the flexibility to run quicker than AA at a cost of bounds, or for longer than AA to obtain tighter bounds.

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 track the effect of finite precision errors and compute tight bounds on the relative error introduced by the use of finite precision arithmetic. This enables us to understand the trade-offs between precision and parallelism, or to design hardware that meets a given relative error specification with less silicon area; in the case of the successive over relaxation and the NLMS filter examples, saving over 80% of the slices in comparison to IEEE 754 double precision, for the MINRES example, saving over 65% of the slices in comparison to IEEE 754 double precision.

VIII. 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.

REFERENCES

[1] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2004, pp. 171–180.

[2] S. Kestur, J. Davis, and O. Williams, "Blas comparison on fpga, cpu and gpu," in *IEEE Computer Society Annual Symposium on VLSI*, 2010, pp. 288 –293.

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

[6] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," in *Proc. ACM/IEEE conference on Supercomputing*, 2006, p. 113.

[7] Nvdia, "Tesla c1060 computing processor board," Tech. Rep., January 2007. [Online]. Available: http://www.nvidia.com/docs/IO/43395/BD-04111-001_v05.pdf

[8] D. Boland and G. Constantinides, "A scalable approach for automated precision analysis," in *Proc. int. symp. on Field Programmable Gate Arrays*, 2012, pp. 185–194.

[9] 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.

[10] 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.

[11] A. B. 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.

[12] 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.

[13] 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, 2010.

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

[15] 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.

[16] 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.

[17] 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, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[18] B. Fisher, *Polynomial Based Iteration Methods for Symmetric Linear Systems*. Baltimore, MD, USA: Wiley, Teubner, 1996.

[19] J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, no. 90, pp. 297–301, 1965.

[20] M. T. Heath, *Scientific Computing*. McGraw-Hill Higher Education, 2001.

[21] J. M. Maciejowski, *Predictive control with constraints*. Essex, England: Prentice Hall, 2002.

[22] E. Biglieri, R. Calderbank, A. Constantinides, A. Goldsmith, A. Paulraj, and H. V. Poor, *MIMO Wireless Communications*. Cambridge University Press, 2007.

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

[24] W. Sung and K.-I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE Trans. on Signal Processing*, vol. 43, no. 12, pp. 3087 –3090, 1995.

[25] K.-I. Kum and W. Sung, "Word-length optimization for high-level synthesis of digital signal processing systems," in *IEEE Workshop on Signal Processing Systems*, 1998, pp. 569 –578.

[26] ——, "Combined word-length optimization and high-level synthesis of digital signal processing systems," *IEEE Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 20, no. 8, pp. 921–930, 2001.

[27] M.-A. Cantin, Y. Savaria, D. Prodanos, and P. Lavoie, "An automatic word length determination method," in *Proc. Int. Symp. on Circuits and Systems*, vol. 5, 2001, pp. 53 –56 vol. 5.

[28] M.-A. Cantin, Y. Savaria, and P. Lavoie, "A comparison of automatic word length optimization procedures," in *Proc. Int. Symp. on Circuits and Systems*, vol. 2, 2002, pp. II–612 – II–615 vol.2.

[29] S. Roy and P. Banerjee, "An algorithm for trading off quantization error with hardware resources for matlab-based fpga design," *IEEE Trans. on Computers*, vol. 54, no. 7, pp. 886 – 896, 2005.

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

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

[32] M. L. Chang and S. Hauck, "Précis: A design-time precision analysis tool," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2002, pp. 229–238.

[33] H. Keding, M. Willems, M. Coors, and H. Meyr, "Fridge: a fixed-point design and simulation environment," in *Proc. Conf. on Design, Automation and Test in Europe*, 1998, pp. 429–435.

[34] M. Willems, V. Bursgens, H. Keding, T. Grotker, and H. Meyr, "System level fixed-point design based on an interpolative approach," in *Proc. Design Automation Conference*, 1997, pp. 293 –298.

[35] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens, "A methodology and design environment for DSP ASIC fixed point refinement," in *Proc. Conf. on design, automation and test in Europe*, 1999, pp. 271 –276.

[36] A. A. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang, "Automating customisation of floating-point designs," in *Proc. Int. Conf. on Reconfigurable Computing Is Going Mainstream, Field-Programmable Logic and Applications*, ser. FPL '02, 2002, pp. 523–533.

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

[38] 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.

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

[40] 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.

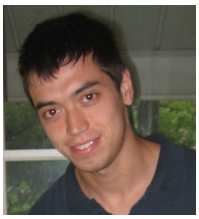
[41] L. H. de Figueiredo and J. Stolfi, *Self-Validated Numerical Methods and Applications*. IMPA/CNPq, Rio de Janeiro, Brazil, 1997.

[42] 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.

[43] D.-U. Lee, A. A. Gaffar, O. Mencer, and W. Luk, "Minibit: bit-width optimization via affine arithmetic," in *Proc. Design Automation Conference*, 2005, pp. 837–840.

[44] W. Osborne, R. Cheung, J. Coutinho, W. Luk, and O. Mencer, "Automatic accuracy-guaranteed bit-width optimization for fixed and floating-

point systems," in *Int. Conf. on Field Programmable Logic and Applications*, 2007, pp. 617–620.
[45] G. Caffarena, C. Carreras, J. A. López, and A. Fernández, "Sqnr estimation of fixed-point dsp algorithms," *EURASIP J. Adv. Signal Process*, vol. 2010, pp. 21:1–21:12, 2010.
[46] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*. Birkhauser, 2005.
[47] G. Constantinides, P. Cheung, and W. Luk, "The multiple wordlength paradigm," in *Proc. IEEE Symp. on Field-Programmable Custom Computing Machines*, 2001, pp. 51 –60.
[48] D. Menard, R. Rocher, and O. Sentieys, "Analytical fixed-point accuracy evaluation in linear time-invariant systems," *IEEE Trans. on Circuits and Systems I*, vol. 55, no. 10, pp. 3197 –3208, 2008.
[49] W. S. Brown, "A simple but realistic model of floating-point computation," *ACM Trans. Math. Softw.*, vol. 7, pp. 445–480, 1981.
[50] H. Ratschek, "Centered forms," *SIAM Journal on Numerical Analysis*, vol. 17, no. 5, pp. pp. 656–662, 1980.
[51] D. Boland and G. Constantinides, "An FPGA-based implementation of the MINRES algorithm," in *Proc. Int. Conf. Field Programmable Logic and Applications*, 2008, pp. 379–384.



David Boland David Boland received an M.Eng. (with honors) degree in information systems engineering from Imperial College London, London, U.K. in 2007, and received his Ph.D. degree in the Circuits and Systems research group in the electrical and electronic engineering department at Imperial College London in 2011. His research interests include numerical analysis, optimisation, design automation, and iterative algorithms.

George A. Constantinides (S.96-M.01-SM.08) received the M.Eng. degree (with honors) in information systems engineering and the Ph.D. degree from Imperial College London, London, U.K., in 1998 and 2001, respectively. Since 2002, he has been with the faculty at Imperial College London, where he is currently Reader (Associate Professor) in Digital Systems and Head of the Circuits and Systems research group. Dr. Constantinides is a Fellow of the BCS and a Senior Member of the IEEE. He is an Associate Editor of the IEEE Transactions on Computers and the Journal of VLSI Signal Processing. He was a Program Cochair of the IEEE International Conference on Field-Programmable Technology in 2006 and Field Programmable Logic and Applications in 2003, and is a member of the steering committee of the International Symposium on Applied Reconfigurable Computing. He serves on the technical program committees of several conferences, including DAC, FPT and FPL.