Automated Precision Analysis: A Polynomial Algebraic Approach

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

***Abstract*—When migrating an algorithm onto hardware, the potential saving that can be obtained by tuning the precision used in the algorithm to meet a range or error specification is often overlooked; the major reason is that it is hard to choose a number system which can guarantee any such specification can be met. Instead, the problem is mitigated by opting to use IEEE standard single or double precision so as to be 'no worse' than a software implementation. However, the flexibility in the number representation is one of the key factors that can only be exploited on FPGAs, unlike GPUs and general purpose processors, and hence ignoring this potential significantly limits the performance achievable on an FPGA. To this end, this paper describes a tool which analyses algorithms with given input ranges under a finite precision to provide information that could be used to tune the hardware to the algorithm specifications. We demonstrate the proposed procedure on an iteration of the conjugate gradient algorithm, achieving a reduction in slices of over 40% when meeting the same error specification found by traditional methods. We also show it achieves comparable bounds to recent literature in a small fraction of the execution time, with greater scalability.**

I. Introduction

The reconfigurable hardware community has produced various accelerators for a wide range of applications, demonstrating that through exploiting the inherent parallelism within algorithms, it is possible to achieve significant performance improvements over general purpose processors (GPPs). When implementing these algorithms using hardware, with the notable exception of digital signal processing applications [1], most approaches are based upon simply using floating point cores in hardware. Whilst this is a suitable paradigm for GPUs, given that they largely consist of many parallel floating point units, on FPGAs, this is significantly limiting the flexibility in that FPGAs have the ability to implement any precision required to meet a given specification. Furthermore, one should note that even in the GPU space, though people often try to improve the performance by moving from double to single precision, rarely is this move accompanied by a proof of the numerical stability. Ideally, one requires an automated method to calculate a proof of numerical stability when using single precision for any given algorithm. However, any such method could similarly be used to calculate a proof for the *minimum* precision required to ensure numerical stability, and this would hand the advantage back toward FPGAs.

The motivation behind this paper is to work towards removing this limitation by ensuring that any comparison of performance should be for meeting a given design criterion, as opposed to implementing the equivalent operations of software in hardware; such a design criterion may be an error or range specification so as to ensure convergence or a given safety margin. The reason that, to date, such comparisons are rarely made is that there is no known method that can exactly and tractably calculate tight bounds for the error or range of any variable within an algorithm, given that they are affected by both input ranges and finite precision effects; existing procedures either sacrifice tightness or tractability.

The effect of using a finite precision number system is that rounding will often occur in order to represent values. Whilst the error introduced by the rounding of any single value may be small, over the course of an algorithm the accumulation of these errors can cause a significant deviation from the nominal result. Though, in general, one can reduce this error by increasing the precision, such an action would cause a decrease in performance; this is true in software as well as hardware. As an example, recent figures for the difference in performance, in terms of peak theoretical FLOPS, between single and double precision is approximately a factor of 2 for a CPU [2], 9 for a GPU [3] and 14 for the IBM Cell multiprocessor [4]. However, for hardware platforms such as FPGAs or ASICs, not only is there a larger choice for precision, the choice of precision will affect far more factors: the silicon area, clock speed, latency, memory use and data transfer. This makes choosing the optimum precision even more important.

The FPGA community has taken considerable interest in exploiting the freedom to customise the precision used throughout an algorithm. For example, precision was highlighted as one of the key factors, along with parallelism and I/O, when estimating the amenability of an application to hardware in the high level RAT Methodology [5], and furthermore, there has been research into the potential benefits of fine-tuning an algorithm on an FPGA taking into effect precision, notably in the DSP domain [6]. However, whilst there exist sophisticated models to estimate the error of a system from which it is possible to optimise word-lengths for an algorithm in DSP, in general, there is a need for methods to calculate tight bounds on variables in an

algorithm so as to perform similar word-length optimisation for an arbitrary algorithm, as well as to make best use of tools such as those in [5].

This paper describes a new method which takes ranges of input variables for an algorithm and calculates bounds on the range of any other variable within an algorithm under a given finite precision representation. The bounds this method provides are shown, in general, to be superior in comparison to existing methods; this in turn can be used to create hardware that still achieves the same error specification as existing methods, but using less silicon area. The claim of superior bounds is justified against comparative examples from existing literature, whilst its potential use for word-length optimisation is demonstrated on a 'toy' floating point conjugate gradient example. The majority of this paper discusses floating point algorithms, largely due to the growing collection of publications of floating point hardware implementations [7], [8]. We note, however, that the background theory and heuristic described in this work could easily be adapted to word-length optimisation for fixed point designs by using a different model of error. A summary of the main contributions of this work are as follows:

- a method to find provable bounds for any variables within an algorithm given input data ranges and a finite precision,
- results demonstrating that applying our approach to a 'toy' example of the conjugate gradient algorithm, in comparison to using the standard approach of interval arithmetic [9], finds bounds that could translate to a reduction in slices of over 40%,
- results that demonstrate our approach can calculate bounds which are comparable to a sophisticated existing solver for range analysis [10], in a small fraction of the compute time.

This paper is organised as follows. It first discusses in detail the current literature regarding the methods used to calculate error bounds for use in word-length optimisation in Section II. The new approach, along with a description of the source of floating point error and a general method to represent this error using polynomials, as well as some relevant background theory, is then put forward in Section III. Section IV discusses the methods to test the results, which are then shown in the following section, Section V. Finally, Section VI discusses the conclusions of this work and its limitations that must be addressed in the future.

II. Background

Due to the potential benefits on silicon area, clock speed and power that can be obtained by optimising the precision used throughout a circuit, there exists a large amount of literature focused upon word-length optimisation, summaries of which can be found in [11], [12], as well as a class of work in practical tools for precision analysis [13]. One fundamental aspect of these solvers is that they require a method which can both validate that overflow will not occur for a chosen word-length and calculate the cost of the chosen word-length in terms of the error observed in the final computational result. It is interesting that despite the wealth of optimisation strategies, there are only a handful of methods to perform this function. These methods will be elaborated in this section.

The most straightforward way to estimate an error is through simulation. The aim of any simulation-based approach is to find the inputs which 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, and hence the simulation will either require random sampling of the input data space [14], statistical profiling [15] or be based upon a representative training data set [16]. However, whilst the quality of the estimate can be improved by increasing the size of the training set or the search time, in either case, the estimate does not form a bound because corner cases can be missed.

In contrast to simulation, to calculate true bounds for general algorithms, traditionally there have only been two main analytical approaches: interval arithmetic (IA) [9] or affine arithmetic (AA) [17]. Interval arithmetic represents every value as lying within some interval, $[x_1, x_2]$, then propagates these bounds through an algorithm by calculating worst case bounds for every individual operation. This approach suffers from the so-called dependency problem, where if the same variable is used twice, the correlation is ignored; several simple examples can demonstrate how this problem may cause bounds that are significantly wider than the tightest bounds [18]. These problems have led to an active community of researchers in *robust computing*, who have developed ways to mitigate this problem by modifying algorithms to make them 'interval arithmetic friendly' at the cost of computational complexity or average case numerical robustness [19]. However, whilst such approaches are useful to obtain reliable proofs using IA, the modifications to the algorithm do not necessarily improve the true numerical stability, indeed in some cases it may make this stability worse, instead the modifications simply reduce the sensitivity to which IA bounds this error. Ideally, it is preferable to find proofs of tighter bounds for the unmodified algorithm, unless it can be proven that the true numerical properties of the modified algorithm have been improved.

Affine arithmetic is an alternative method which mitigates the dependency problem by representing every variable in an affine form which keeps track of interdependencies [20]. However, many functions, including general multiplication, are not affine and hence approximations must be made and these cause a widening of the derived bounds.

Given the limitations of interval and affine arithmetic, recently, a new approach has been published which uses Satisfiability-Modulo Theories (SMT) [10] to refine the bounds given by interval or affine arithmetic by searching

for a set of inputs breaking the bounds, using a Boolean Satisfiability solver. This bound is refined over several iterations depending upon the results of this test using a binary search method. The main problem with this approach is that the run time grows considerably with the problem complexity [10].

This run-time issue limits the SMT approach to consider only the so called 'range analysis' problem which involves ensuring that over the range of input data, there is sufficient dynamic range to prevent overflow. However, when optimising word-lengths, determining the range as a result of the inputs is only a part of the problem; it is also important to perform 'precision analysis', which involves ensuring that the error at the output, caused by the use of a finite precision, lies below a threshold. Although several of the optimisation strategies do perform range analysis as a first step before precision analysis [12], these are all targeted towards DSP systems which do not include division. We argue that for a word-length optimiser targeted towards a general algorithm, these problems cannot be viewed separately because due to division, the range can be heavily affected by the errors caused by the use of finite precision, typically because a divisor approaches zero. Such effects will be seen in Section V-A. Furthermore, due to the significantly larger dynamic range of data in floating point, the issue of range analysis is often only of interest in fixed point designs.

This work describes a new general analytical approach which can provide provable bounds. It is argued in this paper that this method can achieve significantly tighter bounds than both interval and affine arithmetic, while running significantly faster, with better scalability, than the SMT approach.

III. Proposed Algorithm

The aim of this work is, for a given floating point precision, to find bounds on the value of any chosen variable within an algorithm, and hence bound the worst case computational error induced. The suggested method to achieve this consists of several stages. The first stage involves creating a polynomial for the variable of interest, a function of new bounded variables, each representing roundoff errors introduced after an operation. This polynomial is then simplified into a specific canonical form, from which bounds for the extrema of the polynomial can be found algorithmically. An optional final stage extends the approach for polynomials to rational functions, allowing all algorithms consisting of the basic operators $\{+,-,*,/\}$ to be automatically analysed.

1) Notation: To formalise the discussion of the method, some simple notation is used. We consider polynomials in n variables $\delta_1, \delta_2, ..., \delta_n$. We use the notation $\boldsymbol{\delta}^{\boldsymbol{\lambda}}$ for a term, which is a product of the variables raised to some integer powers (1), where $\boldsymbol{\lambda}$ is a vector collecting the exponents (2). We define by $|\boldsymbol{\lambda}|$ the *degree* of the term, given by (3). A monomial is defined as a term multiplied by some real coefficient, *i.e.* $c\boldsymbol{\delta}^{\boldsymbol{\lambda}}$, and a polynomial is the sum of one or more monomials.

$$\boldsymbol{\delta}^{\boldsymbol{\lambda}} = \delta_1^{\lambda_1}\delta_2^{\lambda_2}...\delta_n^{\lambda_n}. \quad (1)$$

$$\boldsymbol{\lambda} = (\lambda_1, ..., \lambda_n), \text{where } \lambda_i \in \mathbb{N}. \quad (2)$$

$$|\boldsymbol{\lambda}| = \lambda_1 + ... + \lambda_n. \quad (3)$$

A. Creating a Polynomial Representation of Potential Range

It can be shown that for a real value x, the closest floating point approximation $\hat{x}$ of x can be expressed as in (4) [21], where m is the number of mantissa bits used (referred to as the precision). It is similarly possible to specify that the floating point result of any scalar operation ($\odot \in \{+,-,*,/\}$) is bounded as in (5), provided the exponent is sufficiently large to span the range of the result. Operations complying with IEEE standard arithmetic exhibit this behaviour.

$$\hat{x} = x(1+\delta_1) \quad (|\delta_1| \leq \Delta, \text{where } \Delta = 2^{-m}). \quad (4)$$

$$\widehat{x \odot y} = (x \odot y)(1+\delta_1). \quad (5)$$

In our approach, a simple compiler is written which takes input pseudo code and applies this model of floating point error on the result of every computation throughout an algorithm such that every output variable can be represented by a single polynomial in all the error variables, as shown for a simple example in Table I.

Table I
Construction of polynomials

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

B. Minimising the polynomial

Given a polynomial $f(\boldsymbol{\delta})$ representing the value of a variable, as in Table I, we want to find $\gamma_{lower} = \inf_{|\delta_i|\leq\Delta} f(\boldsymbol{\delta})$, the lower bound on the variable or function of intent, and $\gamma_{upper} = \sup_{|\delta_i|\leq\Delta} f(\boldsymbol{\delta})$. Unfortunately this is a non-convex optimisation problem, which is NP-hard. As a result, we focus on finding a computationally tractable lower bound $\hat{\gamma}_{lower} \leq \gamma_{lower}$ and upper bound $\hat{\gamma}_{upper} \geq \gamma_{upper}$.

In this work, a new approach is taken to find these bounds such that $\gamma_{lower} - \hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper} - \gamma_{upper}$ are as small as possible. This section first describes the background theory, which is based upon a result from real algebra, after which a new heuristic based upon this theory is proposed.

1) Background: This work is based upon a result emanating from real algebra discovered by Handelman.

Theorem 1. [22] *a polynomial $p(x)$ is positive in the interior of a compact set of linear inequalities g_i, i.e. over*

the set $S = \{x \in \mathbb{R}^n | g_i(x) \geq 0\}$*, if and only if* p *has a* Handelman representation *of the form* (6)*.*

$$p = \sum_{\alpha \in \mathbb{N}^n} c_\alpha \prod_{i=1}^{m} g_i^{\alpha_i}, \tag{6}$$

where each c_α is a positive constant and $\mathbb{N}$ is the set of natural numbers.

This theorem can be applied to find lower and upper bounds to satisfy $\hat{\gamma}_{lower} \leq f(\boldsymbol{\delta}) \leq \hat{\gamma}_{upper}$ by considering that we are trying to show that the functions $f(\boldsymbol{\delta}) - \hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper} - f(\boldsymbol{\delta})$ are positive over the compact set of inequalities specifying the bounds on $\boldsymbol{\delta}$ given in (7). By Theorem 1, this is equivalent to showing $f(\boldsymbol{\delta}) - \hat{\gamma}_{lower}$ has a Handelman representation of the form (8), or similarly satisfying (9) for the upper bound.

$$S = \{\boldsymbol{\delta} \in \mathbb{R}^n | \forall i (\Delta - \delta_i \geq 0) \wedge (\Delta + \delta_i \geq 0)\}. \tag{7}$$

$$f(\boldsymbol{\delta}) - \hat{\gamma}_{lower} = \sum_{\alpha,\beta \in \mathbb{N}^n \times \mathbb{N}^n} c_{\alpha,\beta} \prod_{i=1}^{n} (\Delta - \delta_i)^{\alpha_i} (\Delta + \delta_i)^{\beta_i}. \tag{8}$$

$$\hat{\gamma}_{upper} - f(\boldsymbol{\delta}) = \sum_{\alpha,\beta \in \mathbb{N}^n \times \mathbb{N}^n} c_{\alpha,\beta} \prod_{i=1}^{n} (\Delta - \delta_i)^{\alpha_i} (\Delta + \delta_i)^{\beta_i}. \tag{9}$$

For our purposes, it will be easier to work with a generalised version of the Handelman representation that we propose, given in (10), which we refer to as a *Generalised Handelman Representation* (GHR).

Theorem 2. *A polynomial* p_{ghr} *is positive on the interior of the compact set* S*, defined in* (7)*, if and only if* p_{ghr} *can be represented by a* Generalised Handelman Representation *of the form* (10)*.*

Proof: If p_{ghr} is positive on a compact set S, then it has a Handelman representation; by choosing $\mu_{i,j} = i$, it also has a GHR. If p_{ghr} has a GHR, then because any individual variable $\delta_i \in S$ is bounded by $|\delta_i| \leq \Delta$, then any term can also be bounded, $|\boldsymbol{\delta}^{\boldsymbol{\lambda}}| \leq \Delta^{|\boldsymbol{\lambda}|}$. This means $\Delta^{|\boldsymbol{\lambda}|} + \boldsymbol{\delta}^{\boldsymbol{\lambda}} \geq 0$ and $\Delta^{|\boldsymbol{\lambda}|} - \boldsymbol{\delta}^{\boldsymbol{\lambda}} \geq 0$ and because $c_i > 0$, it holds that $p_{ghr} \geq 0$, and positive in the interior of S. ■

$$p_{ghr} = \sum_{j \in \mathbb{N}} c_j \prod_{i=1}^{n} (\Delta^{|\boldsymbol{\mu_{i,j}}|} - \boldsymbol{\delta}^{\boldsymbol{\mu_{i,j}}})^{\alpha_{i,j}} (\Delta^{|\boldsymbol{\mu_{i,j}}|} + \boldsymbol{\delta}^{\boldsymbol{\mu_{i,j}}})^{\beta_{i,j}},$$

where $\mu_{i,j}$ are arbitrary integer vectors. (10)

Using this theorem, if we can find GHRs to satisfy (11) and (12), then we know a Handelman representation also exists for the left-hand side of these equations, and hence again by Theorem 1, the left-hand side is positive over the set of inequalities (7), and from this a guaranteed bound follows.

$$f(\boldsymbol{\delta}) - \hat{\gamma}_{lower} = p_{lower\_ghr}. \tag{11}$$

$$\hat{\gamma}_{upper} - f(\boldsymbol{\delta}) = p_{upper\_ghr}. \tag{12}$$

2) Example: In order to demonstrate the use of this theory, we will consider the function $f(\delta_1) = \delta_1^2 - \delta_1$ over the set $|\delta_1| \leq 1/2$. For this simple function of one variable, we can first derive lower and upper bounds using familiar calculus arguments. We will then and demonstrate that IA is unable to calculate the same bounds, before finally showing that GHRs can be used to prove the ideal bounds.

Using calculus to calculate bounds of $f(\delta_1) = \delta_1^2 - \delta_1$, we know that because this is a convex function, the minimum will lie where the derivative $f'(\delta_1) = 0$ and the maximum will lie at one of the extremes of the range of δ_1. Thus by differentiating $f(\delta_1)$ to get $f'(\delta_1) = 2\delta_1 - 1$, we find the minimum lies at $\delta_1 = 1/2$, leaving the maximum to be where $\delta_1 = -1/2$; this gives the range of $f(\delta_1)$ to be $[-1/4, 3/4]$. In comparison, interval arithmetic is unable to calculate such the true bounds, as shown in (13).

$$\begin{aligned} \delta_1 &\in [-1/2, 1/2] \\ \delta_1^2 &\in [-1/4, 1/4] \\ \delta_1^2 - \delta_1 &\in [-3/4, 3/4] \end{aligned} \tag{13}$$

To find bounds using the theory presented in each section, we want to search for GHRs to satisfy (11) and (12). Two such GHRs are $p_{lower\_ghr} = (1/2 - \delta_1)^2$ and $p_{upper\_ghr} = (1/2 - \delta_1)(1/2 + \delta_1) + (1/2 + \delta_1)$. After equating these respective functions, as shown in (14) and (15), we find $\hat{\gamma}_{lower} = -1/4$ and $\hat{\gamma}_{upper} = 3/4$. Finally, by Theorem 2, we now know that $f(\delta_1) - (-1/4) \geq 0$ and $3/4 - f(\delta_1) \geq 0$ or that $-1/4 \leq f(\delta_1) \leq 3/4$.

$$\begin{aligned} \delta_1^2 - \delta_1 - \hat{\gamma}_{lower} &= (1/2 - \delta_1)^2 \\ \delta_1^2 - \delta_1 - \hat{\gamma}_{lower} &= 1/4 - \delta_1 + \delta_1^2 \\ \hat{\gamma}_{lower} &= -1/4 \end{aligned} \tag{14}$$

$$\begin{aligned} \hat{\gamma}_{upper} - (\delta_1^2 - \delta_1) &= (1/2 - \delta_1)(1/2 + \delta_1) + (1/2 + \delta_1) \\ \hat{\gamma}_{upper} - \delta_1^2 + \delta_1 &= 1/4 - \delta_1^2 + 1/2 + \delta_1 \\ \hat{\gamma}_{upper} &= 3/4 \end{aligned} \tag{15}$$

Unfortunately, in general it is difficult to find GHRs to satisfy equations (11) and (12) and ensure that the calculated bounds for $\hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper}$ are as tight as possible. In the next section we propose a heuristic which is guaranteed to terminate at the same time as aiming to find practically useful bounds.

3) Our approach: The proposed algorithm to bound a polynomial, given in Figure 1, is based upon finding GHRs to satisfy equations (11) and (12), similar to the above example. To achieve this, we first completely expand the polynomials of $f(\boldsymbol{\delta})$ to separate all the monomials. In order to demonstrate this step, consider the earlier example from Table I which can be expanded into the polynomial (16), when neglecting the variable δ_3, since the worst case value of this variable is trivially known to lie at the extremes.

$$\begin{aligned} f(\boldsymbol{\delta}) = &-xy(xy-1) - xy(2xy-1)\delta_1 - x^2y^2\delta_2 \\ &-x^2y^2\delta_1^2 - 2x^2y^2\delta_1\delta_2 - x^2y^2\delta_1^2\delta_2 \end{aligned} \tag{16}$$

After this expansion, it is easily possible to 'cancel' each

Algorithm $\gamma = BoundPoly(g, \Delta)$
// BoundPoly takes a polynomial g in the formal vector variable $\boldsymbol{\delta}$, and a real
// bound Δ on the absolute value of each element $\delta_i (1 \leq i \leq n)$.
// It returns a lower bound on $g(\delta)$ valid over $\delta \in [-\Delta, +\Delta]^n$.

Set $f = -g$
while f is not constant
  Find greatest order monomial in f: $c\boldsymbol{\delta}^{\boldsymbol{\mu}}$

  // Selecting Cancellation Terms
  Set degree = 0;
  repeat
    degree++;
    Using a greedy search algorithm find a set S where
    each element is a monomial of the form $a_i \boldsymbol{\delta}^{\boldsymbol{\lambda_i}}$
    in f such that (letting $n = |S|$):
    $(\sum_{i=1}^{n} \boldsymbol{\lambda_i} = \boldsymbol{\mu}) \wedge \forall i(|\boldsymbol{\lambda_i}| \leq$ degree)
  until (S is nonempty)

  // Selecting a Subset of Cancellation Terms
  if $(n > m)$
    Form a subset $S' \subset S$ of the m lexicographically
    lowest monomials in S
    Add an extra monomial $c\boldsymbol{\delta}^{\boldsymbol{\mu}} / \prod_{i=1}^{|S'|} \boldsymbol{\delta}^{\boldsymbol{\lambda_i}}$ to S' to
    complete the cover
  else
    Let $S' = S$
  Let $n' = |S'|$

  // Choosing Signs
  Let $d\boldsymbol{\delta}^{\boldsymbol{\beta}}$ be the lexicographically greatest monomial in S'.
  Modify the sign of this monomial by setting:
    $S' = S' \cup \{|d|sgn(c)\Pi_{i=1}^{n'-1} sgn(a_i)\boldsymbol{\delta}^{\boldsymbol{\beta}}\} \setminus \{d\boldsymbol{\delta}^{\boldsymbol{\beta}}\}$.
  for (each monomial $a_i\boldsymbol{\delta}^{\boldsymbol{\lambda_i}}$ in S')
    Create a new polynomial $g_i = (\Delta^{|\boldsymbol{\lambda_i}|} - sgn(a_i)\boldsymbol{\delta}^{\boldsymbol{\lambda_i}})$

  // Choosing the initial multiplier
  Create $h = \prod_{i=1}^{n'} g_i$
  Let Q be the set of terms present both in f and in h.
  For each term $\boldsymbol{\delta}^{\boldsymbol{\rho_i}}$ in Q, let the corresponding
  coefficient in f and h be f_i and h_i respectively.

  Form $q = \min_i(|f_i|/|h_i|)$
  Compute $f = f + qh$
end
Set $\hat{\gamma} = -f$.

Figure 1. Cancellation Algorithm.

individual monomial from the left hand side of equations (11) and (12), using a polynomial of the form (17), and we then note that the sum of polynomials created in this fashion would be a GHR. After cancelling all the monomials, we would be left with a constant from which the bound $\hat{\gamma}_{lower}$ and $\hat{\gamma}_{upper}$ could be derived in a similar fashion to the earlier example.

$$h(\boldsymbol{\delta}) = c\prod_{j=1}^{n}(\Delta^{|\boldsymbol{\mu_j}|} - \boldsymbol{\delta}^{\boldsymbol{\mu_j}})^{\alpha_j}(\Delta^{|\boldsymbol{\mu_j}|} + \boldsymbol{\delta}^{\boldsymbol{\mu_j}})^{\beta_j}. \quad (17)$$

The complexity of this approach is that there are many monomials in $f(\boldsymbol{\delta})$ to cancel, and many ways to cancel any given monomial using polynomials of the form (17). Our heuristic is based upon the idea that while there are several ways to cancel a high order monomial, there is only one way to cancel any first order monomial. For example, consider the earlier example from Table I expanded into the polynomial (16). If we assume x and y and constants, then the only way to cancel the monomial $-xy(2xy-1)\delta_1$ (assuming $xy(2xy-1) > 0$) is the polynomial $xy(2xy-1)(\Delta + \delta_1)$. We therefore cancel monomials starting with the largest order monomial and try to reduce the absolute value of the coefficients of lower order monomials at the same time as cancelling higher order monomials, as this will lead to better bounds. We also note that by cancelling monomials starting with the highest order monomial, termination is guaranteed.

Given this philosophy, the first two stages of our heuristic 'selecting cancellation terms' and 'finding a subset of these terms' determine the values $\boldsymbol{\mu_{i,j}}$ and n of $h(\boldsymbol{\delta})$. The first stage finds non-zero monomials in $f(\boldsymbol{\delta})$ whose product will provide the desired higher order term. Monomials are chosen to prevent the creation of new low order terms, while ensuring that the product is equal to the desired higher order term guarantees that the absolute value of the coefficient for this monomial is reduced. The next stage then limits the number of monomials to a user chosen maximum m to minimize the execution time, because the size the canonical representation of $h(\boldsymbol{\delta})$ grows exponentially in the number of terms; this allows a trade off between execution time and quality of the bound.

After the terms are chosen, the signs α_j, β_j are chosen to reduce as many of the chosen low order monomials as possible, as well as the original highest order term to ensure algorithm termination. Finally, the initial multiplier c is chosen such that when $h(\boldsymbol{\delta})$ is added to $f(\boldsymbol{\delta})$, no coefficient in $f(\boldsymbol{\delta})$ changes sign.

C. Division

Because Theorem 2 only applies to polynomials, the above method cannot be directly applied for algorithms including division. However, we note that any algorithm which consists of the basic operators $\{+, -, *, /\}$ can be converted into a rational function by first applying the multiplicative model of error and then performing simple algebraic multiplication, an example of such manipulation is shown in (18); we can then perform an iterative refinement to calculate a bound.

$$\frac{\delta_1}{\delta_2 + \delta_3/\delta_4} = \frac{\delta_1\delta_4}{\delta_2\delta_4 + \delta_3}. \quad (18)$$

Assuming the rational representing the range of the chosen output variable is of the form n/d, where n and d are polynomials, then we need to show $n/d \geq \hat{\gamma}_{lower}$ inside the compact set of intent. For $d > 0$, this could be re-written as $n - d\hat{\gamma}_{lower} \geq 0$, which is a polynomial inequality. The previous approach may then be repeatedly applied to find a GHR for $n - d\hat{\gamma}_{lower}$; if a representation is found for a value of $\hat{\gamma}_{lower}$, we tighten the value of $\hat{\gamma}_{lower}$, if it fails then one could loosen the value of $\hat{\gamma}_{lower}$. A similar approach could be taken for the upper bound.

In order to minimise the search time, this is performed as a binary search with the initial range given by equation (19), where the ranges of n and d are found using the original method on the numerator and denominator polynomials alone. Using this range also has the added benefit of checking that the denominator does not include the zero value, in which case no such bound exists.

$$[\min(\frac{n_{min}}{d_{max}}, \frac{n_{max}}{d_{min}}, \frac{n_{max}}{d_{max}}, \frac{n_{min}}{d_{min}}) \quad \max(\frac{n_{min}}{d_{max}}, \frac{n_{max}}{d_{min}}, \frac{n_{max}}{d_{max}}, \frac{n_{min}}{d_{min}})]. \tag{19}$$

IV. Testing methodology

In order to characterise the performance of this work, we have conducted two main tests. The first test cases consist of an iteration of the conjugate gradient algorithm applied to a matrix of order two (the operations and inputs are given in Figure 2). Note that our method is able to allow inputs specified as ranges, unlike simulation approaches; this is illustrated by specifying a range for the input vector b.

$$\text{A} = \begin{pmatrix} 4.5 & -6 \\ -6 & 4.5 \end{pmatrix}, \quad \text{x} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \text{b} \in \begin{pmatrix} [5.4 & 6.6] \\ [3.6 & 4.4] \end{pmatrix}$$

$d_1 = b_1$	(1)	$\alpha = \delta_{new}/\alpha_{den}$	(17)
$d_2 = b_2$	(2)	$x_{t1} = \alpha * d_1$	(18)
$r_1 = b_1$	(3)	$x_{t2} = \alpha * d_2$	(19)
$r_2 = b_2$	(4)	$x_1 = x_1 + x_{t1}$	(20)
$\delta_{n\_t1} = r_1 * r_1$	(5)	$x_2 = x_2 + x_{t2}$	(21)
$\delta_{n\_t2} = r_2 * r_2$	(6)	$r_{t1} = \alpha * q_1$	(22)
$\delta_{new} = \delta_{n\_t1} + \delta_{n\_t2}$	(7)	$r_{t2} = \alpha * q_2$	(23)
$q_{t1} = A_{11} * d_1$	(8)	$r_1 = r_1 - r_{t1}$	(24)
$q_{t2} = A_{12} * d_2$	(9)	$r_2 = r_2 - r_{t2}$	(25)
$q_{t3} = A_{21} * d_1$	(10)	$\delta_{old} = \delta_{new}$	(26)
$q_{t4} = A_{22} * d_2$	(11)	$\delta_{n\_t1} = r_1 * r_1$	(27)
$q_1 = q_{t1} + q_{t2}$	(12)	$\delta_{n\_t2} = r_2 * r_2$	(28)
$q_2 = q_{t3} + q_{t4}$	(13)	$\delta_{new} = \delta_{n\_t1} + \delta_{n\_t2}$	(29)
$\alpha_{d\_t1} = d_1 * q_1$	(14)	$\beta = \delta_{new}/\delta_{old}$	(30)
$\alpha_{d\_t2} = d_2 * q_2$	(15)	$d_1 = \beta * d_1$	(31)
$\alpha_{den} = \alpha_{d\_t1} + \alpha_{d\_t2}$	(16)	$d_2 = \beta * d_2$	(32)

Figure 2. Pseudo Code for one iteration of the conjugate gradient Algorithm on a 2x2 Matrix to solve $Ax = b$.

For this test, we attempt to quantify the performance of our algorithm by comparing it against interval arithmetic and two simulation based approaches. The first simulation approach uses the MPFR multiple precision floating library [23] with a random sampling of input data, the second approach performs random sampling over the relevant ranges for all the variables (both input ranges and error variables δ_i) of the polynomial in the expanded form. The reason for performing these two simulations is that the former can be seen as a 'standard' simulation approach, whereas the latter estimates the best possible bound achievable using the multiplicative model of floating point error (equations (4) and (5)). This difference is important to quantify, because the model of error used in this work, and throughout the numerical analysis literature, is a conservative approximation; in practice, each δ_i is a function of the input variables, but this information is lost. As an example, multiplying any value by any power of two will always have zero error, but this effect is not considered by the multiplicative model of floating point error. Overall, with these tests, we wish to show that our approach is significantly better than interval arithmetic, and also if we can show the difference between our approach and the simulations is small, we know that our work approaches the ideal bounds and the multiplicative model of floating point error is suitable.

The second test is a direct comparison against the various test cases in [10], with the details of the benchmarks contained in that paper. As mentioned in Section II, the test cases from [10] are actually focused upon on the range analysis problem as opposed to the effects of finite precision, but these are included to demonstrate that our approach is also applicable to this problem and performs well not only in comparison to SMT, but also to Affine Arithmetic to which SMT was compared in [10].

V. Results

A. The effects of finite precision on range

Figure 3 shows the upper bound on the range of the variable d_1, after an iteration of the conjugate gradient algorithm (operation 31 from Figure 2), and highlights the effect precision has on the conjugate gradient algorithm and the quality with which our approach can characterise this effect. On this graph, the vertical dotted lines illustrate the values of precision for realisable word-lengths, *i.e.* the difference in word-length between any two adjacent dotted lines is one bit. It is clear to see that there is a significant difference between the ranges calculated by our approach in comparison to IA, a difference that can translate to a significant improvement in hardware. As an example, assume it is a requirement for the upper bound to be less than 10^4; interval arithmetic would state a precision of approximately 2×10^{-4} or 12 bits would be required, whereas our approach shows that only a precision of approximately 2×10^{-2} or 6 bits would actually be required. In order to view this in another fashion, Table II shows the number of slices required, the latency, and the maximum frequency achievable when using single or double precision units, or by performing optimisation for a given bound on range using either IA or our approach. In this test, these figures are post place and route results where the floating point components are generated using Xilinx Coregen. From this table, it should be clear that our approach can achieve a reduction in slices of over 40% in comparison to optimising the design using the more traditional method of IA, and significantly larger savings in comparison to using full IEEE single or double precision arithmetic. It also demonstrates the design would run at a faster frequency and an iteration would complete in fewer cycles by performing the optimisation.

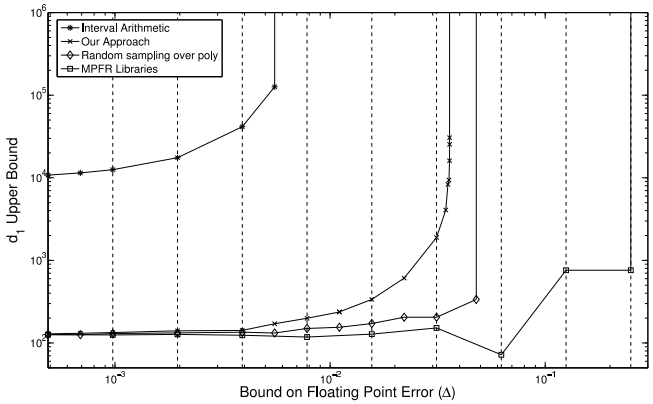

Figure 3. 'New Direction' d_1 (operation 31 from Figure 2) Upper Bound.

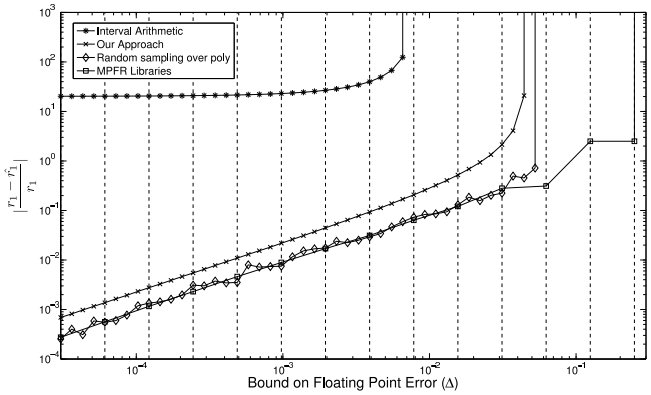

Figure 4. CG 'Residual' relative error bound. r_1 (operation 24 from Figure 2) is the nominal 'residual', whilst $\hat{r}_1$ is the residual taking into account floating point error

Table II
RESOURCE USAGE, MAX FREQUENCY AND LATENCY OF CONJUGATE GRADIENT IMPLEMENTATIONS

Method	Precision (# bits)	Slices	Frequency (MHz)	Latency (cycles)
Guaranteed Using Our Approach	6	4140	369	82
Guaranteed Using IA	12	7111	352	99
IEEE Single Precision	23	17209	286	144
IEEE Double Precision	52	58228	256	207

One should also note that a designer using only interval arithmetic would have to assume that a precision of 6 bits would be impossible, because the graph shows any precision less than approximately 7×10^{-3}, *i.e.* 7 bits or fewer, would give an unbounded result due to the divisions in the code. Finally, it is interesting that our full approach becomes very close to the results from simulation, especially for small precisions, implying that the bound is tight.

It should, however, be mentioned that looking at the variable d_1 after one iteration was as far through the conjugate gradient algorithm that our approach was tested. The reason for this is that the simplified polynomial for this variable consisted of approximately 2 million monomials, each of which would need to be cancelled by an appropriate polynomial such that a GHR could be formed, as described in Section III-A. This is time consuming, especially given that this value is a rational function and therefore requires the iterative refinement mentioned in Section III-C.

B. The effects of finite precision on relative error

Figure 4 demonstrates the bound on relative error. This figure clearly demonstrates that the relative error decreases with precision, and also that our algorithm tracks this relationship well, unlike a naïve application of interval arithmetic. The reason that interval arithmetic does not show this relationship is that it can only find wide bounds as a result of the input data ranges and the fact that the correlation between r_1 and $\hat{r}_1$ is lost when performing the subtraction $r_1 - \hat{r}_1$, and this overshadows any finite precision effects. As in the previous section, this could be translated to a significant saving in hardware.

C. Range analysis vs other works

Figure 5 shows the performance of our work normalised against the ranges given in [10], which given sufficient run time, should be optimal. Interestingly, our approach in some cases gives slightly superior results. This is likely to be a result of the fact the SMT is a refinement process which is potentially time consuming and hence the refinement stops once at a given level of accuracy. On the other hand, it is important to note that in some cases SMT does outperform our approach - to the extent that in some cases our work gives undefined results, whereas the SMT solver returned bounds. Similarly, our approach either outperforms or is equivalent to affine arithmetic in all but two of these benchmarks. The reason our approach does not always find the ideal bound is a limitation of our heuristic at finding the best Handelman representation, and whilst achieving better bounds is possible, it would require a more sophisticated search for Handelman representations which will be time consuming. This shall be reserved for future research. However, it should be mentioned that our solver calculates these values within hundreds of milliseconds, as shown in Table III, whereas in [10], the results for each value are reported as around the order of 100 seconds on a comparable machine. Furthermore, one should also note that our approach could be used as an input to the SMT solver, which currently uses initial estimates based on interval or affine arithmetic, to decrease the time to find a good solution.

VI. CONCLUSION

This paper has demonstrated a heuristic, based upon a result from real algebra, that can be used to find analytical bounds for any value within an algorithm. It has shown that this method can achieve superior results to the existing methods, and has demonstrated its potential use to design hardware with minimised precision.

Whilst this research has shown a high degree of promise, there are several limitations that will need further research.

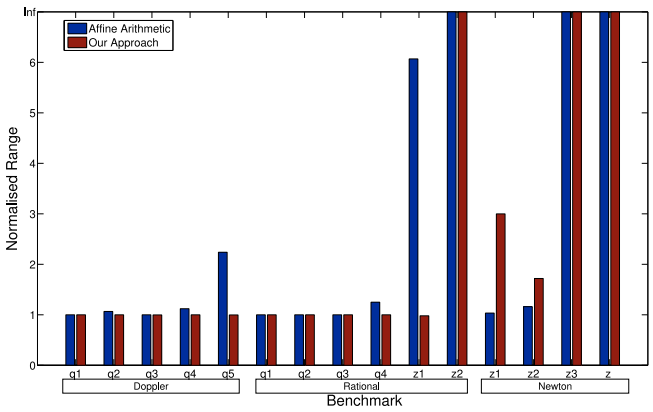

Figure 5. Comparison vs SMT. Range widths found for the benchmarks using our approach and AA are normalised with respect to the values stated in [10].

Table III
EXECUTION TIMES OF OUR ALGORITHM. N/A INDICATES THAT NO BOUND CAN BE FOUND.

Benchmark	Time (ms)
Doppler q1	118
Doppler q2	148
Doppler q3	116
Doppler q4	176
Doppler q5	3000
Rational q1	121
Rational q2	124
Rational q3	114
Rational q4	189
Rational z1	4000
Rational z2	N/A
Newton z1	171
Newton z2	142
Newton z3	N/A
Newton z	N/A

These include improving the scalability in that this approach is still time consuming for large algorithms, improving the heuristic to obtain better Handelman representations, as alluded to in Section V-C, as well as extending the method to handle more complex functions such as square roots and exponentials. With the addition of this research, it is hoped that this method will be able to calculate robust and useful bounds for many algorithms, which in turn should be able to significantly aid hardware design.

REFERENCES

[1] G. A. Constantinides, P. Y. K. Cheung, and W. Luk, *Synthesis And Optimization Of DSP Algorithms*. Norwell, MA, USA: Kluwer Academic Publishers, 2004.

[2] A. Buttari, J. Dongarra, J. Kurzak, P. Luszczek, and S. Tomov, "Using mixed precision for sparse matrix computations to enhance the performance while achieving 64-bit accuracy," *ACM Trans. Math. Softw.*, vol. 34, no. 4, pp. 1–22, 2008.

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

[4] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[5] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, and A. D. George, "RAT: a methodology for predicting performance in application design migration to FPGAs," in *Proc. Workshop on High-performance reconfigurable computing technology and applications*. New York, NY, USA: ACM, 2007, pp. 1–10.

[6] G. Constantinides, P. Cheung, and W. Luk, "Wordlength optimization for linear digital signal processing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 22, no. 10, pp. 1432–1442, Oct. 2003.

[7] D. Boland and G. Constantinides, "An FPGA-based implementation of the MINRES algorithm," in *Proc. Int. Conf. Field Programmable Logic and Applications*, Sept. 2008, pp. 379–384.

[8] A. Lopes and G. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation," *Reconfigurable Computing: Architectures, Tools and Applications*, pp. 75–86, 2008.

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

[10] A. B. Kinsman and N. Nicolici., "Finite precision bit-width allocation using SAT-modulo theory," in *Proc. Int. Conf. DATE*, 2009.

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

[12] D.-U. Lee, A. Gaffar, R. Cheung, O. Mencer, W. Luk, and G. Constantinides, "Accuracy-guaranteed bit-width optimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 10, pp. 1990–2000, Oct. 2006.

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

[14] H. Choi and W. Burleson, "Search-based wordlength optimization for VLSI/DSP synthesis," in *Proc. Workshop on VLSI Signal Processing, VII*, 1994, pp. 198–207.

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

[16] K.-I. Kum and W. Sung, "Combined word-length optimization and high-level synthesis of digital signal processing systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 8, pp. 921–930, Aug 2001.

[17] J. de Figueiredo, Luiz Henrique Stolfi, "Affine arithmetic: concepts and applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, 2004.

[18] R. E. Moore and F. Bierbaum, *Methods and Applications of Interval Analysis*. Soc for Industrial & Applied Math, 1979.

[19] B. Einarsson, *Handbook on Accuracy and Reliability in Scientific Computation*. Soc for Industrial & Applied Math, 2005, ch. 10, p. 195240.

[20] L. H. D. Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, 2003.

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

[22] D. Handelman, "Representing polynomials by positive linear functions on compact convex polyhedra," *Pac. J. Math*, vol. 132, no. 1, pp. 35–62, 1988.

[23] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, and P. Zimmermann, "MPFR: A multiple-precision binary floating-point library with correct rounding," *ACM Trans. Math. Softw.*, vol. 33, no. 2, June 2007.