Unifying Bit-width Optimisation for Fixed-point and Floating-point Designs

Altaf Abdul Gaffar[1], Oskar Mencer[1], Wayne Luk[1] and Peter Y.K. Cheung[2]

[1] Department of Computing, Imperial College, London SW7 2BZ, UK.

[2] Department of EEE, Imperial College, London SW7 2BT, UK.

Abstract

This paper presents a method that offers a uniform treatment for bit-width optimisation of both fixed-point and floating-point designs. Our work utilises automatic differentiation to compute the sensitivities of outputs to the bit-width of the various operands in the design. This sensitivity analysis enables us to explore and compare fixed-point and floating-point implementation for a particular design. As a result we can automate the selection of the optimal number representation for each variable in a design to optimize area and performance. We implement our method in the BitSize tool targeting reconfigurable architectures, which takes user-defined constraints to direct the optimisation procedure. We illustrate our approach using applications such as ray-tracing and function approximation.

1 Introduction

One of the main challenges facing a hardware designer is to determine the appropriate bit-widths for the components in a design that meet system requirements. The increase in design complexity, enabled by Moore's Law, renders hand optimisation of bit-widths unattractive except for small designs. An automated method which can perform bit-width optimisation is vital to accelerate the hardware design cycle.

This paper describes a method capable of deducing operator bit-widths automatically for a given software description of an algorithm. The method can cover both floating-point and fixed-point hardware implementations. We have implemented this method in a tool called BitSize.

The choice of fixed-point or floating-point representations is largely driven by the dynamic range required by an application. Our tool provides a unique facility for system designers to explore the trade-offs between various parameters, such as accuracy, dynamic range, area and speed.

While our approach is particularly relevant to reconfigurable designs which can be produced directly by application developers, it can also be used in optimising application-specific integrated circuits.

The key elements of our work include:

- a framework that offers a unified treatment of bit-width analysis for different number representations;
- the use of this framework in determining bit-widths for fixed-point and floating-point designs;
- the implementation of this framework in the BitSize tool that targets reconfigurable devices such as FPGAs (Field-Programmable Gate Arrays).
- the evaluation of our approach using four case studies: ray-tracing, function approximation, Finite-Impulse Response (FIR) filtering and Discrete Cosine Transform (DCT).

Note that while our discussion in this paper is focused on area reduction, recent work [6] has demonstrated that bit-width optimisation can also result in significant reduction in power consumption.

The rest of the paper is organised as follows. Section 2 presents an overview and background of our proposed technique. Section 3 discusses the trade-offs between floating-point and fixed-point arithmetic in hardware. Section 4 explains the mathematical reasoning behind the use of automatic differentiation for bit-width analysis. Section 5 presents the method employed to calculate the optimal bit-widths for both floating-point and fixed-point designs, along with a discussion of the BitSize algorithm which we use to select between the two number representations. Section 6 describes the implementation of our tool, BitSize. Section 7 presents four case studies: a ray-tracer, a function approximator, a discrete cosine transform circuit, and an FIR filter. Section 8 contains concluding remarks.

2 Overview and background

This section contains three parts. Section 2.1 provides an overview of our approach, Section 2.2 considers the trade-offs between fixed-point and floating-point designs and Section 2.3 presents the theoretical background for the use of Automatic Differentiation for bit-width analysis.

2.1 Overview

Figure 1 illustrates the design flow of our method. The main part of this method is performed by our tool BitSize, the input to which is either a C/C++ design description or a Xilinx System Generator [21] design description. The output of the tool is an annotated data flow graph which can then be used to produce either fixed-point or floating-point implementations.

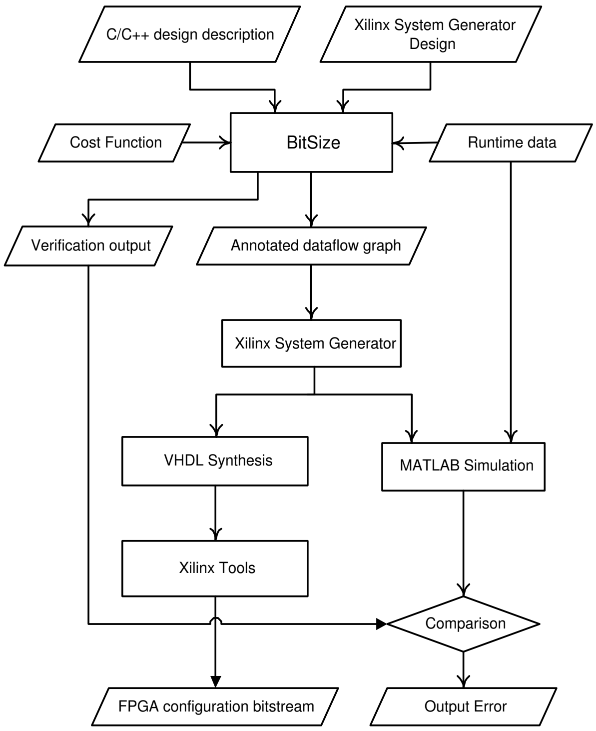

Figure 1. The design flow of our method.

BitSize also supports a verification path which is used to check that the proposed design meets user specified error requirements. Fixed-point designs can be implemented using the Xilinx System Generator hardware design suite, and the simulation facilities provided by Matlab can be used for design verification. Floating-point designs can be implemented with the aid of a parameterisable floating-point hardware library. We verify floating-point designs in software using a parameterisable floating-point simulation library. Once the designs pass the verification stage, the hardware description is synthesised, placed and routed to produce an FPGA configuration bit-stream.

The following describes our approach in more detail, and compares it to other methods. We split the problem of minimising the design bit-widths into two parts: range analysis and precision analysis. Range analysis has received much attention within recent integer bit-width analysis work [4], [17], [18], [20]. Precision analysis, on the other hand, involves analysing the "sensitivity" of the output from a computation to slight changes in the inputs; more specifically, the sensitivity of an output to the computational precision within an arithmetic unit. So far research into precision analysis has mainly focused on fixed-point implementations [5], [6], [7], [8], [12], [19].

The most straight-forward bit-width optimisation method is to try out various bit-widths and observe the output for each design [1]. This technique, however, involves an enormous search space. Another method for the calculation of bit-widths is the use of automatic differentiation [2]. In [2] only floating-point designs are considered, while in [10] only numerical software is considered. This paper proposes a technique which can target both floating-point and fixed-point hardware implementations.

In addition to sensitivity, we also consider the dynamic range of the operations, which is used to determine the integer bit-widths for fixed-point operations and the exponent bit-widths for floating-point operations.

2.2 Fixed-point versus floating-point designs

Hardware arithmetic has traditionally focused on either integer or fixed-point arithmetic representations. Due to the significant increase in resources in the latest FPGAs, it is now feasible to support more complex arithmetic formats such as floating-point [3], [11] and logarithmic representations [13] in hardware. It is therefore attractive to have a bit-width optimisation tool that can support various arithmetic formats. This paper describes the theory and practice of such a tool that can cover both fixed-point and floating-point designs.

Floating-point implementations are efficient when a large dynamic range is required, which would otherwise involve a fixed-point representation with large bit-widths. Many applications currently being developed for FPGAs require the support for large dynamic ranges.

The software floating-point standard most commonly implemented today is the IEEE 754 floating-point standard. This standard specifies several floating-point formats, the most common being the single and double precision formats. The former allocates 23 bits for the mantissa and 8 bits for the exponent, whilst the latter allocates 53 bits for the mantissa and 10 bits for the exponent.

Fixed-point arithmetic is the more straight-forward of the two number representations. In fixed-point representation, an implicit binary point is used to separate the integer part and the fractional part within a single data word. Fixed-point number representation facilitates implementation of most of the calculations as integer arithmetic, as little pre- or post-normalisation is required.

The pre- and post-normalisation steps used in floating-point arithmetic require the use of priority encoders and variable shifters. These components are expensive in terms of area usage and power consumption, and tend to have large combinational delays. Hence when we considering identical range and precision, floating-point addition is always more costly than fixed-point addition in terms of speed, area and power consumption. The dynamic range of floating-point multipliers allows us to keep the area close to a constant when increasing the dynamic range of the data. This is not the case with fixed-point multipliers where an increase in dynamic range range requires a large increase in area.

We shall illustrate in Section 7 how dynamic range can be used to select the number representation for a given application.

Our method exploits the opportunity to use customised arithmetic formats, where we can have arbitrary integer and fractional widths for fixed-point designs and arbitrary mantissa and exponent widths for floating-point designs. Our tool performs analysis for both fixed-point and floating-point designs, and then recommends the best format to use. It is based on a simulation technique that requires a sample data set as an input to perform the analysis.

2.3 Automatic differentiation framework

Automatic differentiation [10] is a method developed by the applied mathematics community for the differentiation of algorithms. The main advantage that automatic differentiation provides is the ability to calculate the differentials as a side effect of the execution of the user algorithm, with few changes to the algorithm itself.

Automatic differentiation offers us a faster alternative than simulation-only methods for bit-width analysis. Only a single iteration is required to calculate the maximum error tolerance at a node in the data flow graph for a given output error specification, thereby significantly reducing the design search space.

This section introduces an automatic differentiation framework that provides a unified treatment of bit-width analysis applicable to different number representations.

In Figure 2, we show a simple example of the operation of automatic differentiation on the data flow graph of a computational schema for the function $f = x_1 \times x_2 + x_3$. When the data flow graph is being evaluated, at each operator node automatic differentiation calculates the gradients with respect to the inputs to that node and then annotates the respective edges. From this illustration we derive definition 1 for the sensitivity relationship between the output and the input.

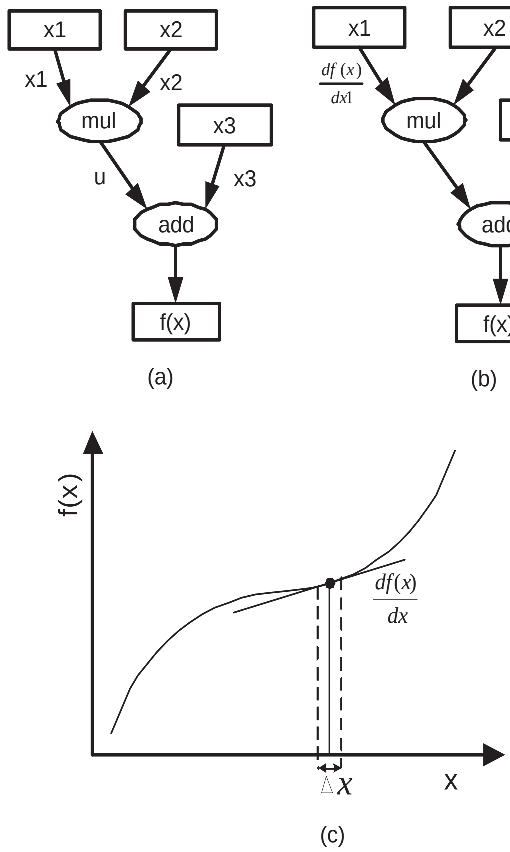

Figure 2. Data flow graph (a) shows the computation of the function $f = (x_1 \times x_2) + x_3$, while data flow graph (b) shows the same function where the edges are annotated with the gradients. Graph (c) highlights the relationship between the gradient and the input error in x, Δx.

Definition 1 (Sensitivity) *The sensitivity (s) of an output (y) is a function of the input (x), such that a change in the input causes a change in the output :* $\Delta y = s(\Delta x) \simeq f'(x)\Delta x$

The derivation of the relationship mentioned in definition 1, between an output and a single input is demonstrated by the set of equations 1 - 4. It is also possible to extend definition 1 to the case where there is more than one input to the function node.

$$y = f(x) \quad (1)$$

$$y' = f'(x) = \frac{dy}{dx} \quad (2)$$

$$dy = f'(x)dx \quad (3)$$

$$\Delta y \simeq f'(x)\Delta x \quad (4)$$

Next we extend our definition of sensitivity, by considering a node with n inputs $U_0 \cdots U_n$, and output y. The inputs are related to the output by the differentiable function f_j as shown in equation 5.

$$Y = f_j(U_0, U_1, \cdots, U_n) \tag{5}$$

Let ΔU_i be the error introduced when U_i is represented in finite precision. ΔU_i is also known as the absolute error and can be expressed in equation 6.

$$\Delta U_i \leq |\bar{U}_i - U_i| \tag{6}$$

where $\bar{U}_i$ is the value of U_i in finite precision.

Since the use of infinite precision arithmetic for our analysis is cumbersome, we represent U_i in IEEE double precision floating-point format. We follow this method on the basis that most hardware designs are derived from software designs using IEEE floating-point format. However, our approach can easily be adapted to other approximations of infinite precision arithmetic, such as the exact computation format [22].

Let ΔY be the effect on Y in response to the changes in U_i, it can be expressed using the Taylorian approximation shown in equation (7):

$$\Delta Y \geq \Delta U_1 \frac{dY}{dU_1} + \cdots + \Delta U_n \frac{dY}{dU_n} \tag{7}$$

where dY/dU_i is the sensitivity or gradient of Y, to changes in U_i. The higher order terms in the Taylorian approximation are ignored, under the assumption that the contribution of their values is negligible. Automatic differentiation provides us with the values of the gradients.

This approximation holds when $\Delta U_i << U_i$. In a typical application of our method the user specifies the maximum tolerable error at the output either as an absolute error ΔY or as a relative error ΨY.

We use a backward propagation method to calculate the values of ΔU_i while ensuring that the inequality in equation (7) is satisfied.

In equation (8) we express ΔU_i in terms of the bit-width of the node, where E_{flt} and E_{fix} are the error functions which relate the mantissa and fractional bit-widths to the computational error at the node.

$$\Delta U_i = \begin{cases} Err_{flt}(man\_bw) & \text{if Type = Float} \\ Err_{fix}(frac\_bw) & \text{if Type = Fixed} \end{cases} \tag{8}$$

where Type refers to the arithmetic format selected for the design under analysis, while man_bw represents the mantissa bit-width for floating-point and $frac\_bw$ represents the fractional bit-width for fixed-point. The precision analysis problem is now simplified to finding the values of ΔU_i while satisfying the condition of equation (7). This is in contrast to the large number of iterations that we would require if a naive simulation based method is employed.

3 From error to bit-width calculation and design selection

This section shows how the framework in the preceding section can be used in calculating bit-widths for two number representations. Our analysis treats the problems of precision and range analysis separately. In the case of floating-point the precision depends on the mantissa bit-width, while the range depends on the exponent bit-width. In the case of fixed-point the range depends on the integer bit-width, while the precision depends on the fractional bit-width.

Targeting floating-point designs

Let U_i represent a floating-point number $(-1)^S \cdot M \cdot 2^E$, where S is the sign bit, M is the mantissa with a bit-width of m bits, and E is the exponent with a bit-width of e bits.

S	$a_0\ a_1\ a_2 \cdots\ a_{m-1}$	$b_{e-1} \cdots b_2\ b_1\ b_0$

The value of the mantissa M is expressed as:

$$M = \sum_{i=0}^{m-1} a_i 2^{-i} \tag{9}$$

where $a_i \in \{0, 1\}$.

From equations (8) and (9), it is possible to relate the bit-width m of the mantissa of the node to the error when representing the mantissa by a finite bit-width Err_{flt}, as follows:

$$Err_{flt}(m) = \begin{cases} 2^{-m} \times 2^E & \text{if round-to-nearest} \\ 2^{-(m-1)} \times 2^E & \text{if truncation} \end{cases} \tag{10}$$

where E is the value of the exponent at the node. In addition to the dependence on the bit-width of the mantissa m, Err_{flt} also depends on the rounding mode used when converting the floating-point value to finite precision value. A rounding mode, such as round-to-nearest, while giving better error bounds than truncation would require additional hardware to implement. Truncation would require one extra bit to provide the same error bound as round-to-nearest.

We select truncation for our hardware implementations since the area cost of having an extra bit in the bit-width is less than the cost of implementing round-to-nearest. After calculating the ΔU_i values, from equation (10) we derive equation (11) for calculating the mantissa bit-width m:

$$m \geq E_{U_i} - \lceil \log_2(|\Delta U_i|) \rceil + 1 \tag{11}$$

where E_{U_i} is the value of the exponent of U_i, which can be found by $E_{U_i} = \lceil log_2(|U_i|) \rceil$.

The dynamic range of the operation is given by $|\max(U_i) - \min(U_i)|$. The exponent bit-width of U_i, e can be calculated as follows. The exponent bit-width is related to the dynamic range of the number:

$$e \geq \lceil log_2(|\max(E_{U_i}) - \min(E_{U_i})|) \rceil \qquad (12)$$

Targeting fixed-point designs

Now we consider the case when U_i is represented as a fixed-point number, with an integer part I which is k bits in length, and a fraction part F which is l bits in length.

$p_{k-1} \cdots p_2\, p_1\, p_0$	$q_0\, q_1\, q_2 \cdots q_{l-1}$

The integer bit-width, which represents the dynamic range of the number, is calculated according to equation (13):

$$k \geq \lceil \log_2(|\max(U_i) - \min(U_i)|) \rceil \qquad (13)$$

As in the case of floating-point, we introduce an error function to reflect representing the fractional part of the fixed-point value by a finite bit-width. This error function represented by Err_{fix} in equation (8), can be related to the fractional bit-width l as follows:

$$Err_{fix}(l) = \begin{cases} 2^{-l} & \text{if round-to-nearest} \\ 2^{-(l-1)} & \text{if truncation} \end{cases} \qquad (14)$$

Once again we select truncation as opposed to rounding for the hardware implementation, based on the same justification we presented in the floating-point case.

From equation (14) and the value of ΔU_i calculated as before we derive equation (15) to express the bit-width:

$$l \geq \lceil \log_2(|\Delta U_i|) \rceil + 1 \qquad (15)$$

When a number is represented in fixed-point format, the bit-width of the integer part should be large enough to cover the dynamic range. As an example, a dynamic range of 10^6 requires 20 bits in the integer part, while the same dynamic range requires only 5 bits in the exponent of a floating-point number.

Design selection

Next, we present our BitSize algorithm for reducing the bit-widths while satisfying user-specified design constraints, the operation of the algorithm is as shown in Figure 3.

Our bit-width optimisation technique is guided by the user-specified design constraints. These design constraints can include, in addition to the maximum permitted output error specification, the required dynamic range, the maximum area usage and the maximum combinational delay of the design.

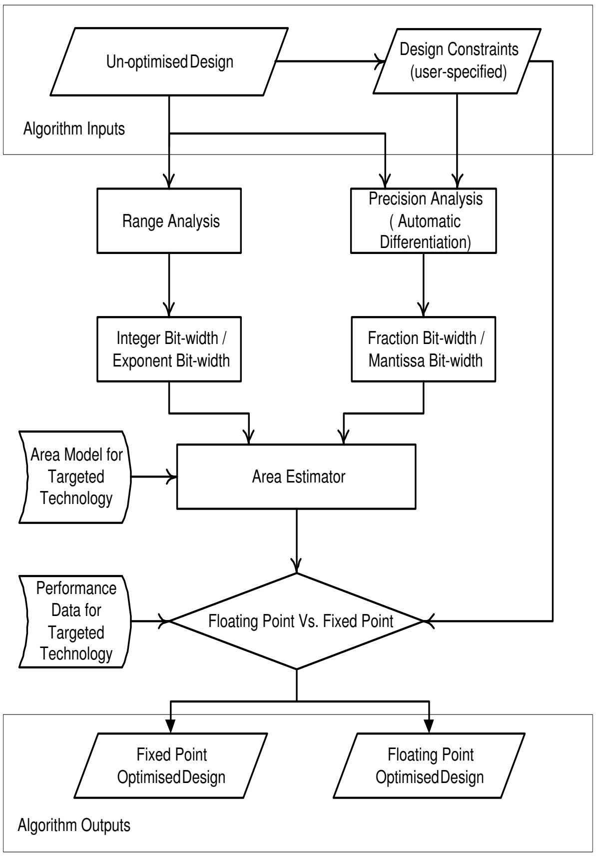

Figure 3. The BitSize algorithm flow.

The user-specified design constraints and the un-optimised design form the inputs to the BitSize algorithm. The algorithm gives priority to the maximum permitted output error in the design constraints over the other design constraints such as area, speed or power consumption. The main analysis phase of the algorithm consists of two parallel sub-analysis phases: (1) range analysis and (2) precision analysis. The range analysis observes the values passing through the nodes of the data flow graph of the un-optimised design and determines the dynamic range at each. This would be translated as the integer bit-width in fixed-point arithmetic from equation (13), or as the exponent bit-width in floating-point arithmetic from equation (12).

The precision analysis sub-phase uses automatic differentiation to determine the maximum error tolerance at each node, for a user-specified output error specification. Again depending on whether fixed-point or floating-point is selected for the node under analysis, this is translated to become the fraction bit-width from equation (15) or as the mantissa bit-width from equation (11).

The area estimator calculates the area usage of the bit-width optimised design, based on an area model related to the target technology. This area model describes the area usage in terms of the bit-width of the operator for all the operators available in our implementation libraries. The area model does not consider the routing or input/output overheads in the implemented design.The area estimator obtains the total area usage of the design from of equations (16) and (17), where the area is modelled as a function of the operator nodes in the design, their operator types, arithmetic types and bit-width.

$$TotalArea = \sum_{i}^{N} A_i \tag{16}$$

where A_i the area of an individual node.

$$A_i = \begin{cases} G_{flt}(W_{exp}, W_{man}, OP) & \text{if Type = Float} \\ \\ G_{fix}(W_{int}, W_{frac}, OP) & \text{if Type = Fixed} \end{cases} \tag{17}$$

where $OP \in \{+, -, \times, /\}$ and Type is the arithmetic type selected for the node. Equation (17) expresses the area usage of an single node i, A_i. This is expressed by G_{flt} when floating-point is selected for implementation, where W_{exp} and W_{man} are the floating-point exponent and mantissa bit-widths respectively. When fixed-point is selected G_{fix} is used to calculate the area, where W_{int} and W_{frac} represent the fixed-point integer and fractional bit-widths respectively.

The resulting values from the area estimator form one of the inputs to the decision making phase of the analysis which determines the most suitable data type to employ for the implementation of the design. The other inputs to the this decision phase include a performance data model for the target technology along with user-specified design constraints. The performance model contains maximum possible performance data vectors for all the operator blocks in the fixed-point and floating-point libraries and is specific to the implementation target technology. These performance vectors are obtained empirically and only provide a rough guide for the decision making process. It is possible to include other performance metrics such as power consumption in addition to speed.

The user-specified constraints can be used to further guide or in some cases override the decision making process in favor of one particular data format. For example the user might request a large dynamic range to be used than that found in the analysis phase, resulting in a preference for one data format over the other.

In cases where the algorithm fails to find a design which satisfies all the given user specifications, user intervention is required to alter the design constraints. In this mode the algorithm is iterated until a design which satisfies the user requirements is found.

4 The BitSize tool

BitSize is implemented as a C++ object library, and currently supports two main front ends: (a) an operator overloaded C++ interface and (b) a Xilinx System Generator interface. The advantages of method (a) include the ability to analyse stock C/C++ code with few changes to it. The advantages of method (b) include the ability to describe our designs inside System Generator, where we can use its design verification and synthesis features.

A standard C++ compiler, such as Microsoft Visual C Compiler (MSVC) or the GNU Compiler Collection (GCC), is employed to compile transformed source code along with the BitSize library. The precision analysis stage

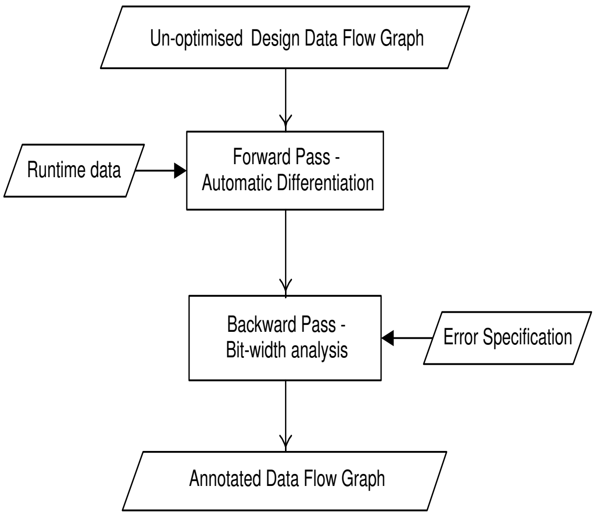

Figure 4. The Precision Analysis stages of the BitSize tool.

of BitSize takes place when executing the compiled code. The execution consists of two passes: forward analysis and backward analysis. The forward pass involves automatic differentiation of the nodes in the data flow graph. The user-supplied sample data set is used in this pass. Although traditional automatic differentiation tools [9] could have been used for this pass, most of these tools are found to be either too complicated or too slow for our purpose.

In the backward pass, the user-provided error specification is used in conjunction with the sensitivity values calculated in the forward pass to perform bit-width calculation as described in Section 5. In this pass we calculate the maximum error tolerance possible at each operator node in the data flow graph of the design. The annotated data flow graph output of the precision analysis stage of BitSize is then used as one of the inputs to area estimator, which together with the results of the range analysis and other user-specified constraints, selects between the floating-point and fixed-point implementations.

Once the analysis is completed, our tool provides several back-ends which enable us to target different hardware implementation systems. By converting the data flow graph in to a Matlab script file, we can realise and evaluate both fixed-point and floating-point designs via the Xilinx System Generator design suite.

Alternatively it is possible to convert the designs into either a VHDL, a Handel-C design description or an ASC [15], [16] design description. Handel-C design descriptions require the Celoxica DK2 system, while VHDL designs require the Synplicity VHDL synthesis tool and the ASC design descriptions require a standard C/C++ compiler such as GCC to synthesise the designs. All the hardware designs presented in Section 7 of this paper are targeted towards Xilinx FPGAs, and hence we use Xilinx software for the placement and routing stage of the synthesis process.

5 Case studies

We illustrate the application of our BitSize technique by four case studies: ray-tracing, function approximation, Finite-Impulse Response (FIR) filtering and a Discrete Cosine Transform (DCT).

Ray-Tracing

The first case study to illustrate our method is ray-tracing. Ray-tracing is used in 3D graphics rendering. For our case study we explore the bit-width minimisation of the determinant of the equation in ray-tracing to find the intersection points between a ray and a sphere.

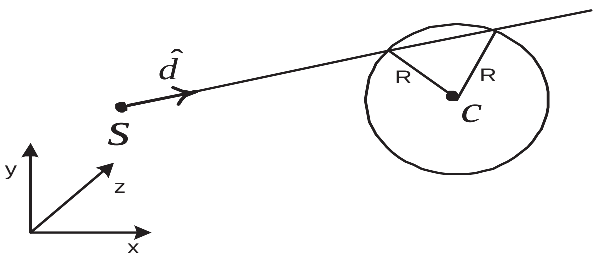

Figure 5. The intersection between a ray with direction vector $\hat{d}$ from point $\vec{s}$, and a sphere with center at point $\vec{c}$ and radius R.

From Figure 5 a point p on the ray starting at $\vec{s}$ and direction $\hat{d}$ can be expressed as: $\vec{p} = \vec{s} + \mu\hat{d}$. At the points of intersection between the ray and the sphere $|\vec{c} - \vec{s} + \mu\hat{d}| = R$. Solving for μ yields a quadratic, the determinant D of which is:

$$D = b^2 - (\vec{v} \cdot \vec{v} - R^2) \tag{18}$$

where $\vec{v} = \vec{c} - \vec{s}$ and $b = \vec{v} \cdot \hat{d}$.

We implement equation (18) in hardware using the Xilinx System Generator for design entry. Next we use this design description as input to our BitSize analysis tool, along with a specification for the maximum output error. The bit-width annotated data flow graph produced as output by BitSize is then used to modify the original design specification. The modified design is then synthesied with System Generator. For the purposes of illustrating our method, we implement the design in both fixed-point and floating-point arithmetic.

Output Error (%)	0.0	0.1	0.2	0.5	0.75
Flip Flops	3885	3510	3388	3230	3090
LUTs	6726	5957	5734	5361	5053
Embedded Mults	7	7	7	7	7

Table 1. FPGA resource usage versus relative error for ray-tracer, implemented in floating-point.

Output Error (%)	0.0	0.1	0.2	0.5	0.75
Flip Flops	2000	1709	1620	1407	1243
LUTs	1932	1630	1532	1328	1173
Embedded Mults	28	28	28	25	22

Table 2. FPGA resource usage versus relative error for ray-tracer implemented in fixed-point with the use of embedded multipliers.

Output Error (%)	Ref	0.1	0.2	0.5	0.75
Flip Flops	4455	4674	4352	4225	3982
LUTs	5506	4559	4244	4108	3819

Table 3. FPGA resource usage versus relative error for ray-tracer, implemented in fixed-point without the use of embedded multipliers.

The FPGA resource utilisations for the floating-point and fixed-point implementations of the ray-trace are presented in Tables 1, 2, 3 and Figure 6. The dynamic range of these designs is 10^3. All the designs are targeted for the Xilinx Virtex2 XC2V2000 chip. From these results we observe that:

- The fixed-point implementations on average use 50% fewer flip-flops and 75% fewer LUTs than floating-point implementation for a similar output error specification. A reason for the lower LUT usage in the

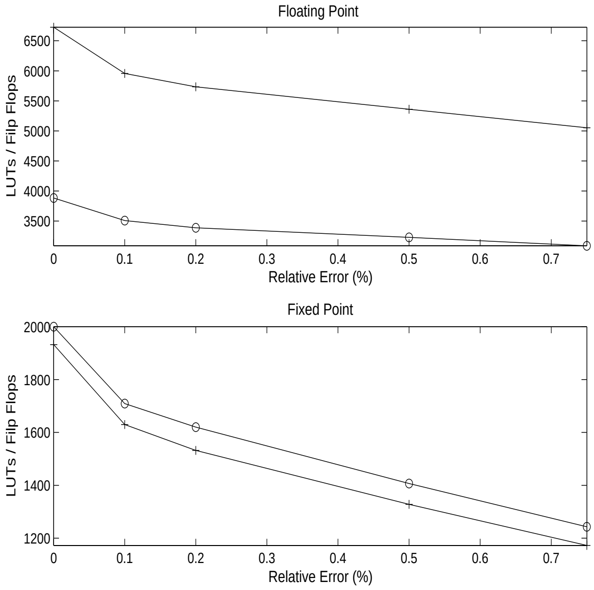

Figure 6. The variation of FPGA resource utilisation with output relative error specification for the ray-tracing example. The variation in Look Up Table (LUT) usage is shown by (+) and the variation in Flip Flop usage by (o).

fixed-point implementations, is that they use 3 times more embedded multipliers than in the floating-point implementations.

- From the graphs in Figure 7, plotting variation of the FPGA resource utilisation and the dynamic range, we can see that the cross-over point for this particular design where floating-point design requires fewer flip-flops then the fixed-point design, lies at a dynamic range of 10^6.

- A 0.1% output error specification gives us a 10% reduction in LUT usage and 9% reduction in flip flop usage when we consider the fixed-point implementation. The rate of change in area decreases to 3% and 2.5% for LUTs and flip-flops when we increase the error specification from 0.2% to 0.5%.

- A similar trend is also noted for the floating-point implementations where the greatest reduction in area occurs when the error specification is 0.1%.

- After placement and routing, we find that the fixed-point designs can operate at 100MHz while the floating-point designs can operate at 80MHz. The fixed-point designs use arithmetic IP cores provided by Xilinx and are therefore optimised towards the target FPGA, whereas the floating-point libraries we use are FPGA target independent and are less efficient.

- Table 3 presents the result of implementing the ray-tracer in fixed-point, but without the use of the embedded multipliers on the FPGA. The fixed-point designs use 20% more flip-flops and use 25% fewer LUTs compared to floating-point designs.

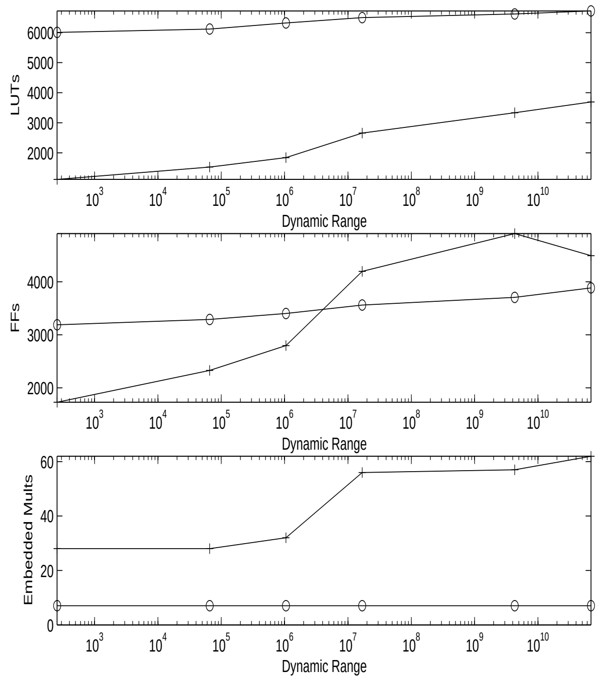

Figure 7. The variation of the FPGA resource utilisation with increasing dynamic range for floating-point (o) and fixed-point (+) implementations of the ray-tracer. All implementations have a relative output error specification of 0.1%.

Function Approximation

The next case study involves the determination of the bit-widths of the operations used in hardware function approximation, expressed in equation (19):

$$f(x) = C_2 \cdot x^2 + C_1 \cdot x + C_0 \qquad (19)$$

where the values of the constants C_0, C_1 and C_2 are selected according to the function being approximated. By changing these values appropriately it is possible to approximate a wide range of elementary functions [14]. For our example we try the linear approximations of the functions $f(x) = \sqrt{-ln(x)}$ and $f(x) = xlog(x)$.

Figure 8 illustrates the variation in area for the two function approximations. The variation is shown for both the fixed-point and floating-point implementations. For the approximation of $ln(x)$, a 20% area reduction is possible in

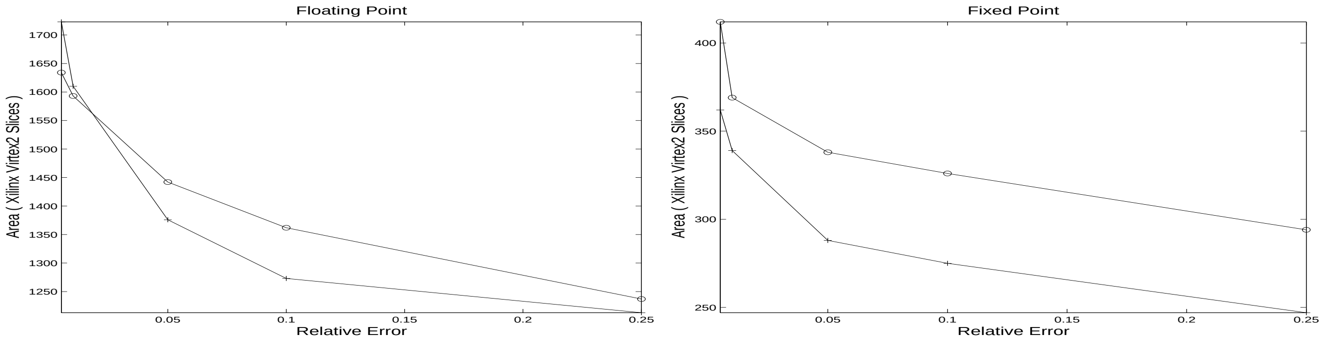

Figure 8. The variation in area with maximum output error for the approximation of $ln(x)$ (+) and $xlog(x)$ (o), for both floating-point and fixed-point implementations on a Xilinx Virtex II device.

both the fixed-point and floating-point implementations for a relative output error of 5%. For the $xlog(x)$ function approximation, with a similar output error specification an 18% reduction is possible for the fixed-point implementation, while a 12% reduction is possible for the floating-point implementation.

The maximum dynamic range for this example is 10^{10}. Hence this design, which contains the same number of multipliers and adders, tends to favor a fixed-point implementation. If the dynamic range increases beyond 10^{16}, the floating-point implementation would become more area efficient.

FIR Filtering

For this case study, we look at bit-width optimisation of an FIR filter. The result of the area versus output error specification is shown in Figure 9.

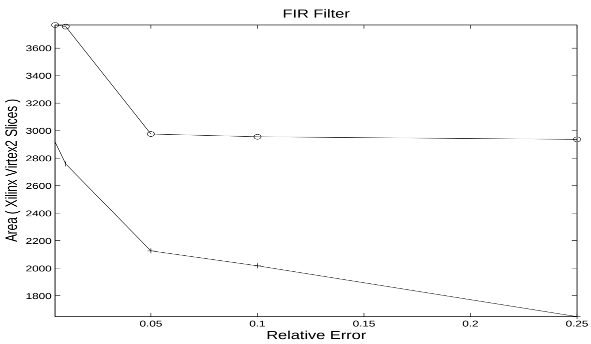

Figure 9. The variation in area with output error for the floating-point (+) and fixed-point (-) implementation of the FIR filter.

The dynamic range of the calculations in this example is 10^{16}. The floating-point implementation on average uses 30% less area than the fixed-point implementation for a given error specification. Our experimental results show that when the dynamic range of the input remains below 10^{12}, the fixed-point design would become more area efficient.

Discrete Cosine Transform

The last case study we consider is a design for 8-point discrete cosine transform. This transform is commonly used in many image compression algorithms, including JPEG and MPEG. The design is described using the Xilinx System Generator and analysed with our BitSize tool. We consider the area-usage of the design for various output error specifications.

Output Error (%)	0.0	0.1	0.2	0.5	0.75
Flip Flops	2130	1615	1539	1161	1127
LUTs	2136	1709	1612	1373	1322
Embedded Mults	60	54	54	46	46

Table 4. The variation of area usage with relative error for DCT implementations in fixed-point. The dynamic range of the designs is 10^2.

Output Error (%)	0.0	0.1	0.2	0.5	0.75
Flip Flops	13012	11800	11400	10779	10724
LUTs	24207	21507	20649	19536	19482
Embedded Mults	16	16	16	16	16

Table 5. The variation of area usage with relative error for DCT implementations in floating-point. The dynamic range of the designs is 10^2.

The results in Tables 4 and 5 show that for a similar output error specification, the fixed-point implementation requires fewer than 10% of the LUTs and flip-flops in the floating-point implementation. On the other hand the floating-point designs use 70% fewer embedded multipliers than the fixed-point designs. Therefore based on the user-specified constraints on area, if there is a tight constraint on the available embedded multipliers, the floating-point implementations would be selected, while the fixed-point designs will be selected if there is a tight constraint on the LUTs or flip-flop usage. In addition, if the dynamic range of the design is increased as illustrated in Figure 7, the floating-point designs would seem more promising since their rate of increase in resource usage with dynamic range is smaller than fixed-point designs.

6 Conclusion

We have presented a method for automatic determination of operator bit-widths for hardware design, which is useful not only for reconfigurable computing but also for VLSI design in general. We show that our framework, based on automatic differentiation, provides a unified treatment for bit-width optimisation of both fixed-point and floating-point designs. Current and future work includes improving the interface between the BitSize tool and other related tools such as Xilinx System Generator and Handel-C, enhancing our approach to support power consumption optimisation and hardware/software co-design, and extending our method to cover (a) other number representations and (b) designs with multiple number representations.

Acknowledgements

We thank David Gay for initial discussions on automatic differentiation. Many thanks to Tim Todman, Dong-U Lee, Ray Cheung, Per Haglund and Jun Jiang for their assistance. The support of Xilinx, Inc., the ORS Award Scheme and the UK Engineering and Physical Sciences Research Council (Grant numbers GR/R 31409, GR/R 55931 and GR/N 66599) is gratefully acknowledged.

References

[1] A. Abdul. Gaffar, W. Luk, P. Y. K. Cheung, N. Shirazi, and J. Hwang. "Automating customisation of floating-point designs". *Field-Programmable Logic and Applications, LNCS 2438*, Springer, 2002.

[2] A. Abdul. Gaffar, O. Mencer, W. Luk, P. Y. K. Cheung, and N. Shirazi. Floating-point bitwidth analysis via automatic differentiation. *Proc. IEEE Int. Conf. Field-Programmable Technology*, IEEE, 2002.

[3] P. Belanovic and M. Lesser. A library of parameterised floating point modules and their use, *Field-Programmable Logic and Applications, LNCS 2438*, Springer, 2002.

[4] M. Budiu, S. Goldstein, K. Walker, and M. Sakr. "BitValue inference:detecting and exploiting narrow bitwidth compilations", *Proc. EuroPar Conf.*, June 2000.

[5] R. Cmar, L. Rijnders, P. Schaumont, S. Vernalde, and I. Bolsens. "A methodology and design environment for DSP ASIC fixed point refinement". *Proc. Design Automation and Test Europe Conf.*, 1999.

[6] G. A. Constantinides. Perturbation analysis for wordlength optimization, *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, IEEE Computer Society Press, 2003.

[7] G. A. Constantinides, P. Y. K. Cheung, and W. Luk. "The multiple wordlength paradigm". *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, IEEE Computer Society Press, 2001.

[8] P. D. Fiore. *A Custom Computing Framework for Orientation and Photography*, PhD thesis, Massachusetts Institute of Technology, 2000.

[9] A. Griewank et al. A package for the automatic differentiation of alogorithms written in C/C++. Technical report, Technical University, Dresden Germany, Ohio University, March 1999.

[10] A. Griewank and G. F. Corlis, editors. *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, Society for Industrial and Applied Mathematics, 1991.

[11] A. Jaenicke and W. Luk. Parameterised floating-point arithmetic on FPGAs, *Proc. IEEE Int. Conf. on Acoustics, Speech, and Signal Processing*, IEEE, 2001.

[12] K. Kum and W. Sung. "Combined word-length optimization and high-level synthesis of digital signal processing systems", *IEEE Trans. on Computer-Aided Design*, August 2001.

[13] B. Lee and N. Burgess. A dual-path logarithmic number system addition/subtraction scheme for fpga. *Field-Programmable Logic and Applications, LNCS 2778*, Springer, 2003.

[14] D.-U. Lee, W. Luk, J. Villasenor, and P. Y. K. Cheung. Non-uniform segmentation for hardware function evaluation. *Field-Programmable Logic and Applications, LNCS 2778*, Springer, 2003.

[15] J. Liang, R. Tessier, and O. Mencer. Floating point unit generation and evaluation for FPGAs. *Proc. IEEE Symp. on Field Programmable Custom Computing Machines*, IEEE Computer Society Press, 2003.

[16] O. Mencer, D. J. Pearce, L. W. Howes, and W. Luk. Design space explorartion with a Stream compiler. *Proc. IEEE Int. Conf. Field-Programmable Technology*, IEEE, 2003.

[17] S. Mahlke et al. "Bitwidth cognizant architecture synthesis of custom hardware accelerators", *IEEE Trans. on Computer-Aided Design*, November 2001.

[18] R. Razdan. *PRISC: Programmable Reduced Instruction Set Computers*, PhD thesis, Harvard University, 1994.

[19] S. A. Wadekar and A. C. Parker. "Accuracy sensitive wordlength selection for algorithm optimization", *Computer Design: VLSI in Computers and Processors*, 1998.

[20] M. Stephenson, J. Babb, and S. Amarasinghe. "Bitwidth analysis with application to silicon compilation", *Proc. SIGPLAN conference on Programming Language Design and Implementation*, ACM, June 2000.

[21] Xilinx Inc., San Jose, USA. *Xilinx System Generator*, 2003.

[22] C. Yap and T. Dube. The exact computation paradigm, *Computing in Euclidean Geometry*, World Scientific Press, 1995.