A Software Tool for Designing Fixed-Point Implementations of Computational Data Paths for Embedded and Reconfigurable Computational Environments

David M. Buehler
dbuehler@cambr.uidaho.edu
University of Idaho

Gregory W. Donohoe
donohoe@cambr.uidaho.edu
University of Idaho

Pen-Shu Yeh
pen-shu.yeh@gsfc.nasa.gov
NASA GSFC

1 Introduction

In computational environments where the cost of floating-point circuitry is prohibitive, such as deeply embedded computing and reconfigurable computing, computations requiring values from the mathematical field of reals can often be performed using fixed-point representations. However, designing fixed-point implementations of computational algorithms has a history of being a difficult, time-consuming and error-prone task which can result in sub-optimal implementations due to unnecessarily high computational error or even incorrect results.

In conjunction with the Field Programmable Processor Array (FPPA) project, the software tool `SIFOpt` has been developed which helps algorithm designers create fixed-point implementations of computational algorithms. `SIFOpt` performs a static analysis of the computation, determining scaling factors for computed variables, alignment operations for additions and subtractions, rescaling operations for multiplications and determining optimal representations for the constants which are used in the computation.

2 Fixed-Point Implementations of Computational Algorithms

A fixed-point implementation of a computational algorithm has something of a dual nature. At *run time*, the computational operations and computed values are integer operations and integer values. However, at *design time* a scaling factor is associated with each value which will be computed at run time. Thus the integer run time values *represent* real values, which are determined by multiplying a run time integer value with the design time scaling factor associated with it.

The choice of the scaling factor for each computed value is left for the algorithm implementor to determine. In a fixed word length computational environment, the choice of a particular scaling factor establishes both the range of real values that can be represented (the product of the run time integer and the scaling factor) and the granularity of the values that are represented. As the scaling factor is increased, value range increases and granularity decreases. As the scaling factor is decreased, value range decreases and granularity increases.

At run time, if the result of a computation is outside the range of values which can be represented, an overflow condition exists. The consequences of the overflow condition range from loss of accuracy in the result to incorrect results.

The primary challenge for the algorithm designer is to assign scaling factors to each computed value which are as small as possible yet not so small that they result in overflow errors at run time. In addition, the choice of scaling factors effects the operations required to align values for addition operations, and the operations required to prescale values for multiplications (in computational environments where the result of a multiplication are not as wide as the sum of the widths of the multiplicands.)

3 Related Research – Dynamic Approaches

Recent research into creating fixed-point implementations of computations have focused on dynamic approaches. With a dynamic approach, the computation is implemented in a floating-point environment and the computed values are “instrumented” to collect statistics about their run time values.

Sample data sets are then run through the instrumented versions of the computation and run time statistics are gathered. The run time statistics are then used to assign scaling factors to each computed value. This step acts as a training step which is used to "learn" the scaling factors to use for each computed value.

Researchers from Seoul National University [6], The University of Toronto [1] and Aachen University of Technology [5] use this approach to perform automatic floating-point to fixed-point conversion, taking a C program which includes floating-point variables and constants and generating a C program which uses only integer variables and constants.

Researchers at the University of Washington [3] and MIT [9] use this approach to minimize the data path width of FPGA and ASIC implementations of fixed-point computations.

4 The Static Approach

The static approach we use is based upon a design-time notation which tracks the partitioning of the run-time integer values into Sign, Integer and Fraction (`SIF`) regions. Taken together, the Integer and Fraction regions of a run time value are similar to the mantissa region of a floating-point value representation, so they (the Integer and Fraction regions together) are referred to as the mantissa region.

4.1 `SIF` Partitionings

The notation we use to indicate the design time partitioning of the run time integer values in the computation originated with researchers working at Kansas State University and Sandia National Laboratories in the late 1970s under the name Block Floating Point notation (BFP) [8]. Members of our group were exposed to the notation at Sandia National Laboratories and suggested it as a starting point for a fixed-point design tool.

Though the notation itself remains the same, the theory underlying the original work has been completely reworked and reintroduced with the name "`SIF` partitionings" in [2].

`SIF` partitionings are associated with the values which will be computed at run time. The `SIF` partitioning indicates how the integer value is partitioned between replicated sign bits, bits carrying integral information, and bits carrying fractional information. In addition, an `SIF` partitioning in conjunction with its integer's word length indicates how many of the integer's bits are unused, right-padding bits.

The notation used for `SIF` partitionings is described by the following specification. Characters in single quotes are literal characters, characters inside square brackets are optional, and a horizontal bar separates list entries of an exclusive-or choice between a list of characters.

$$\text{`(' [ `+' | `-' ]}\ v_S\ \text{`/'}\ v_I\ \text{`/'}\ v_F\ \text{`)' [`\^{}'}\ v_n\ \text{]}$$

The optional '`+`' or '`-`' character before the value of v_S indicates if the run time value is known to be positive (in the case of a '`+`') or negative (in the case of a '`-`'.) v_S is a non-negative integer indicating how many of the run time value's bits (beginning from the most-significant bit) are replicated sign bits. v_I is a non-negative integer indicating how many of the bits following the replicated sign bits represent integral information. v_F is a non-negative integer indicating how many of the bits following the integral bits represent fractional information. v_n is an optional integer value parameter which indicates an additional shift of the binary point by v_n positions. (If $v_n < 0$, the binary point is shifted to the left, otherwise it is shifted to the right.)

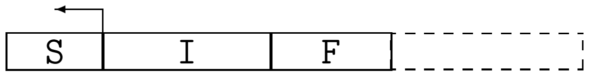
4.2 Operations on `SIF` partitionings

Given an `SIF` partitioning, we define operations which can be used to adjust the boundaries of the regions defined by the `SIF` partitioning.

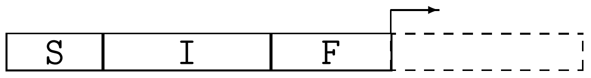
4.2.1 Moving the boundaries of the mantissa region

There are four operations on `SIF` partitionings which adjust a boundary of the mantissa region. These operations are not usually required for implementing a computation, and should only be used in "extraordinary circumstances" to represent designer knowledge of non-computational behaviors.

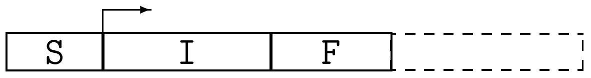
Expand Left (exl) - moves the boundary between the sign bits and the mantissa to the left, causing some bits which were sign bits to be interpreted as mantissa bits.



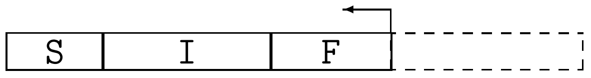
Expand Right (exr) - moves the boundary between the mantissa region and the unused (padding) region to the right, causing some bits which were interpreted as unused padding bits to be inprepreted as mantissa bits.



Truncate Left (trl) - moves the boundary between the mantissa region and the sign region to the right, causing some bits which were interpreted as mantissa bits to be interpreted as sign bits.



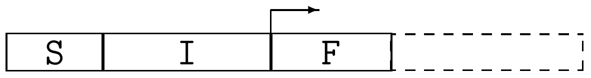
Truncate Right (trr) - moves the boundary between the mantissa region and the unused (padding) region to the left, causing some bits which were interpreted as mantissa bits to be interpreted as padding bits.



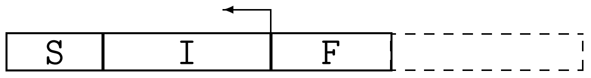
4.2.2 Moving the binary point

Moving the boundary between the integer an fractional regions has the effect of multiplying and dividing the real value represented by the run time integer by powers of two, with no run-time operation. Note that the binary point can lie outside the mantissa region, and even beyond the edge of the machine word.

Shift Binary Point Right (sbpr) - shift the location of the binary point to the right. The value mapped to by the integer and scaling factor is divided by 2^n.



Shift Binary Point Left (sbpl) - shift the location of the binary point to the left. The value mapped to by the integer and scaling factor is multiplied by 2^n.

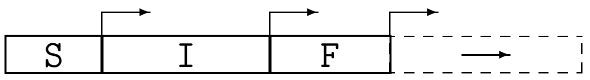


4.2.3 Run time shift with `SIF` partitioning adjustment

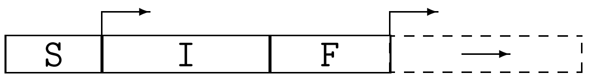
In addition to boundary manipulation operations, we define operators which correspond to run time shifting of the integer value.

When the integer value is *right* shifted at run time, we can interpret the result as either adding sign bits (binary point shifts with integer), or dividing by a power of two (binary point retains its previous location.) When the integer value is shifted *left* at run time, we can interpret the result as either removing sign bits (binary point shifts with the integer) or as multiplying by a power of two (binary point retains its original location.)

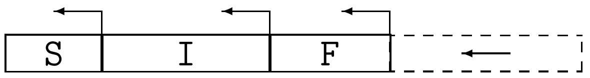
Add Sign Bits (asb) - All boundaries are shifted the same amount as run time shift. The real value represented by the integer stays the same, up to truncation error.



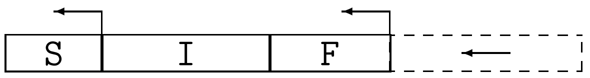
Divide By Shifting (dbs) - The location of the binary point is held constant while the mantissa boundaries are shifted along with the run time value. The real value represented by the integer is divided by 2^n.



Remove Sign Bits (rsb) - All boundaries are shifted along with the run time value. The real value represented by the integer stays the same (up to overflow.)



Multiply By Shifting (mbs) - The location of the binary point is held constant while the mantissa boundaries are shifted along with the run time value. The real value represented by the integer is multiplied by 2^n.



5 `SIFOpt`

The `SIFOpt` software tool creates fixed-point implementations based on a description of the computation that is to be performed. This description must have annotations specifying the fixed-point formats of the computation's inputs. In addition, the designer can annotate the computation's description with information that is specific to the run time data on which the computation will be performed.

`SIFOpt` performs a static analysis of the described computation, extracting a computation tree and determining the following:

- Scaling factors (in the form of `SIF` partitionings), (integer) value ranges and the estimated maximum absolute error for each computed value
- Alignment operations required for addition and subtraction operations
- Rescaling operations required for multiplications
- Optimal integer equivalent values for constants

`SIFOpt` can produce several types of output. The default output is to print the computation tree annotated with the computed fixed-point information. `SIFOpt` can also produce integer-only C code. Finally, `SIFOpt` can produce code which will use a C++ mixed-point class which I have created as part of this work (and which will be described later in this chapter.)

6 Description of a numerical computation

An algebraic language has been implemented in which the algorithm designer specifies the numerical computation to be performed. This language supports two kinds of named values: variables and constants. Named values must be declared before (or when) they are used. Variables which are input values of the computation must be annotated with `SIF` partitionings, and can optionally be annotated with value range information. Other variables in the computation can be annotated with `SIF` partitionings and/or value ranges. This allows the algorithm designer to specify information about run time dependencies which cannot be determined by the static analysis.

6.1 Expressions

Expressions available to the algorithm designer are similar to algebraic expressions available in most high-level languages.

6.1.1 Binary mathematical operators

`SIFOpt` provides the three binary mathematical operators: '`+`', '`-`' and '`*`', which provide addition, subtraction and multiplication, respectively. The '`*`' operator has higher precedence than the '`+`' and '`-`' operators. Arguments of a string of '`+`' and '`-`' operators are grouped left-to-right. Parenthesis ('`(`', '`)`') can be used to group binary operations for clarity or to override precedence. Division will be added in the near future.

6.1.2 Unary mathematical operator

The '`-`' unary negation mathematical operator is also provided, and has higher precedence than multiplication. At this moment it is only available for literal Real values.

6.1.3 Word length specification operator

The word length specification operator "@n" is provided as a postfix operator, and has higher precedence than unary negation.

6.1.4 Built-in functions

All of the unary `SIF` operations introduced in section 4.1 (`exl`, `exr`, `trl`, `trr`, `sbpl`, `sbpr`, `asb`, `rsb`, `dbs` and `mbs`) are available in the computation description language.

In addition, the functions `C`(*expr*) and `NC`(*expr*) are provided to specify that the outermost binary operation of the enclosed expression will result in a carry-out or will not result in a carry out, respectively.

7 Static analysis

`SIFOpt` creates a fixed-point implementation of a computational algorithm by performing a static analysis of the computation, which is specified by a `SIFOpt` input file. An "optimization tree" (which mirrors the input file's parse tree) is constructed by performing a post-order traversal of each statement in the parse tree. All values in the algorithm description must have `SIF` partitionings associated with them before they are used in an expression. This limits the type of computations which can be input to `SIFOpt` to computations which have parse trees that can be expressed as directed acyclic graphs (DAGs).

7.1 Single assignment

Single assignment[1] languages [7] meet the requirement that the input algorithm be expressible as a DAG. The input language for `SIFOpt` now restricts the types of algorithms specified to single assignment algorithms.

7.2 Propagation of `SIF` partitionings and value ranges

The most fundamental task that `SIFOpt` performs is to compute `SIF` partitionings and value ranges for the results of mathematical operations. These result values are computed from the `SIF` partitionings and value ranges of the arguments of each mathematical operation by performing a post-order traversal of the computation tree.

Value ranges are propagated via Interval Arithmetic by default. An option is available to cause `SIFOpt` to exhaustively compute value ranges for nodes of the computation tree which have common variables in the two child trees. Exhaustive computation is not in general computationally feasible, but we have found it useful for the size of the problems we are currently addressing.

Using these methods of value range propagation, computed values will not overflow so long as the inputs stay within the bounds specified by the designer. However, computed values may be subject to unnecessarily reduced precision. Reduced precision can be caused by correlations between the run time input values, or dependencies between values in the computation tree (if exhaustive computation of value ranges is turned off.) The user can use the `C()` and `NC()` functions (mentioned in section 6.1.4) or specify value ranges for result values in order to correct this problem, but use of these language features make run time overflow errors possible.

7.2.1 Placement of the result within the word length

In cases where there is flexibility in the placement of the result value within the result word, `SIFOpt` aligns operands using a heuristic which depends on the operation being performed.

The heuristic for determining what shift operations to perform in order to align values for addition (and subtraction) is:

1. If there is only one alignment which minimizes truncation error in the result, that alignment is used.
2. Otherwise, if the operands can be aligned by shifting only one of the operands, only that operand is shifted.
3. Otherwise (both operands have to be shifted to align the binary point) the number of sign bits in the result are maximized.

That is, our first priority is minimizing the number of shift operations which must occur, and if both arguments must be shifted, then our priority is maximizing the number of sign bits in the result. The reason we maximize the number of sign bits in the result as a second priority is that unexpected overflow which might occur will not be immediately harmful if there are additional sign bits into which the result value can overflow.

The heuristic for determining the shift operations to perform in order to prescale values for multiplication is:

1. If there is only one prescaling which minimizes truncation error in the result, that prescaling is used.
2. Otherwise, if the values are already appropriately scaled, then no scaling operations are added.

[1]As the name implies, single-assignment languages allow variables to be assigned to only once, when they are defined.

3. Otherwise if the values can be prescaled by shifting only one of the multiplicands, then that multiplicand is prescaled to be as small as possible.

4. Otherwise both multiplicands are prescaled to be as small as possible.

The first priority of the heuristic for multiplication is to minimize the number of shift operations required for prescaling. The second priority is to maximize the number of sign bits in the result.

For addition, the static analysis stores information about the alignment operation implemented and how much flexibility there was in the decision about how to align the addends in the addition tree node. This allows the optimizer to re-visit the node and request a change in the alignment operation(s), in an attempt to minimize number of shift operations required *overall* by the implementation. However, this feature is currently "stubbed out" in the `SIFOpt` code and has no effect.

For multiplication, the first case may occur in such a way that we can preserve one more bin in one multiplicand than in the other, and there is no basis for choosing which multiplicand gets the extra bit preserved. In this case, the arbitrary choice is made to preserve an extra bit in the left hand argument of the multiplication.

7.3 Constants

Constant values in the computation can be optimized quite effectively. Rounding is performed on all constants.

All constant values, whether literal values (e.g. "3.14") or named constants (knowns) can be followed in the `SIFOpt` algorithm specification by an `SIF` partitioning that specifies the format to be used, in particular the number of significant bits to be used. If the user specifies an `SIF` partitioning for a constant value, and the requested value for the number of sign bits is smaller than the number `SIFOpt` determines using the heuristics to be described below, then `SIFOpt` ignores the `SIF` partitioning and optimizes the constant independently of the following heuristics. Therefore, when used with constant values, providing a v_S value of zero tells `SIFOpt` to compute an optimal `SIF` partitioning for the constant.

7.3.1 Optimal `SIF` partitioning for a constant

Given an `SIF` partitioning and a word length, I define their optimal `SIF` partitioning to have the minimum number of sign bits possible (1 if the constant is negative, 0 otherwise), and as many unused bits as there are 0's to the right of the right-most 1 in the constant's (rounded) integer representation within the given word length.

7.3.2 Optimizing constant values for addition and subtraction

Ideally, when we wish to compute the sum of a constant and a variable we will not be required to shift the variable before performing the addition. When this is possible, the value computed for the constant will have the same scaling factor as the variable's scaling factor, and will use only as many bits of resolution as are available in the machine word.

Using this heuristic it is possible that the desired constant will fall outside both the range of significant bits for the variable and the size of the word length, and thus become 0. In this case a warning is printed. In cases such as this, the philosophy I've followed with `SIFOpt` is that the designer should be alerted to the fact that they've added a value having only n fractional bits with a value whose most significant bit lies more than n places to the right of the binary point. However, a means is provided for the designer to override the way `SIFOpt` normally works. As noted in section 7.3, an `SIF` partitioning can follow any constant value, and setting the number of sign bits to 0 in that `SIF` partitioning tells `SIFOpt` to compute the optimal number of sign bits on its own. Providing an `SIF` partitioning for the constant is the only way to cause a variable to be left shifted (remove sign bits) when adding with a constant value.

There are two cases which can cause sign bits to be added to the variable before summing with a constant. First is the case in which sign bits must be added for overflow protection. Second is the case in which the constant value extends to the left of the variable and the variable's sign bit region. This case occurs when equation 1 holds. In both cases, the variable is right-shifted as little as possible. The decision to shift as little as possible made it easy to get the maximum resolution in the fixed-point representation of the constant.

$$\log_2(\texttt{var}) < \log_2(\texttt{const}) \tag{1}$$

7.3.3 Optimizing constant values for multiplication

Multiplication by constant values is implemented by first computing an optimal `SIF` partitioning for the constant value (up to 32 bits) then using the heuristics given in section 7.2.1 to determine any required rescaling operations. Recall however, that an arbitrary decision was made that when the number of mantissa bits allowed in the multiplicands is odd an extra bit would be removed from the right-hand argument of the multiplication operator.

When one of the arguments is a constant we know the values of all of the bits in its integer representation, therefore we can optimize the choice of which argument has extra mantissa bit preserved by checking the actual value of the bit which might get truncated from the constant and truncating it if it is a zero. On the other hand, if the bit is a one in the constant, I choose to preserve it on the philosophy that it is better to preserve a known 1 bit then to preserve a bit which may or may not be a 1.

This leads to the second optimization. If we detected that the constant's least significant preserved bit is a zero, we should check the next-least significant bit, and if it is also a zero then preserve an additional bit from the variable, and so on.

When no rescaling is required, the constant will be implemented so that the result has the maximum number of sign bits possible.

7.4 Estimating truncation error

The static analysis performs one additional function: estimation of truncation error. With every right shift, the maximum truncation error is assumed to occur (i.e. it is assumed that all truncated mantissa bits held values of 1.) Each multiplication is assumed to multiply by the maximum magnitude value in its multiplicand's value range (resulting in the maximum error magnification possible.)

8 `SIFOpt` output

`SIFOpt` can generate three types of output. The first type of output is comprised of information about the fixed-point implementation that has been computed. The second type of output is integer-only C code. The third type is C++ code which uses my "mixed-point" C++ class.

8.1 Fixed-point implementation information

The fixed-point implementation information output from `SIFOpt` is composed of two sections.

The first section displays the optimization tree created by `SIFOpt`. Every node in the computation is displayed on one or more lines, indented in tree hierarchy. Variable declarations display the `SIF` partitioning and Real value range that was either declared or determined for the variable. Known declarations display their Real value, their optimal integral value and optimal `SIF` partitioning. Expressions display information about the computation being performed, including `SIF` partitioning, value range, absolute error, information about operator alignment and prescaling operations implemented, clipping and padding information.

A short extract of an optimization tree follows. This extract covers the statement `out1 = w2 * in1 + out0;` from a convolution implementation. This is not verbatim output from `SIFOpt`, some cleanup has been performed by hand. Note that absolute error values are placed between balanced '\' and '/' characters.

```
Assignment
    Variable - out1: (1/0/31)^-1, [-0.499985:0.499985] \1.52583e-06/
=
    Operation: AddSub - k = 0, kmin = 0, kmax = 0
    BinaryOp: '+' result is: (1/0/31)^-1, [-0.499985:0.499985] \1.52583e-06/
        Operation: Mult
        BinaryOp: '*' result is: (1/0/31)^-1, [-0.349988:0.349988] \1.52583e-06/
            Known - w2 = 0.35 as 45875=0xb333 in 32 bits with (+16/0/16)^-1 =
                                                    0.349998 \1.52588e-06/
            ShiftResize - m_delta = 0, m_shift = 16 (17/0/15), [-0.999969:0.999969]
                Variable - in1: (1/0/15), [-0.999969:0.999969]
        Variable - out0: (2/0/30)^-2, [-0.149997:0.149997]
```

After the optimization tree, SIFOpt prints a list of the variables in the computation with the SIF partitionings, the range of real values that can be represented, the range of values on the underlying integer (optional) and absolute error values that have been computed (if non-zero). The output from the convolution example is shown below.

```
in0 - (1/0/15) [-2147418112:2147418112]->[-0.999969:0.999969]
in1 - (1/0/15) [-2147418112:2147418112]->[-0.999969:0.999969]
in2 - (1/0/15) [-2147418112:2147418112]->[-0.999969:0.999969]
in3 - (1/0/15) [-2147418112:2147418112]->[-0.999969:0.999969]
out0 - (1/0/30)^-2 [-1288463974:1288463974]->[-0.149997:0.149997] \1.52583e-06/
out1 - (1/0/31)^-1 [-2147418112:2147418112]->[-0.499985:0.499985] \3.05166e-06/
out2 - (1/0/31) [-1825302119:1825302118]->[-0.849973:0.849973] \4.57796e-06/
out3 - (1/0/31) [-2147418113:2147418111]->[-0.999969:0.999969] \6.10403e-06/
```

8.2 Integer-only C code

The command-line switch "-C" can be used to tell SIFOpt to generate integer-only C code. The output from the convolution example is given below.

```
int in0;
int in1;
int in2;
int in3;
int out0 = (39322 * ((in0) >> 16));
int out1 = ((45875 * ((in1) >> 16)) + ((out0) >> 1));
int out2 = ((((45875 * ((in2) >> 16))) >> 1) + ((out1) >> 1));
int out3 = ((((39322 * ((in3) >> 16))) >> 2) + out2);
```

8.3 Mixed-point C++ code

The command-line switch "-M" can be used to tell SIFOpt to generate output that targets my "mixed-point" C++ class. The output has been slightly cleaned up to remove some redundant parenthesis.

```
mixedpoint in0;
mixedpoint in1;
mixedpoint in2;
mixedpoint in3;
mixedpoint out0 = (mixedpoint::constant( 0x999a, -18, 0.15, 0) * (in0 >> 16));
mixedpoint out1 =
  (mixedpoint::constant( 0xb333, -17, 0.35, 0) * (in1 >> 16)) + (out0 >> 1);
mixedpoint out2 =
  ((mixedpoint::constant( 0xb333, -17, 0.35, 0) * (in2 >> 16)) >> 1) + (out1 >> 1);
mixedpoint out3 =
  ((mixedpoint::constant( 0x999a, -18, 0.15, 0) * (in3 >> 16)) >> 2) + out2;
```

9 Mixed-point C++ class

The mixed-point class is intended to be used to compare a fixed-point implementation of a computation with a floating-point implementation of the same computation. As a computation is performed using the mixed-point data type, both a fixed-point value and a floating-point value are computed for every mathematical operation performed.

9.1 Mixed-point data

Each mixed-point variable carries the following information:

- An integral value: I.
- A value for the exponent of the scaling factor (an integer): e.
- A long double-precision truncation error value: t.
- A double-precision floating-point value: R.

The mixed-point class currently has no notion of word length. Values of algorithms implemented with limited word length will be found in the least significant bits of mixed-point values, so overflow errors caused by reduced word length will not be modeled correctly.

Accessor functions are provided for the values of I, t, R and $I \cdot 2^e$ (which is the real value mapped to by the fixed-point representation.)

Most of the mathematical operators have been overloaded to work with the mixed-point class. The exceptions are "/=", "/", "%", "%=", "++" and "- -". The shift operators have all been overloaded, and are interpreted as adding and removing sign bits. The function `mixedpoint::ldexp( const mixedpoint&, int )` is provided for manipulating the scaling factor exponent, similar to the `ldexp` function defined in the C language [4].

In normal use equation 2 will hold. This equation is only an approximation because both t and R are subject to their own truncation errors, most likely at different resolutions.

$$R \approx (I \cdot 2^e) + t \tag{2}$$

9.2 Truncation error

Right shift operations are the only source of truncation error at runtime. Any time a right-shift operation is performed, the bits which will be shifted out of the word are extracted into a floating-point value, multiplied by the scaling factor and added to any existing truncation error already held by the variable.

Constant values can be a source of design-time truncation error. Constants are declared by providing the integer (I) and scaling-factor exponent (e), along with a double-precision value (R). Truncation error is computed in the mixed-point constant constructor as the difference between R and $I \cdot 2^e$.

9.3 Use of the mixed-point implementation

Mixed-point implementations have been used for the purpose of debugging and evaluating fixed-point algorithm implementations generated by `SIFOpt`.

10 Results

SIFOpt has been used to create fixed-point implementations of many algorithms. A few of the interesting results are presented here. Each algorithm was actually implemented in C++ using the mixed-point class described in section 9 along with additional "instrumentation" to collect maximum and minimum values and run time truncation error amounts. The tests were run on a system with 32 bit integer values, and in most cases (exceptions noted) SIFOpt was set to allow the integer values to grow that large.

There are two measures of interest in gauging the quality of SIFOpt's output. First is the question of whether the fixed-point implementation utilizes the full range of bits for each computed value. Second is how closely the truncation error estimate is to actual run time truncation error amounts.

10.1 8-weight convolution

Several 8-weight convolutions were implemented with a variety of weight distributions, but always with the weight values summing to 1.0. The frequency responses of the resulting filters were computed, and "rail-to-rail" inputs for a variety of the frequency responses were used to test the fixed-point implementations. The input values were encoded as 16-bit values.

Several implementations, differing in internal data path widths, were created: 32-bit internal data paths, 32-bit multiplication results reduced to 16 bits post-multiply and 16-bit internal data paths (multiplications have 16-bit arguments and 16-bit results.)

In each case, the implementation generated by SIFOpt used the full range of bits for each computed value – at least for signals of a frequency that should be "passed".

Implementations having a 32-bit internal data path (hence requiring very few truncations of bits) had truncation error closely matching the estimates computed by SIFOpt. In the other cases, SIFOpt's estimates were pessimistic by several orders of magnitude.

(Note that there are no static leaf dependencies for any of the computed values, so the exhaustive value range computation method is not applicable to this computation.)

10.2 Division by repeated multiplications

The division by repeated multiplications algorithm was implemented to check the accuracy of an 8-bit by 8-bit division with just eight numerator computations. The computation tree has static dependencies between the leaves of some computed values. Before the exhaustive value range computation was implemented the designer had to identify computed values for which SIFOpt incorrectly determined value ranges. These nodes were easily identified due to known characteristics of the computation, and protection of the computations using the NC() function resulted in fixed-point implementations which used the full range of bits for each computed value. The exhaustive computation of value ranges eliminates the need for the designer to protect any computations with NC() functions, and results in an improved fixed-point implementation at the nineth numerator computation and beyond.

After the third numerator computation in the algorithm, large numbers of bits begin to be truncated. Up to that point in the computation, SIFOpt's absolute error estimation tracked within an order of magnitude with run time truncation amounts. Beyond that point, the error estimation and run time error amounts began to diverge, and the estimate of the truncation error in the result was several orders of magnitude out of line with run time truncation amounts.

10.3 Goertzel's Fourier Transform algorithm

The challenge with implementing Goertzel's Fourier Transform algorithm was that the computation has feedback from the output to the inputs. Because SIFOpt requires us to provide an SIF partitioning for each input, we had to determine an SIF partitioning to use as a starting point. This was achieved by implementing the computation in a floating-point environment and running the computation on sample data sets. The resulting maximum magnitude value was then used to fix an `SIF` partitioning for the algorithm inputs and `SIFOpt` was used to generate a fixed-point implementation with this information. Special care had to be taken to ensure that the scaling factor of the resulting output matched the scaling factor of the feedback values.

Due to the fact that we pre-computed the maximum feedback value using datasets that matched our testing and the very short computation performed, the fixed-point implementation was virtually guaranteed to have little truncation error and to use the full range of bits. The interesting part of this example was having to deal with a loop in the computation tree.

11 Conclusions

`SIFOpt` has been used effectively to create fixed-point implementations of computational data paths for the Field Programmable Processor Array. We have demonstrated applications including FIR filtering, image pixel non-uniformity correction, and discrete Fourier transform algorithms. In each case, we were able to use `SIFOpt` to create a fixed-point implementation in which runtime values utilize the full range of integer bit positions.

Estimation of run time truncation error is inaccurate for any implementation which has large numbers of truncated bits.

We intend to work on methods for noting run time dependencies between input values which will result in loss of precision, and to add decision-making language features in the near future.

References

[1] Tor Aamodt and Paul Chow. Embedded ISA support for enhanced floating-point to fixed-point ANSI C compilation. In *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embeded Systems*, pages 128–137, 2000.

[2] David M. Buehler. *A Methodology for Designing and Analyzing Fixed-Point Implementations of Computational Data Paths*. PhD thesis, University of Idaho, 2004.

[3] Mark L. Chang and Scott Hauck. Precis: A design-time precision analysis tool. In *Proceedings of the 2002 IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[4] Samuel P. Harbison and Guy L. Steele Jr. *C: A Reference Manual*. Tartan, Inc., fourth edition, 1995.

[5] Holger Keding, Frank Hürtgen, Markus Willems, and Martin Coors. Transformation of floating-point into fixed-point algorithms by interpolation applying a statistical approach. In *Proceedings of the 9th International Conference On Signal Processing Applications and Technology 1998*, 1998.

[6] Ki-Il Kum, Jiyang Kang, and Wonyong Sung. AUTOSCALER for C: An optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Tr. Circuits and Systems II*, 47(9):840–848, September 2000.

[7] Walid A. Najjar, Wim Böhm, Bruce A. Draper, Jeff Hammes, Robert Rinker, J. Ross Beveridge, Monica Chawathe, and Charles Ross. High-level language abstraction for reconfigurable computing. *IEEE Computer*, 38(8), August 2003.

[8] James E. Simpson. A block floating-point notation for signal processes. Technical Report SAND79-1823, Sandia National Laboratories, 1979.

[9] Mark Stephenson, Johathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *Proceedings of the ACM SIGPLAN '2000 Conference on Programming Language Design and Implementation (PLDI)*, June 2000.