Converting Executable Floating-Point Models to Executable and Synthesizable Fixed-Point Models

Taylor L. Riché
National Instruments
Austin, TX, USA
taylor.riche@ni.com

Jim Nagle
National Instruments
Austin, TX, USA
jim.nagle@ni.com

Joyce Xu
National Instruments
Shanghai, People's Republic of China
joyce.xu@ni.com

Don Hubbard
National Instruments
Austin, TX, USA
don.hubbard@ni.com

***Abstract*—Execution on field programmable gate arrays (FPGAs) is now necessary for many areas of algorithm development and prototyping, whether it be for the performance that a hardware implementation gives, or the ability to prove an algorithm works, "in the real world."**

A problem with FPGAs, however, is that the hardware resources are limited. Most algorithm experts design their algorithms using *floating-point math* which gives flexible precision. Floating point is unfortunately expensive to implement in hardware. Therefore, algorithm designers employ experts in *fixed-point math* to transform their algorithm to one that will work in hardware, incurring added cost and time to market.

We present a novel tool as part of the LabVIEW NXG FPGA Module that uses executable model-driven techniques to guide an algorithm expert to a fixed-point version of their original algorithm model. We walk through a case-study for use of our tool, as well as explain the underlying mathematical and model-driven formalisms on which we build the tool.

I. Introduction

Execution on *field programmable gate arrays (FPGAs)* [1] is now necessary for many areas of algorithm development and prototyping, whether it be for the performance that a hardware implementation gives, or the ability to prove an algorithm works, "in the real world." Unfortunately, FPGAs are hard to program. *Digital design*, the art and science of creating hardware implementations of algorithmic models, requires a different mode of thinking than traditional modeling or programming for a general purpose processor. A digital designer takes into account the clock rate of the hardware target, manages the validity of data throughout their design, and even whether or not their code will *physically fit* on the FPGA.

The problem of limited hardware resources is important because most algorithm experts design their algorithms using *floating-point math* which gives flexible precision. Floating point is unfortunately expensive to implement in hardware—in double-precision floating point (a common implementation in most programming environments) each operation requires a full 64 bits and must have logic to deal with special cases of overflow, underflow, divide by zero, etc..

Often, algorithm designers employ experts in both *fixed-point math* as well as digital design to transform their algorithm to one that will work in hardware, incurring the added cost and time to market. Ideally, an algorithm designer could use a tool to design their algorithm model that also abstracts away the complexity of transforming their algorithm to execute in hardware.

We present a novel tool as part of the LabVIEW NXG FPGA Module [2] that uses executable model-driven techniques to guide an algorithm expert to a fixed-point, FPGA-executable version of their original algorithm model. The tool works on executable models built in G, the main graphical, dataflow model of computation in LabVIEW NXG.

While there is a large body of work on automatically moving floating-point models to fixed-point models (see Section V), our work is novel in that it takes a practical, profiling-based approach to the problem. Instead of trying to create the "perfect" solution automatically through analysis of just the algorithm, we use the user's own testbench data to generate an initial suggestion of fixed-point types that work within the constraints they give us. From there, we provide tools that help the user find hot spots of imprecision in the code, and then conveniently make changes too add precision where necessary. The novelty in our contribution is realizing that the simplistic approach of guiding the user toward a quality implementation is more effective in many cases than a fully automatic one.

Because our models are executable, we guide the user to create a transformed model that is *correct-enough by construction*. The user is willing to accept some deviation from correctness (i.e. arithmetic imprecision) to obtain a version of their model that fits on an FPGA. While this concept is unusual in the modeling community, it is a key ingredient in solving the float-to-fixed problem for a user. The user provides us with their error, or correct-enough constraint, and we derive transformations using that constraint from the beginning.

In Section II we dive into more detail about why the transforming of floating-point algorithms to fixed-point versions is difficult for algorithm designers. In Section III, we present the mathematical and modeling formalisms that underlie LabVIEW NXG's fixed-point transition tool. Section IV steps through the user's workflow as they move from floating point to fixed point. Finally, we discuss related work and wrap up in Sections V and VI.

II. Problem Description

A. Challenges of Digital Design

Today, communication and signal processing algorithm designers are increasingly being asked to show that their algorithms work on more than just a white board or in a simulation. Funding agencies and standard boards want to see the algorithm work in, "the real world," and this often means in hardware. For most people, this means making their algorithm work on a field-programmable gate array (FPGA).

FPGAs are much harder to program than a general purpose CPU as the programmer must take into account hardware-centric details such as clock timing and data validity. Also, a design that works perfectly fine in a standard software platform may not even be able to be compiled into hardware due to the fact that only certain parts of hardware design languages (e.g. VHDL [3] or Verilog [4]) are what is referred to as *synthesizable*. Synthesizable means that a compiler is able to take all the structures in the program and turn these into hardware constructs such as flip-flops and look-up tables.Needless to say, expertise is required to design efficient FPGA implementations. Many people new to hardware-design-language programming create code that simulates correctly but would never synthesize into hardware. An example of this is transferring data through variables in a way that creates a latch in hardware—this latch will not synthesize on an FPGA.

However, an expert in communication algorithm design (say somebody working on an encoder for the new 5G wireless standard [5]) rarely is an expert in digital design (the art and practice of creating FPGA implementations) or fixed-point arithmetic. They often have to hire experts in digital design and fixed point to take their golden models to a hardware implementation. We refer to the starting model as "golden" to signify that it represents the most correct version of the model. Algorithm designers do not want to alter the golden model once it is created—this is their reference. So they make a copy to send to the experts in digital design and fixed point, and then iterate back and forth to make sure that these translated models still meet the requirements. The process is time consuming and expensive.

There are many challenges in creating efficient hardware implementations, many of which we mention above, but this paper focuses on one: the need to remove floating-point arithmetic from the golden model.

B. Challenges of Fixed-Point Arithmetic

Algorithm designers use *floating-point math* [6] to design the golden models of their algorithms. Floating-point arithmetic allows for arbitrary precision. Unfortunately, floating-point operations are still relatively expensive to implement on FPGAs. In a double-precision floating point implementation, each operation requires a full 64 bits and must have logic to deal with special cases of overflow, underflow, divide by zero, etc.

To save resources, digital designers use *fixed-point arithmetic (FXP)* [7]. In fixed-point arithmetic, a designer can set the number of bits that represent the integer part of a number and set the number of bits that represent the fractional part of the number. When a computer applies a mathematical operation to a number or numbers (such as add or multiply), a

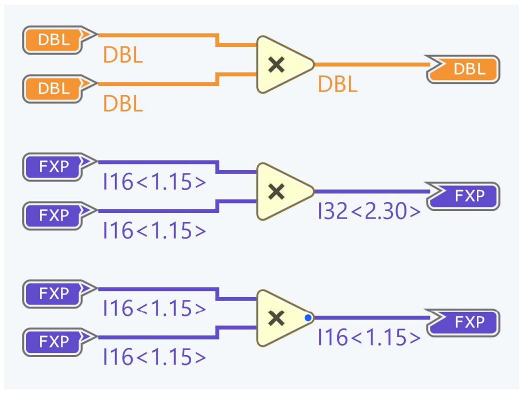


Fig. 1. Three multiplies with different floating-point and fixed-point type output

computer system will either keep the representation the same (and thus lose precision) or grow the number of bits. Often, even growing to keep the full precision of the resultant number is less expensive than floating point. However, often the bits must be constrained to represent a number that is less precise than what the operation would dictate to keep the design from getting too expensive.

Figure 1 shows a basic arithmetic operation of multiply represented in our graphical, dataflow model. The top row shows the multiply using floating-point arithmetic. The middle line we change the inputs to fixed-point types, specifically types that have 1 integer bit and 15 fractional bits. In this example, the type grows to 2 integer bits and 30 fractional bits (we describe some of the rules of bit growth in the fixed-point arithmetic in Section III). This new type is the amount of bits required to deal with any growth of the number (increasing the integer bits) and any required increase in precision (growth in the fractional bits) that comes from multiplying the numbers together. In our environment, we default to fixed-point mathematical operations growing so that they do not lose any data that the user may have. If the user chooses to overwrite this growth by restricting the output type (the third multiply in Figure 1), we designate this restriction with the blue dot at the output of the operator.

The main challenge with using fixed-point arithmetic is selecting which fixed-point types to use. In other words, how many bits are appropriate for each operation. Generally, this process of transforming a floating-point golden model of an algorithm to a fixed-point model of the algorithm is done manually. The original algorithm expert often has to hire an expert in fixed-point arithmetic to transform the model of their algorithm to fixed-point. This extra person adds cost and time to the product or research project.

Figure 2 shows an abstract representation of the fixed-point transformation process. The algorithm designer creates M_{golden} in their preferred high-level, mathematical modeling tool.

Then, the fixed point expert uses a variety of techniques and tools (spreadsheets, MATLAB [8], Simulink [9], etc.) to create

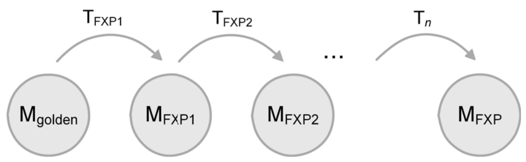


Fig. 2. Standard transformation of the golden model to an FXP version.

$T = T_{FXP1}...T_{FXPn}$. The fixed-point expert applies these transformations to M_{golden} until they reach M_{FXP}, which is defined as the fixed-point model that meets the precision constraints of the algorithm designer, but also works for the particular FPGA on which the designer wants to run their algorithm[1].

The transforms in the sequence T are often all applied manually by the fixed-point expert. They derive these transforms through analyzing the algorithm, often using spreadsheets, and much trial and error, which is expensive.

The series of transforms T are not correct by construction in the strictest sense. The algorithm expert agrees upon bounds of correctness, and as long as the transformations stay with those bounds, we say that the algorithm is *correct enough* by construction.

C. Finding a Correct-Enough Solution

The transformation of the golden, floating point model M_{golden} to the FPGA-ready fixed-point model, M_{FXP}, is an exercise in balancing two constraints: FPGA resources and algorithm correctness. As we introduce more error into the algorithm, it generally uses less resources. For many of the applications that our users are concerned with, error is measured in decibels of signal-to-noise ratio, or SNR. SNR is a comparison of the amount of good data (i.e. signal) to the interference, or noise.

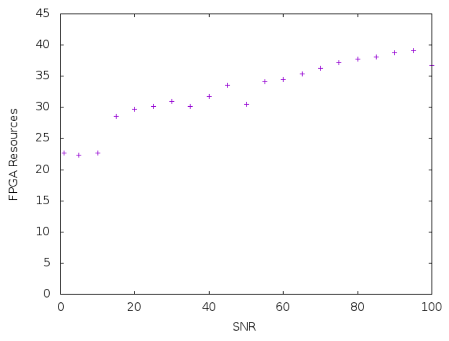


Fig. 3. Approximate Pareto front of error vs. resources for a real algorithmic model in our environment

[1]A separate set of transforms is necessary to take a high-level model in fixed point to an implementation that can actually synthesize on FPGA. While our tool does make this process easier than many traditional approaches, that benefit, and the techniques that went in to the design of our FPGA compiler, are beyond the scope of this paper.

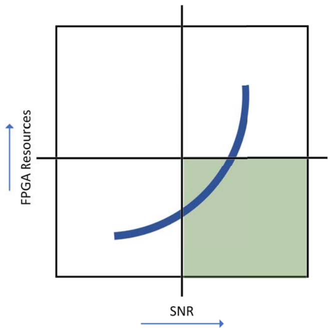


Fig. 4. Pareto front of error vs. resources and the acceptable region for final FPGA implementations

Figure 3 shows the approximate Pareto front of a manual exploration of different fixed-point types and the FPGA resources used for a basic IIR filter algorithm. On the x-axis we increase the SNR we can tolerate in our algorithm. We then create an FPGA implementation of the design using the fixed point types that our tool suggests. On the y-axis, we record the percentage of one type of the FPGA resources (flip-flops) that the implementation of the design uses.

Figure 4 shows a stylized Pareto front similar to the experimentally gathered one in Figure 3. The vertical and horizontal lines represent the user's limits on how many FPGA resources they can use with their algorithm implementation and the error they are willing to accept in that implementation, respectively. The shaded region in Figure 4 represents the space of solutions on that meet the requirements. The points of the curve in this region are the solutions achievable with the tool that created the point. M_{FXP} is one of these points, and the challenge is transforming original M_{golden} to a point on the Pareto front for that algorithm that lies within the acceptable region.

III. The LabVIEW NXG Solution

A. Basic Description

In the LabVIEW NXG FPGA module, we provide a tool that creates T_{FXP1} for the algorithm designer *automatically*. We can do this because our models are executable during the entire transformation process[2]. We then guide the algorithm developer along the process of coming up with and applying the transformations $T_{FXP2}...TFXPn$.

Our solution is based on profiling the numeric outputs of each operation in M_{golden}. The designer gives us an error constraint in terms of signal-to-noise ratio (SNR) and provides test data that they consider representative of the environment to which the algorithm will be deployed in hardware. The user provides the test data as an executable testbench model.

Not all executions are profiled. We allow the user to determine when and if they want a model to be profiled. This user-facing option allows us to take advantage of our executable models when we need to gather data, but not incur a constant overhead during all executions.

[2]We provide a cycle-accurate FPGA simulator, as well as instant feedback for whether code is synthesizable.

As we execute each mathematical operation in the profiled algorithm, we record the corresponding output values for these operations. After all the test data have been executed we analyze the resulting profiled data to determine a number of integer bits and fractional bits that can represent all the output values within the given error constraint.

B. Transform Composition

The set of transforms T is composed of a series of transforms $T_{FXP1} \dots T_{FXPn}$. Each one of these transforms may do more than one thing to the model being transformed. Given a model with k operations that can affect the precision of the overall result, we say that $T_{FXPi} = \{t_1, t_2, \dots t_k\}$.

For a $t_j \in T_{FXPi}$, this transform may either change the output type of the operation, or do nothing, which we signify with $\varnothing$. If a transform does change the type, we represent that differently. Some examples of transforms that change the type are, along with their representations, $[DBL \rightarrow (3.5)]$ (taking a floating-point output type to the fixed-point output type with three integer bits and five fractional bits) or $[(3.5) \rightarrow (3.8)]$ (adding three fractional bits to increase precision of this particular operator).

Our generated T_{FXP1} has a transform for each operation in the model. Our analysis produces a t_i for each operation i, as each operation will produce a set of output values when we run the testbench. In Figure 1, if that multiply was the only operation in our golden model, T_{FXP1} would just have one component. If going from the top floating-point version to the bottom version represented our initial conversion, we would say that $T_{FXP1} = \{[DBL \rightarrow (1.15)]\}$.

C. Mathematics of Creating T_{FXP1}

As previously mentioned, we record the output values for each operator in our model during profiling. To determine T_{FXP1}, we first start with the constraint that the user gives us in SNR.

For each operator i in our model, we have a vector of outputs $\vec{O_i}$ of size n, where n is the number of times the operator ran when we profiled M_{golden} using the testbench. For each value $o_j \in \vec{O_i}$, we determine the amount of bits necessary to capture the integer part with no overflow [3]. We refer to this number of integer bits as $IntMax$.

To determine the fractional part of the type suggestion we first have to determine the SNR of each possible type choice. This set of choices is bounded as we are starting with double-precision floating point, which uses at most 64 bits to represent the fractional bits. To determine SNR we first calculate the vector that is the golden values converted to a particular fixed-point type. If we are currently evaluating $t = [DBL \rightarrow (I.F)]$, where I is a number of integer bits and F is a number of fractional bits, we represent the converted vector as $FXP_{(I.F)}(\vec{O_i})$. We then calculate the vector norm of the golden values minus the golden values converted to the particular fixed point type which gives us the noise:

[3]Overflow is defined as when the integer part of the number needs more bits than we are using to represent the integer part.

$$\|\vec{O_i} - FXP_{I.F}(\vec{O_i})\|$$

Where the vector norm is defined as:

$$\sqrt{\sum_{j=1}^{n} (o_j - FXP_{I.F}(o_j))^2}$$

We then calculate the vector norm of the golden outputs O_i themselves, giving us the signal:

$$\|\vec{O_i}\| = \sqrt{\sum_{j=1}^{n} o_j^2}$$

Finally, we calculate the SNR in decibels:

$$SNR_i = \log_{10}\left(\frac{\|\vec{O_i}\|}{\|\vec{O_i} - FXP_{I.F}(\vec{O_i})\|}\right)$$

Given these procedures to calculate the SNR, we can find our suggested fixed-point type. We start with the number of fractional bits required by the most precise of the output values. We then step down one bit at a time, calculating the SNR for each type choice. Once we hit an SNR that is below the constraint, we add 1 bit to the number of fractional bits, thus guaranteeing that the SNR this type will produce is above the constraint. We refer to this number of fractional bits as $FracMin$.

Then, for each operator i, we can create a transform suggestion $t_i = [DBL \rightarrow (IntMax.FracMin)]$. We create these type suggestions t_i that make up T_{FXP1} based on the SNRs that we calculate, but note that these SNRs are local. They are based purely on the SNR we obtain by applying the transform to the profiled values—they do not take into account the error that propagates through the model during execution.

D. Error Propagation and $T_{FXP2} \dots T_{FXPn}$

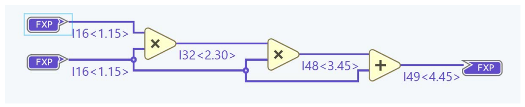


Fig. 5. Bit-width growth to preserve precision.

Figure 5 shows a simple algorithmic model of two multiplies and an add. While these are all fixed-point operations, the settings on the operators are such that the output type will grow according to the rules of fixed-point arithmetic such that no error or overflow are introduced. For multiplies, the number of fractional bits in the product, F_{Prod}, is defined by $F_{Prod} = F_{Input1} + F_{Input2}$. The number of integer bits in the product, I_{Prod}, is defined by $I_{Prod} = I_{Input1} + I_{Input2}$. The add operation introduces less error, so the number of integer bits required, I_{Sum}, is defined as $I_{Sum} = \max(I_{Input1}, I_{Input2}) + 1$. The fractional bits required, F_{Sum}, is defined as $F_{Sum} = \max(F_{Input1}, F_{Input2})$.

In this version of the model, no error is introduced, but that clearly comes with a cost. In LabVIEW NXG, for example, we do not allow fixed-point types that use greater than a total of 64 bits between the integer and fractional part. It is obvious to see that with just a few multiplies one will not be able to have a set of fixed-point types for the operators in their model that would introduce no error.

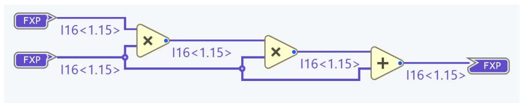


Fig. 6. Here, we restrict the output type at each operator.

Figure 6 shows a version of the model where we overwrite the output of each operator to be a fixed-point type with 1 integer bit and 15 fractional bits. As data moves from left to write, the amount of error introduced is compounded. A simplistic way to see this point is noticing the difference in the number of fractional bits on the output of a given operator in Figure5 versus the number of fractional bits for that corresponding operator in Figure 6. The difference increases dramatically. However, this introduction of error may very well be acceptable to the user as long as the overall error is below their threshold (i.e. an SNR above the constraint they give us), making the final design *correct enough* by construction.

Given that the models M_{golden} and $M_{FXP1} \ldots M_{FXPn}$ are executable, we can continue to execute the fixed-point models during the conversion process. Each time the user executes one of the fixed-point models, we again collect a vector of output values at each operator $i : \vec{O}_i$. From this point on, we refer to the golden vector of values for each operator i as $\vec{O}_i^{golden}$.

On these later executions, we are able to calculate not a local SNR but a true SNR that takes into account the error propagation. To calculate the true SNR, we always compare the profiled data to the original golden data from the floating point run. So the SNR for a given operator j is defined as:

$$SNR_i = \log_{10}\left(\frac{\|\vec{O}_i^{golden}\|}{\|\vec{O}_i^{golden} - \vec{O}_i\|}\right)$$

Performing the transformations $T_{FXP2} \ldots T_{FXPn}$ may change the behavior of the program, other than just introducing imprecision. For example, a particular branching instruction may branch separately due to a compare having a different value. We may end up getting only m profiled values at an operator that previously had n profiled values in M_{golden}. We take a simple approach of truncating the longer vector. We find that this generally produces either an acceptable SNR, or it introduces enough error that it becomes a hot spot of imprecision, and thus noticeable to the user.

As the user continues to run the testbench to get the real SNR values that take error propagation into account, their testbench should pass or fail their design requirements. It is this constant feedback from their testbench, plus the information that our tool gives them, that allows them to continue to move in the correct direction along the Pareto front of possible fixed-point type choices.

IV. Fixed Point Conversion Workflow

A user starts with the golden model of their algorithm, written in some high-level tool. For purposes of explaining the workflow in our LabVIEW NXG, we are showing a golden model written in G, one of the dataflow models of computation present in LabVIEW NXG. Figure 7 shows the golden model of a simple IIR Filter written in G.

The user creates a testbench in G that runs appropriate test data through the golden model. The user would normally create a testbench such that it is easy to determine whether the execution of the golden model is within the acceptable correctness bounds. It is a reasonable assumption that an algorithm expert would be familiar with what a thorough testbench would be. They know their algorithm and domain, and often the requirements of the algorithm are set forth by the customer or funding agency.

We provide a simple mechanism for the user to duplicate their golden model in its entirety. Once the user duplicates the model, the duplicate can be integrated into the testbench such that the output from both models, golden and duplicate, can be compared as the user moves from M_{golden} to M_{FXP}. Once they have their duplicate of the golden model, they can start to profile M_{golden}.

The user tells LabVIEW NXG that they want to profile the execution of the duplicate of their golden model. This command forces LabVIEW NXG to record all the data that flows through the inputs and outputs of every operation in their model. Because our models are executable, there is no extra code that the user has to write to achieve this profiling. The user then runs the testbench with the profiling enabled, and the tool analyzes the profile data to determine the transform T_{FXP1} using the methods we explain in Section III. The tool displays these suggestions in a table at the bottom of the environment. The set of suggestions for the M_{golden} from Figure 7 is shown in Figure 8.

Each row represents a different operation in the golden model. The name of the operations are in the second column. In the third column the tool lists the current type as well as the initial FXP suggestion. The fourth column is the local SNR at each operation if the user chooses to use the suggested FXP type. The fourth and fifth columns show the Overflow and Underflow [4], respectively, that those type choices would cause at that corresponding operation.

At the top of the table there is a field for the target SNR. The user can change this value (it defaults to 10), and see the suggested types change. As they increase the SNR, generally the amount of bits increases. Similarly, as they reduce the SNR the amount of bits will reduce. The user can pick another strategy, bit-width, that will make all the suggestions the same total width, and attempt to change the integer and fractional

[4]Underflow is the percentage of samples that are a number smaller than the smallest number that the suggested amount of fractional bits can represent.

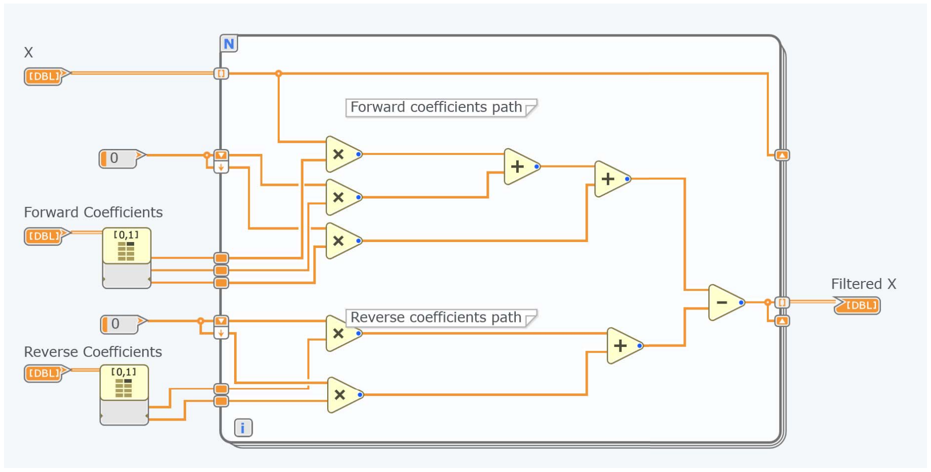
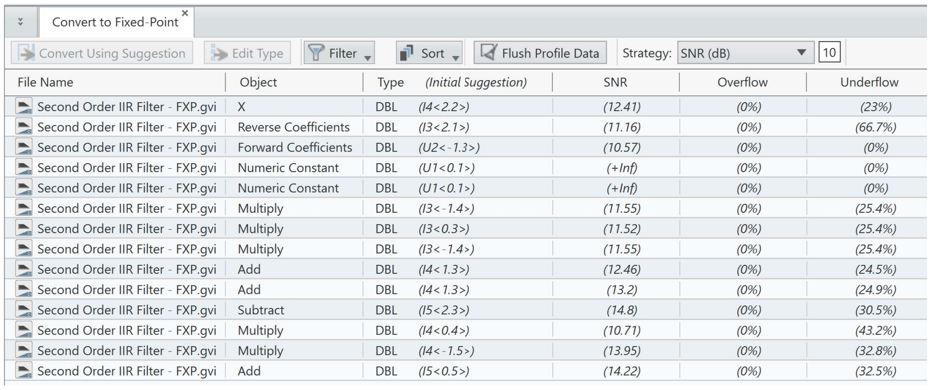


Fig. 7. A golden model of an algorithm written in LabVIEW Communication Systems Design Suite.



Strategy: SNR (dB) 10

File Name	Object	Type	(Initial Suggestion)	SNR	Overflow	Underflow
Second Order IIR Filter - FXP.gvi	X	DBL	(I4<2.2>)	(12.41)	(0%)	(23%)
Second Order IIR Filter - FXP.gvi	Reverse Coefficients	DBL	(I3<2.1>)	(11.16)	(0%)	(66.7%)
Second Order IIR Filter - FXP.gvi	Forward Coefficients	DBL	(U2<-1.3>)	(10.57)	(0%)	(0%)
Second Order IIR Filter - FXP.gvi	Numeric Constant	DBL	(U1<0.1>)	(+Inf)	(0%)	(0%)
Second Order IIR Filter - FXP.gvi	Numeric Constant	DBL	(U1<0.1>)	(+Inf)	(0%)	(0%)
Second Order IIR Filter - FXP.gvi	Multiply	DBL	(I3<-1.4>)	(11.55)	(0%)	(25.4%)
Second Order IIR Filter - FXP.gvi	Multiply	DBL	(I3<0.3>)	(11.52)	(0%)	(25.4%)
Second Order IIR Filter - FXP.gvi	Multiply	DBL	(I3<-1.4>)	(11.55)	(0%)	(25.4%)
Second Order IIR Filter - FXP.gvi	Add	DBL	(I4<1.3>)	(12.46)	(0%)	(24.5%)
Second Order IIR Filter - FXP.gvi	Add	DBL	(I4<1.3>)	(13.2)	(0%)	(24.9%)
Second Order IIR Filter - FXP.gvi	Subtract	DBL	(I5<2.3>)	(14.8)	(0%)	(30.5%)
Second Order IIR Filter - FXP.gvi	Multiply	DBL	(I4<0.4>)	(10.71)	(0%)	(43.2%)
Second Order IIR Filter - FXP.gvi	Multiply	DBL	(I4<-1.5>)	(13.95)	(0%)	(32.8%)
Second Order IIR Filter - FXP.gvi	Add	DBL	(I5<0.5>)	(14.22)	(0%)	(32.5%)

Fig. 8. A suggestion of T_{FXP1} that we provide that makes local recommendations within the given error constraint.

parts to maximize SNR within that bit-width. This box is where the user provides their constraints—the user constraint defines the acceptable error that we attempt to match.

The user would then select some number, or all, of the suggestions, and the hit the Apply button in the table. Figure 9 shows the model M_{FXP1} after the user has applied T_{FXP1} by selecting all the rows and hitting apply. Notice that the model is now annotated with the specific FXP types that each operation will output.

The suggestions that the tool gave the user were optimizing local error. However, error does propagate through the model, increasing as we limit the amount of growth in the type. To understand the actual, propagated error, the user must run their testbench again. We provide a warning to let them know that their SNR values are out-of-date. Once the user runs the testbench again, we then provide the *actual error* in the SNR column for each operation taking into account error propagation through the design. Figure 10 shows the table with actual, propagated error.

While we determined a starting point for the floating point to fixed point transformation, T_{FXP1}, we shift at this point in the workflow to guiding the user to determine the appropriate $T_{FXP2} \ldots T_{FXPn}$. One of the tools we give is to show "hot spots" of low SNR on the model itself, as seen in Figure 11. The user has requested an SNR of 10, but after a run we show that somewhere in the lower part of the model is introducing a large amount of error causing this lower path to have a low SNR. The first step to rectify this situation is to introduce extra fractional bits to allow for more precision in this path.

We give the users a tool to do that, as seen in Figure 12. This tool allows the user to select multiple nodes in their model, and add a relative number of bits to either the fractional or integer components of the output types of the selected operations. This tool allows the user to quickly increase precision along an entire hot spot in the application.

The user then reruns their testbench to see how the error has changed. They repeat this process until they are happy with the error they are seeing and the results they are getting in their testbench, creating a model that is "correct enough."

V. Related Work

LabVIEW [10] is a commercial, graphical dataflow programming environment. It allows engineers and scientists to program in a model of computation that closely matches the

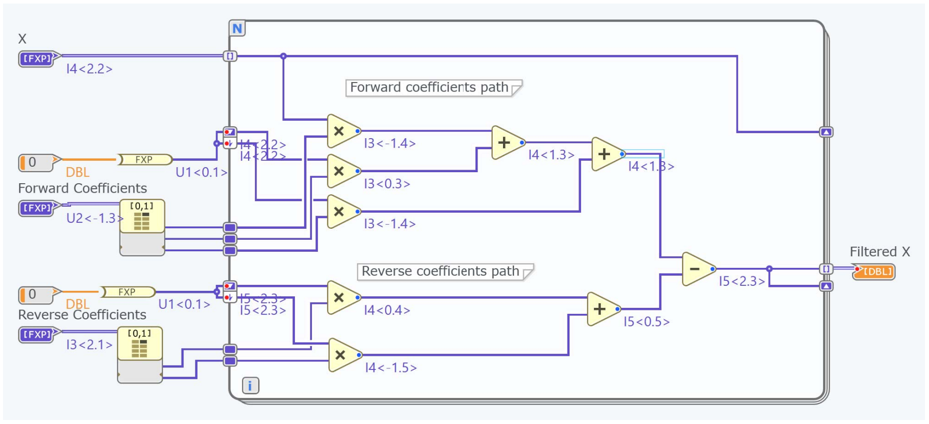
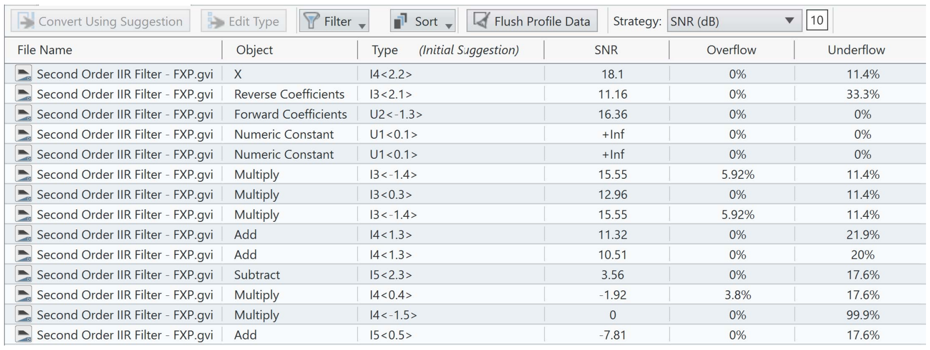


Fig. 9. The executable model M_{FXP1} after the user applies our suggestion for T_{FXP1}.



Strategy: SNR (dB) 10

File Name	Object	Type *(Initial Suggestion)*	SNR	Overflow	Underflow
Second Order IIR Filter - FXP.gvi	X	I4<2.2>	18.1	0%	11.4%
Second Order IIR Filter - FXP.gvi	Reverse Coefficients	I3<2.1>	11.16	0%	33.3%
Second Order IIR Filter - FXP.gvi	Forward Coefficients	U2<-1.3>	16.36	0%	0%
Second Order IIR Filter - FXP.gvi	Numeric Constant	U1<0.1>	+Inf	0%	0%
Second Order IIR Filter - FXP.gvi	Numeric Constant	U1<0.1>	+Inf	0%	0%
Second Order IIR Filter - FXP.gvi	Multiply	I3<-1.4>	15.55	5.92%	11.4%
Second Order IIR Filter - FXP.gvi	Multiply	I3<0.3>	12.96	0%	11.4%
Second Order IIR Filter - FXP.gvi	Multiply	I3<-1.4>	15.55	5.92%	11.4%
Second Order IIR Filter - FXP.gvi	Add	I4<1.3>	11.32	0%	21.9%
Second Order IIR Filter - FXP.gvi	Add	I4<1.3>	10.51	0%	20%
Second Order IIR Filter - FXP.gvi	Subtract	I5<2.3>	3.56	0%	17.6%
Second Order IIR Filter - FXP.gvi	Multiply	I4<0.4>	-1.92	3.8%	17.6%
Second Order IIR Filter - FXP.gvi	Multiply	I4<-1.5>	0	0%	99.9%
Second Order IIR Filter - FXP.gvi	Add	I5<0.5>	-7.81	0%	17.6%

Fig. 10. The error of each operation after applying the initial suggestions, T_{FXP1}, and rerunning the test bench.

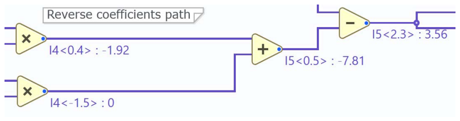
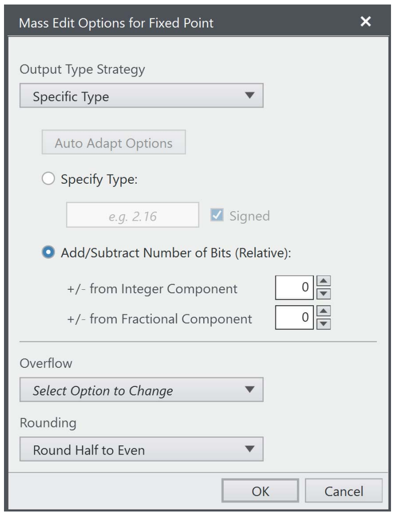


Fig. 11. Nodes in the model that have low SNR pointing the user to a possible hot spot of precision loss.


Mass Edit Options for Fixed Point

Output Type Strategy
Specific Type

Auto Adapt Options

Specify Type:
e.g. 2.16 Signed

Add/Subtract Number of Bits (Relative):
+/- from Integer Component 0
+/- from Fractional Component 0

Overflow
Select Option to Change

Rounding
Round Half to Even

Fig. 12. The dialog for our multinode type transform.

models of how they think and design. It provides an executable model, G, so that the graphical diagrams are runnable at all points of development.

LabVIEW NXG [11] is a new version of LabVIEW with a modern editor, improved design, and many usability improvements. The LabVIEW NXG FPGA [2] module provides the support to write executable models for FPGAs. The LabVIEW NXG FPGA module is what contains our float-to-fixed tool.

There are many other graphical programming environments: Simulink [9], Ptolemy [12], GNU Radio [13], and Blender [14] are some of the more widely known from both academia and industry, but there are many more. G is specifically targeted at engineers and scientists, and we craft the metamodel and available primitives with these applications in mind.

Dataflow programming has a long history [15]. When National Instruments released LabVIEW in 1987, one of the main innovations to visual and dataflow programming was the notion of including program control structures. This decision mixed concepts from traditional, imperative text-based programming with dataflow. LabVIEW also supports object-oriented programming with LabVIEW Classes.

The concept of executable models is not new, as Simulink, Ptolemy, and Executable UML [16] (among others) have all been around for a while. Ciccozzi et. al provide a survey of the current research in executable UML [17]. Executable models is an active area of research, too large to properly survey here. Nothing that LabVIEW provides in terms of execution is particularly novel from a modeling perspective. However, it was utilizing the executable nature of our models that allows us to quickly and easily use actual test data to determine the initial transformation and then guide users through applying the rest of the fixed-point transformations.

Riché et. al discussed using MDE-techniques to apply expert-knowledge transforms to dataflow applications [18]. They then expanded this work to transforming series of transformations, i.e. pushouts, on dataflow applications [19]. The work was continued by Gonçalves et. al in the creation of the ReFlo tool [20].

Oh et. al's work on pareto front optimizations in product lines [21] is a much more thorough look at pareto front optimizations than we present in this work. They use random jumps to make sure that the optimizations do not get stuck in local minima. The future work planned for our tool would build on similar concepts as in Oh et. al's paper.

Floating-point to fixed-point conversion is an active area of research [22]–[25]. However, most work focuses on total automatic conversion of the model. These tools use a variety of techniques in their effort to completely convert the algorithm to fixed point. The novelty in our approach is using simple techniques to get a user started on their conversion to fixed point, and then give them tools that guide them along the rest of the way. By using this approach, we allow for a greater range of algorithmic models to be converted than the fully automatic tools. But obviously, we do still depend on the algorithm experts to create quality testbenches that accurately test the overall correctness of their algorithmic model. We also are not claiming our contribution is that our underlying conversion mathematics are new–we feel that we are simply presenting these ideas in an easy-to-use way.

VI. Conclusion

We present a novel tool as part of LabVIEW NXG FPGA Module that uses model-driven techniques to guide an algorithm expert to a fixed-point, FPGA-executable version of their original algorithm model. The tool works on executable models built in G, the graphical dataflow model of computation in LabVIEW NXG. Our tool helps ease the pain of transforming floating-point, golden algorithm models to fixed point.

Part of the novelty in our solution is we use simple techniques to generate an initial floating-point to fixed-point transform. Our executable models enable this simplicity. We then guide the user along the rest of the way with information about the key sources of error within their model. Tools within our environment make updating types easy.

Our focus is to get people moving in the right direction, not necessarily taking an "automatic transformation or nothing" approach. This guided approach widens the set of algorithms that our tool can help algorithm experts convert.

References

[1] S. D. Brown, R. J. Francis, J. Rose, and Z. G. Vranesic, *Field-Programmable Gate Arrays*, ser. The Springer International Series in Engineering and Computer Science. Springer US, 1992, vol. 180.

[2] N. Instruments, "Labview nxg fpga module," http://www.ni.com/en-us/support/downloads/software-products/download.labview-nxg-fpga-module.html#305496, 2018.

[3] "Ieee standard vhdl language reference manual," *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pp. c1–626, Jan 2009.

[4] "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 1–560, 2006.

[5] 3GPP, "Submission of initial 5g description for imt-2020," http://www.3gpp.org/NEWS-EVENTS/3GPP-NEWS/1937-5G_DESCRIPTION, Jan 2018.

[6] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1–70, Aug 2008.

[7] B. Widrow, "Statistical analysis of amplitude-quantized sampled-data systems," *Transactions of the American Institute of Electrical Engineers, Part II: Applications and Industry*, vol. 79, no. 6, pp. 555–568, Jan 1961.

[8] T. M. Inc., "Matlab," https://www.mathworks.com/products/matlab.html, 2018.

[9] ——, "Simulink," https://www.mathworks.com/products/simulink.html, 2018.

[10] N. I. Corporation, "Labview," http://ni.com/labview, 2018.

[11] N. Instruments, "Labview nxg," https://www.ni.com/en-us/shop/labview/labview-nxg.html, 2017.

[12] *System Design, Modeling, and Simulation using Ptolemy II.* Ptolemy.org, 2014. [Online]. Available: http://ptolemy.org/books/Systems

[13] "Gnu radio," https://gnuradio.org/, 2018.

[14] "Blender," https://www.blender.org/, 2018.

[15] W. M. Johnston, J. R. P. Hanna, and R. J. Millar, "Advances in dataflow programming languages," *ACM Computing Surveys*, vol. 36, no. 1, pp. 1–34, March 2004.

[16] S. J. Mellor and M. J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley Professional, 2002.

[17] F. Ciccozzi, I. Malavolta, and B. Selic, "Execution of uml models: a systematic review of research and practice," *Software & Systems Modeling*, Apr 2018. [Online]. Available: https://doi.org/10.1007/s10270-018-0675-4

[18] T. L. Riché, H. M. Vin, and D. Batory, "Transformation-based parallelization of request-processing applications," in *Proceedings of the 13th International Conference on Model-Driven Engineering, Languages, and Systems (MODELS)*, ser. LNCS. Springer, 2010, pp. 2–16.

[19] T. L. Riché, R. Gonçalves, B. Marker, and D. Batory, "Pushouts in software architecture design," in *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, ser. GPCE '12. New York, NY, USA: ACM, 2012, pp. 84–92. [Online]. Available: http://doi.acm.org/10.1145/2371401.2371415

[20] R. C. Gonçalves, D. Batory, J. L. Sobral, and T. L. Riché, "From software extensions to product lines of dataflow programs," *Software & Systems Modeling*, vol. 16, no. 4, pp. 929–947, Oct 2017. [Online]. Available: https://doi.org/10.1007/s10270-015-0495-8

[21] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding near-optimal configurations in product lines by random sampling," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017. New York, NY, USA: ACM, 2017, pp. 61–71. [Online]. Available: http://doi.acm.org/10.1145/3106237.3106273

[22] S. Lee and A. Gerstlauer, "Fine grain precision scaling for datapath approximations in digital signal processing systems," in *VLSI-SoC: At the Crossroads of Emerging Trends*. Cham: Springer International Publishing, 2015, pp. 119–143.

[23] A. Pedram, R. A. van de Geijn, and A. Gerstlauer, "Codesign tradeoffs for high-performance, low-power linear algebra architectures," *IEEE Transactions on Computers*, vol. 61, no. 12, pp. 1724–1736, Dec 2012.

[24] P. Belanovic and M. Rupp, "Automated floating-point to fixed-point conversion with the fixify environment," in *16th IEEE International Workshop on Rapid System Prototyping (RSP'05)*, June 2005, pp. 172–178.

[25] L. S. Rosa, C. F. M. Toledo, and V. Bonato, "Accelerating floating-point to fixed-point data type conversion with evolutionary algorithms," *Electronics Letters*, vol. 51, no. 3, pp. 244–246, 2015.