Précis: A Design-Time Precision Analysis Tool

Mark L. Chang and Scott Hauck
Department of Electrical Engineering
University of Washington, Seattle, WA
{mchang,hauck}@ee.washington.edu

Abstract

Currently, few tools exist to aid the FPGA developer in translating an algorithm designed for a general-purpose-processor into one that is precision-optimized for FPGAs. This task requires extensive knowledge of both the algorithm and the target hardware. We present a design-time tool, Précis, which assists the developer in analyzing the precision requirements of algorithms specified in MATLAB. Through the combined use of simulation, user input, and program analysis, we demonstrate a methodology for precision analysis that can aid the developer in focusing their manual precision optimization efforts.

1. Introduction

One of the most difficult tasks in implementing an algorithm in an FPGA-like substrate is dealing with precision issues. Typical general-purpose processor concepts such as *word size* and *data type* are no longer valid in the FPGA world, which is dominated by finer-grained computational structures, such as look-up tables. Instead, the designer must use and implement bit-precise data paths.

More specifically, in a general-purpose processor, algorithm designers can typically choose from a predefined set of variable types that have a fixed word length. Examples of these predefined types are the `C` data types such as `char`, `int`, `float`, `double`. These data types correspond to specific memory storage sizes, and subsequently, into different ways of handling operations upon these memory locations within the microprocessor. Much of the work of padding, word-boundary alignment, and operation selection is hidden from the programmer by compilers and assemblers, which make the use of one data type equally easy as another.

In contrast, an FPGA does not have predefined data widths for its data path. Instead, designers must provide all the structures necessary to handle operations on different data widths and types. Therefore, it is paramount that FPGA designers implement their algorithms such that they utilize resources efficiently and accurately. Too many bits allocated to a particular operation is wasteful, while too few can result in erroneous output.

The difficulty is in the translation of an initial algorithm into one that is precision-optimized for FPGAs. This task requires extensive knowledge of both the algorithm and the target hardware. Unfortunately, there are few tools that aid the would-be FPGA developer in this translation. In this paper, we discuss our work in filling that gap by introducing a developer-oriented tool for the design-time analysis of the impact of precision on algorithm implementation.

2. Background

Currently, the typical tool flow for development of an FPGA-targeted algorithm is as shown in Figure 1.

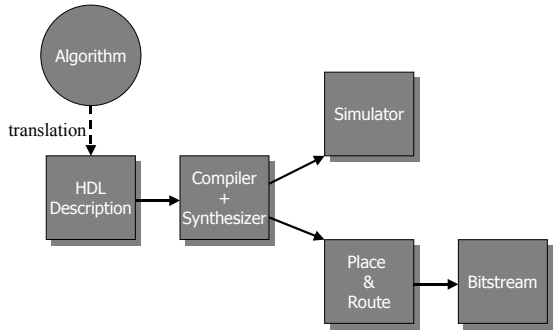

Figure 1. Typical tool flow for implementing a high level language specified algorithm on an FPGA.

At the head of the development chain is the algorithm. Often, the algorithm under consideration has been implemented in some high-level language, such as MATLAB, C, or Java, targeted to run on a general purpose processor, such as a workstation or desktop personal computer. The most compelling reason to utilize a high level language running on a workstation is that it provides infinite flexibility and a comfortable, rich environment in which to rapidly prototype algorithms. Of course, the reason one would convert this algorithm into a hardware implementation is to gain considerable

advantages in terms of speed, size, and power.

This tool flow requires the developer to first convert a software prototyped algorithm into a hardware description. From this hardware description language (HDL) specification, various stages and intermediate tools are used to perform simulation and generate target bitstreams, which are then executed on reconfigurable logic. As mentioned earlier, one of the more difficult steps in implementing the algorithm in hardware is highlighted in Figure 1 with a dashed arrow – the conversion from a high-level software language, such as C, Java, or MATLAB, into an HDL description.

A simple conversion without precision analysis would most likely yield an unreasonably large hardware implementation. For example, by blindly choosing a fixed 32-bit data path throughout the system, the developer may encounter two problems: wasted area and incorrect results. The former arises when the actual data the algorithm operates upon does not require the full 32-bit data path. In this case, much of the area occupied by the oversized data path could be pruned. There are several benefits to area reduction of a hardware implementation: reduced power consumption, reduced critical path delay, and the increased probability of parallelism by freeing up more room on the device to perform other operations simultaneously. On the other hand, the latter case occurs when the algorithm actually requires more precision for some data sets than the 32-bit data path provides. In this case, the results obtained from the algorithm could potentially be incorrect due to unchecked overflow or underflow conditions.

Therefore, within the HDL description, it is important that the developer determine more accurate bounds on the data path. Typically, this involves running a software implementation of the algorithm with representative data sets and performing manual fixed-point analysis. At the very least, this requires the re-engineering of the software implementation to record the ranges of variables throughout the algorithm. From these results, the developer could infer candidate bit-widths for their hardware implementation. Even so, these methods are tedious and often error-prone.

Unfortunately, while many of the other stages of hardware development have well-developed tools to help automate difficult tasks, few tools can automate HDL generation from a processor-oriented higher level language specification. And while there are C-to-Verilog[1] and C-to-VHDL[2] tools in existence, they do not offer such "designer aids" that would help with precision analysis of existing algorithms implemented in a high level language.

3. Précis

In order to fill this void in hardware development tools, we are developing *Précis*, a design-time precision analysis tool. Précis utilizes MATLAB as an input specification for algorithms and is designed to interact with the developer in order to assist them in making the best choices regarding data path precision. Currently, Précis aids the developer by providing a constraint propagation engine, simulation support, range finding capabilities, and performing precision slack analysis.

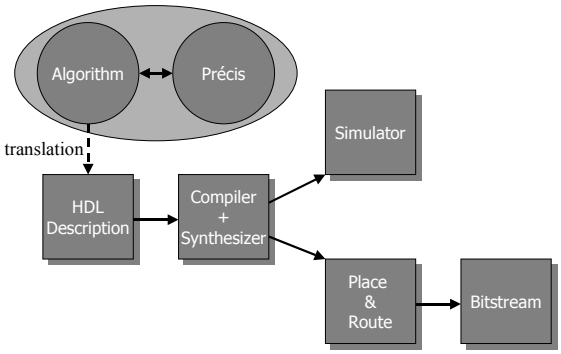

Figure 2. Précis' role in the tool chain.

Précis is designed to complement the existing tool flow in the manner shown in Figure 2. Précis is not meant to be an HDL generator, a MATLAB-to-HDL converter, or an optimizing compiler of any sort. Instead, it is meant to provide a convenient way for the user to interact with the algorithm under consideration. Our goal is for the knowledgeable user, after interacting with our tool, to have a much clearer idea of the precision requirements of their data path. It is our belief that the developer of the algorithm, with suitable software assistance, can perform much better precision analysis and optimization than a fully automated tool could ever achieve. In the following sections, we describe in more detail the constituent parts of Précis.

3.1. MATCH front-end

The front-end of Précis comes from Northwestern University in the form of a modified MATCH compiler[3,4]. The MATCH compiler understands a subset of the MATLAB language and can transform it into efficient implementations on FPGAs, DSPs, and embedded CPUs. It is used here primarily as a pre-processor to parse MATLAB codes. The MATCH compiler was chosen as the basis for the MATLAB code parsing because no official grammar is publicly available for MATLAB. We are not constrained to using the MATCH compiler, though, as our tool may be updated to accommodate an alternate MATLAB-aware parser.

MATLAB was chosen as the target high level language because the researchers involved in this work also contribute to the MATCH project at Northwestern University. From this work, it has become clear that MATLAB is a strong favorite for algorithm prototyping

and exploration, especially among scientists that might have little to no hardware design expertise. With the proliferation of reconfigurable co-processor boards capable of providing great speedups to many classes of algorithms, it would be advantageous to provide tools to help these same scientists target their MATLAB algorithms to FPGAs. Précis can be used both by developers prototyping in MATLAB before hand converting to an HDL, or to develop pragmas (designer hints) for MATCH's automatic compilation.

The MATCH compiler remains a work in progress and is currently being marketed by AccelChip[5]. For our purposes, we have modified the base MATCH compiler to generate a non-hierarchical (flattened) representation of parsed MATLAB code from its internal abstract syntax tree. This representation is then read into the main *Précis* tool for display and user interaction.

3.2. Précis application

The main Précis application is written in Java, in part, due to its relative platform independence and ease of graphical user interface creation. Précis takes the parsed MATLAB code output generated from the MATCH compiler and displays a GUI that formats the code into a tree-like representation of statements and expressions. An example of the GUI in operation is shown in Figure 3. The left half of the interface is the tree representation of the MATLAB code. The user may click on any node and, depending on the node type, receive more information in the right panel. The right panel displayed in the figure is an example of the entry dialog that allows the user to specify fixed-point precision parameters, such as range and type of truncation. With this graphical display the user can then perform the various tasks described in the following sections.

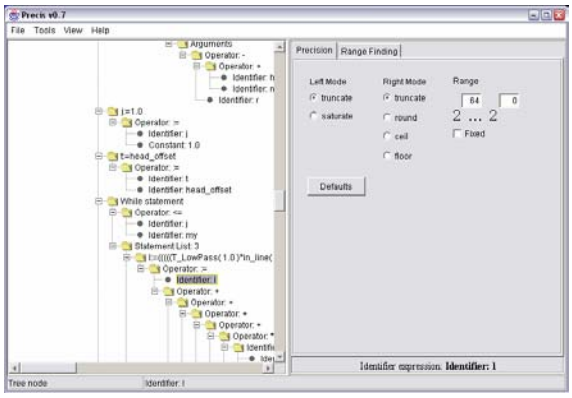

Figure 3. Screen capture of the Précis GUI.

3.3. Propagation engine

A core component of the Précis tool is a constraint propagation engine. The propagation engine simulates the effects of using fixed-point numbers and fixed-point math in hardware. This is done by allowing the user to (optionally) constrain variables to a specific precision by specifying the bit positions of the most significant bit (MSB) and least significant bit (LSB). Variables that are not manually constrained begin with a default width of 64 bits. Typically, a user should be able to provide constraints easily for at least the circuit inputs and outputs.

The propagation engine traverses the expression tree and determines the resultant ranges of each operator expression from its child expressions. This is done by implementing a set of rules governing the change in resultant range that depend upon the input operand(s) range(s) and the type of operation being performed. For example, in the statement `a=b+c`, if `b` and `c` are both constrained by the user to a range of `2^15` to `2^0`, 16 bits, the resulting output range of `a` would have a range of `2^16` to `2^0`, 17 bits, as an addition conservatively requires one additional high order bit for the result in the case of a carry-out from the highest order bit. Similar rules apply for all supported operations.

The propagation engine works in this fashion across all statements of the program, recursively computing the precision for all expressions in the program. This form of propagation is often referred to as value-range propagation. One shortcoming of the currently implemented propagation engine is that it does not handle loop carried variables or conditional branches. This is to be rectified in later revisions of the tool. A more complete study of propagation and its effects upon hardware synthesis can be found in [6]. We plan to continue development of our own propagation tool to a similar extent in the near future.

An example of forward and backward propagation is depicted in Figure 4

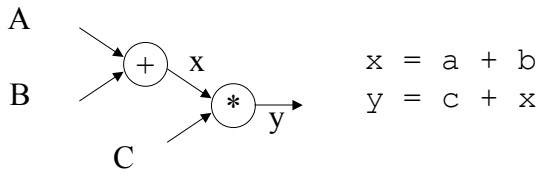

Figure 4. Simple propagation example.

In this trivial example, assume the user sets all input values (a, b, c) to utilize the bits [15,0], i.e. have a range from `2^16-1` to `0`. Forward propagation would result in `x` having a bit range of [16, 0] and `c` having a range of [31, 0]. If, after further manual analysis, the user notes that the output from these statements should be constrained to a range of [10, 0], backwards propagation following forward propagation will constrain the inputs (`c` and `x`) of the multiplication to [10, 0] as well. Propagating yet further, this constrains the input variables `a` and `b` to

the range [10, 0] as well. Obviously, these are very conservative propagation values. Knowing strict values for the variables would increase our accuracy, as can be shown in [6].

The propagation engine can be used to get a quick estimate of the growth rate of variables through the algorithm. This is done by constraining the precision of input variables and a few operators and performing the propagation. This will allow the user to see a conservative estimate of how the input bit width affects the size of operations down stream.

While the propagation engine will provide some information as to the effects of fixed-point operations on the resultant data, it is at best a conservative estimate. It would be appropriate to consider the bit widths determined from the propagation engine to be worst-case results, or in other words, an upper bound. This upper bound will become useful in further analysis phases of Précis.

3.4. Simulation support

As previously mentioned, a typical step in precision analysis is the actual running of the algorithm in a fixed-point environment. Précis can automatically generate annotated MATLAB code to aid in fixed-point simulation of the user's algorithm. The user simply selects variables to constrain and requests that MATLAB simulation code be generated. The code generated by the tool includes calls to MATLAB helper functions that we developed to simulate a fixed-point environment. The simulation flow is shown in Figure 5.

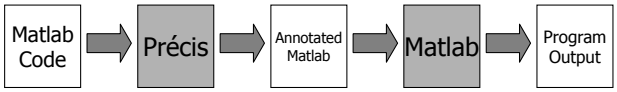

Figure 5. Code generation for simulation.

In particular, a MATLAB support routine, "`fixp`" was developed to simulate a fixed-point environment. Its declaration is `fixp(x,m,n,lmode,rmode)`, where '`x`' denotes the signal to be truncated to '`(m-n+1)`' bits in width. Specifically, '`m`' denotes the MSB bit position and '`n`' the LSB bit position, inclusively, with negative values representing positions to the right of the decimal point. The remaining two parameters, '`lmode`' and '`rmode`' specify the method desired to deal with overflow at the MSB and LSB portions of the variable, respectively. These modes correspond to different methods of hardware implementation. Possible choices for '`lmode`' are `sat` and `trunc`—saturation to `2^(MSB+1)-1` and truncation of all bits above the MSB position, respectively. For the LSB side of the variable, there are four modes, `round`, `trunc`, `ceil`, and `floor`. `Round` rounds the result to the nearest integer, `trunc` truncates all bits below the LSB position, `ceil` rounds up to the next integer level, and `floor` rounds down to the next lower integer level. These modes correspond exactly to the MATLAB functions with the exception of `trunc`, and thus behave as documented by Mathworks. `Trunc` is accomplished through the modulo operation. An example of output generated for simulation is shown in Figure 6.

MATLAB Input	Annotated MATLAB
`a = 1;`	`a=1;`
`b = 2;`	`b=2;`
`c = 3;`	`c=3;`
`d = (a+(b*c));`	`d=(fixpp(a,12,3,'trunc','trunc')+ (b*c));`

Figure 6. Sample output generated for simulation, with the range of a variable constrained.

After the user has constrained the variables of interest and indicated the mechanism by which to control overflow of bits beyond the constrained precision, Précis can generate annotated MATLAB. The user can then run the generated MATLAB code with real data sets. The purpose of these simulations is to determine the effects of constraining variables on the correctness of the implementation. Not only might the eventual output be erroneous, but the algorithm may also fail to operate entirely due to the effects of precision constraints.

If the user finds the algorithm's output to be acceptable, they might consider constraining additional key variables, thereby further reducing the eventual size of the hardware circuit. On the other hand, if the output generates unusable results, the user knows then that their constraints were too aggressive and that they should increase the precision of some of the constrained variables. Note that it is typically not sufficient to merely test whether the fixed precision results are identical to the unconstrained precision results, since this is too restrictive. In situations such as image processing, lossy compression, and speech processing, users may be willing to trade some result quality for a more efficient hardware implementation. Précis, by being a designer assistance tool, allows the designer to create their own "goodness" function, and make this tradeoff as they see fit. With the Précis environment, this iterative development cycle is shortened, as the fixed-point simulation code can be quickly generated.

3.5. Range finding

While the simulation support described above is very useful on its own for fixed-point simulation, it is only truly useful if the user can accurately identify the variables that they feel can be constrained. If the user does not really have an idea of where to begin, one place to start is utilizing the Précis range finding capability. The development cycle utilizing range finding is shown in

Figure 7.

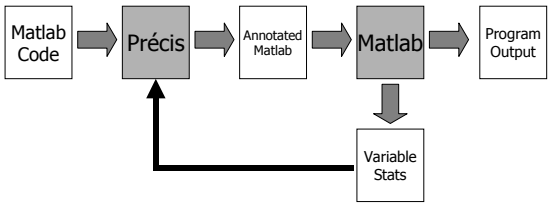

Figure 7. Development cycle for range finding analysis.

After the MATLAB code is parsed into the tool, the user can select variables they are interested in monitoring. Variables are targeted for range analysis and annotated MATLAB is generated, much like the simulation code is generated in the previous section. Instead of fixed-point simulation, though, Précis annotates the code with another MATLAB support routine that monitors the range of the values that the variables under question take on.

This support routine, '`rangefind`', monitors the maximum and minimum values attained by the variables. The annotated MATLAB is run with some sample data sets to gather range information on the variables under consideration. The user can then save these values in data files that can be fed back into Précis with another routine, '`saverangefind`'. Example range finding output is shown in Figure 8.

MATLAB Input	Range Finding Output
`a = 1;`	`a=1;`
`b = 2;`	`b=2;`
`c = 3;`	`c=3;`
`d = (a+(b*c));`	`d=(a+(b*c));`
	`rangeFind(d,'rfv_d');`

Figure 8. Sample range finding output.

The user then loads the resultant range values discovered by `rangefind` back into the Précis tool and (optionally) constrains the variables. The user now has an idea of what precision each variable requires for the sample data. Propagation can now be performed to determine the effect these precisions have on the rest of the system. Another useful step that the user can perform is to constrain the variables under question even further and perform a simulation to see how much error it introduces into the output. The results from this range finding method, however, are data set dependent. If the user is not careful to use representative data sets, the final hardware implementation could still generate erroneous results if the data sets were significantly different in precision requirements, even on the same algorithm.

For this reason we will consider range-gathered precision information to be somewhat of a lower bound. Given that the precisions obtained from propagation are conservative estimates, or an upper bound, manipulating the difference between these two bounds leads us to another method of precision analysis—slack analysis.

4. Slack analysis

One of the goals of this tool is to provide the user with "hints" as to where the developer's manual precision analysis and hardware tuning efforts should be focused. Ultimately, it would be extremely helpful for the developer to be given a list of "tuning points" in decreasing order of potential overall reduction of circuit size. This way, the developer could start a hardware implementation using more generic data path precision and iteratively optimize code sections that would give them the most benefit to meet constraints, such as time, cost, area, performance, or power. We believe this type of "tuning list" would give a developer a head start on precision analysis and put them on the right path of development faster than non-automated techniques.

As mentioned earlier, if the user performs range finding analysis and propagation analysis on the same set of variables, the tool would obtain what would amount to a lower bound from range analysis and an upper bound from propagation. We consider the range analysis a lower bound because it is the result of true data sets. While other data sets may require even lower amounts of precision, we know we need *at least* the ranges gathered from the range analysis. Further testing with other data sets may show that some variables would require more precision. Thus, if we implement the design with the precision found, we might encounter errors on output, thus the premise that this is a lower bound.

On the other hand, propagation analysis is very conservative. For example, in the statement `a=b+c`, where `b` and `c` have been constrained to be 16 bits wide by the user, the resultant bit width of `a` may be *up to* 17 bits due to the addition. But in reality, both `b` and `c` may be well within the limits of 16 bits and an addition might never overflow into the 17^{th} bit position. For example, if `c`=λ-`b`, the range of values `a` could ever take on is governed by λ. To a person investigating section of code, this seems very obvious when `c` is substituted into `a=b+c`, but these types of more "macroscopic" constraints in algorithms can be difficult or impossible to find automatically. It is because of this that we can consider propagated range information to be an upper bound.

Given a lower and upper bound on the bit width of a variable, we can consider the difference between these two bounds to be the slack. The actual precision requirement is most likely to lie between these two bounds. Manipulating the precision of nodes with slack can net gains in precision system-wide, as changes in any single node may impact many other nodes within the

circuit. The reduction in precision requirements and the resultant improvements in area, power, and performance can be considered gain. Through careful analysis of the slack at a node, we can calculate how much gain can be achieved by manipulating the precision between these two bounds. Additionally, by performing this analysis independently for each node with slack, we can generate an ordered list of "tuning points" that the user should consider.

For this paper, we consider the reduction of the area requirement of a circuit to be gain. In order to compute the gain of a node with respect to area, power and performance, we need to develop basic hardware models to capture the effect of precision changes upon these parameters. One simple implementation that we have utilized is to provide simple weighting parameters for different operator types. Thus, for example, if an adder has an area model of *x*, it indicates that as the precision decreases by one bit, the area reduces linearly and the gain increases linearly. In contrast, a multiplier has an area model of *x^2*, indicating that the area reduction and gain achieved are proportional to the square of the word size. Intuitively, this would give a higher overall gain value for bit reduction of a multiplier than of an adder. Using these parameters, our approach can more effectively choose the nodes with the most possible gain to suggest to the user. We detail our methodology in the next section.

4.1. Performing slack analysis

The goal of slack analysis is to identify which nodes, when constrained by the user, are likely to have the greatest impact upon the overall circuit area. While we do not believe it is realistic to expect users to constrain all variables, most users would be able to consider how to constrain a few "controlling" values in the circuit.

Our method seeks to efficiently use designer time by guiding them to the next important variables to consider for constraining. Précis can also provide a stopping criterion for the user: we can measure the maximum possible benefit from future constraints by constraining all variables to their lower bounds. The user can then decide to stop further investigation when the difference between the current and "lower bound" areas is no longer worth further optimization.

Our methodology is straightforward. For each node that has slack, we set the precision to the range-find value, the lower bound. Then, we propagate the impact of that change over all nodes and calculate the overall gain for the change, system-wide. We record this value as the effective gain as a result of modifying that node. We then reset all nodes and repeat for the remaining nodes that have slack. We then order the resultant list of gain values in decreasing order and present this information to the user in a dialog window. The user then can see which nodes to change to get the highest gain and in what order. It is then up to the designer to consider these nodes and determine which, if any, should actually be more tightly constrained.

To further illustrate this analysis method, refer to the pseudo-code shown below.

```
    Algorithm: Slack Analysis
User Step #1: Constrain known variables
User Step #2: Perform propagation
User Step #3: Load range data for some set of variables
'n'

set list_of_gains to empty list
for each variable 'm' in 'n'
  set aggregate_gain = 0
  constrain range of 'm' to the range analysis value
  perform forward and reverse propagation over all
variables
  for all variables
    if range of variable is narrower than range originally
propagated in 'User Step #2
      set aggregate_gain += old_area – new_area
    end
  next
  add (variable 'm' and aggregate_gain) to list_of_gains
  for all variables
    reset range of variable to range originally propagated in
'User Step #2'
  next
next

sort(list_of_gains) by decreasing aggregate_gain
```

5. Benchmarks

In order to determine the effectiveness of our slack analysis methodology, we allowed the tool to perform slack analysis with propagated and range-found range values. To gauge how effective the suggestions were, we constrained the variables the tool suggested in the order they were suggested to us, and calculated the resulting area. The area was determined utilizing the same area model discussed in previous sections, i.e. giving adders a linear area model while multipliers are assigned an area model proportional to the square of their input word size. We also determined an asymptotic lower bound to the area by implementing all suggestions simultaneously to determine how quickly our tool would converge upon the lower bound.

5.1. Wavelet Transform

The first benchmark we present is the wavelet transform. The wavelet transform is a form of image processing, primarily serving as a transformation prior to applying a compression scheme, such as SPIHT[8]. A typical discrete wavelet transform runs a high-pass filter and low-pass filter over the input image in one dimension.

The results are down sampled by a factor of two, effectively spatially compressing the wavelet by a factor of two. The filtering is done in each dimension, vertically and horizontally for images. Each pass results in a new image composed of a high-pass and low-pass sub-band, each half the size of the original input stream. These sub-bands can be used to reconstruct the original image.

This algorithm was hand-mapped to hardware as part of work done by Thomas Fry[8]. The hardware utilized was a WildStar FPGA board from Annapolis Microsystems consisting of three Xilinx Virtex 2000E FPGAs and 48 MBytes of memory. Significant time was spent converting the floating-point source algorithm into a fixed-point representation by utilizing methodologies similar to those we present in this paper. The result was an implementation running at 56MHz, capable of compressing 8-bit images at a rate of 800Mbits/sec. This represents a speedup of nearly 450 times as compared to a software implementation running on a Sun SPARCStation 5.

The wavelet transform was implemented in MATLAB and passed into Précis. In total, 27 variables were selected to be constrained. These variables were then marked for range-finding analysis and annotated MATLAB code was generated. This code was then run in the MATLAB interpreter with a sample image file (Lena) to obtain range values for the selected variables. These values were then loaded into Précis to obtain a lower bound to be used during the slack analysis phase. The results of the slack analysis are shown in Figure 9.

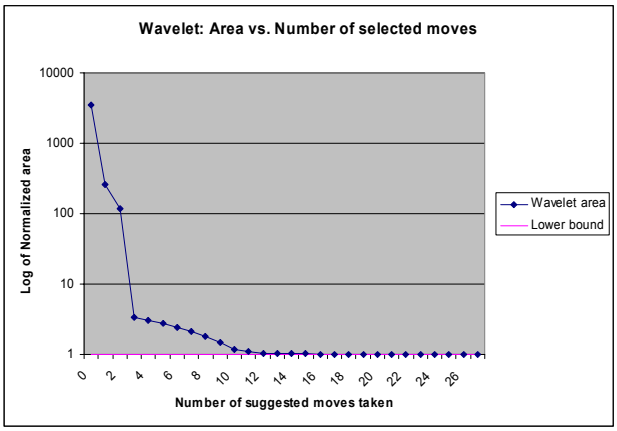

Figure 9. Wavelet area vs. number of suggestions implemented.

These results are normalized to the lower bound obtained by setting all variables to their lower bound constraints and computing the resulting area. This graph shows that between the upper bound and lower bound, there is a theoretical area difference of about three orders of magnitude. The slack analysis results suggested constraining the output image array, then the low and high pass filter coefficients, and then the results of the additions in the multiply-accumulate structure of the filtering operation. By taking the suggested moves in order and recomputing the order at each step, we were able to reach with ten percent of the lower bound area of the system in eleven moves. Perhaps more importantly, the tool was able to suggest a pattern of moves that would allow us to reach within a factor of three from the lower bound in just four moves. Finally, by about thirteen moves, the normalized area was within less than three percent of the lower bound, and further improvements were negligible. At this point a typical user may choose to stop optimizing the system.

It is important to note that the area values obtained by Précis are simply calculated by reducing the range of a number of variables to their range-found lower bounds. This yields what could be considered the "best-case" solution when optimizing. In reality, though, one would add another step to the development cycle whereby upon choosing the variable for optimization as suggested by the tool, the developer would perform an intermediate simulation step to determine if, by lowering the precision requirements of that variable, any error would be introduced in the results. This step is made easier by the automatic generation of annotated simulation code for use in MATLAB. In many cases, there might be an intolerable amount of error introduced by utilizing the lower bound, in which case the user would choose an appropriate precision range, fix that value as a constraint upon that variable in Précis and continue utilizing the slack analysis phase to find the next variable for optimization.

5.2. Probabilistic Neural Network: PNN

Another benchmark we investigated was a multi-spectral image-processing algorithm designed for NASA satellite imagery that is similar to clustering analysis or image compression. More details can be found in [7]. Briefly, each multi-spectral image pixel vector is compared to a set of "training pixels" or "weights" that are known to be representative of a particular class. The probability that the pixel under test belongs to the class under consideration is given by the formula depicted in Equation 1.

$$f(\vec{X} \mid S_k) = \frac{1}{(2\pi)^{d/2}\sigma^d}\frac{1}{P_k}\sum_{i=1}^{P_k}\exp\left[-\frac{(\vec{X}-\vec{W}_{ki})^T(\vec{X}-\vec{W}_{ki})}{2\sigma^2}\right]$$

Equation 1. The core PNN formula.

Here, $\vec{X}$ is the pixel vector under test, $\vec{W}_{ki}$ is the weight i of class k, d is the number of spectral bands, k is the class under consideration, σ is a data-dependent "smoothing" parameter, and P_k is the number of weights in class k. This formula represents the probability that

pixel vector $\vec{X}$ belongs to the class S_k. This comparison is then made for all classes and the class with the highest probability indicates the closest match.

This algorithm was manually implemented on a WildChild board and described in [7]. The WildChild board from Annapolis Microsystems consists of eight Xilinx 4010E FPGAs, a single Xilinx 4028EX FPGA, and 5MBytes of memory. Like the wavelet transform described earlier, significant time and effort was spent on variable range analysis, with particular attention being paid to the large multipliers and the exponentiation required by the algorithm. This implementation obtained speedups of 16 versus a software implementation on an HP workstation.

The algorithm was implemented in MATLAB and passed into Précis. From here, twelve variables were selected for range finding analysis, annotated MATLAB was generated, range-analysis was performed, and slack analysis was run utilizing the derived lower and upper bounds.

Again, all results were normalized to the lower bound area. As shown in Figure 10, the tool behaved similarly to the wavelet benchmark in that it was able to reach within five percent of the lower bound within six moves, where after additional moves serve to make only minor improvements in area. However, with the PNN algorithm, we are able to demonstrate even further refinement of the slack analysis approach.

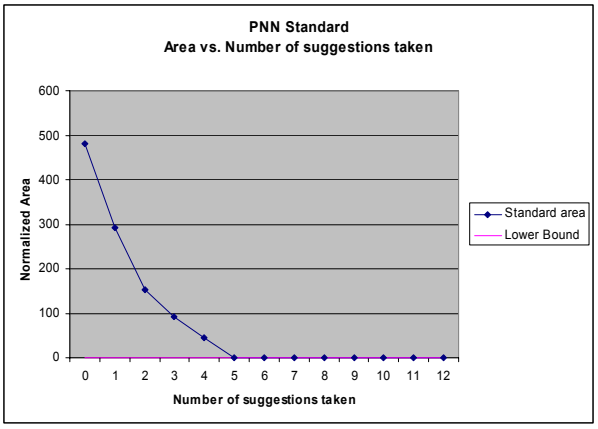

Figure 10. PNN area vs. number of suggestions implemented utilizing only range-analysis-discovered values.

For a seasoned developer that has a deeper insight into the algorithm, or for one that already has an idea of how the algorithm would map to hardware, the range-analysis phase sometimes returns results that are sub-optimal. For example, the range-analysis of the PNN algorithm upon a typical dataset resulted in several variables being constrained to ranges such as [2^0,2^-25], [2^8,2^-135], [2^0,2^-208], and so on. This simply means that the range-finding phase discovered values that were extremely small and thus recorded the range as requiring many bits to the right of decimal point to capture all the precision information. The shortcoming of the automated range-analysis is that it has no means by which to determine at what precision values become too small to affect follow-on calculations, and therefore might be considered unimportant. With this in mind, the developer would typically restrict the variables to narrower ranges that preserve the correctness of the results while requiring fewer bits of precision.

Précis provides the functionality to allow the user to make these decisions in its annotated MATLAB code generation. In this case, the user would choose a narrower precision range and a method by which to constrain the variable to that range consistent with how they will be implementing the operation in hardware—truncation, saturation, rounding, or any of the other methods presented in previous sections. Then, the developer would generate annotated MATLAB code for simulation purposes, and re-run the algorithm in MATLAB with typical data sets. This would allow the user to determine how narrow of a precision range would be tolerable, and subsequently constrain the variables in Précis accordingly. The user can perform this determination either during slack analysis, or prior to beginning slack analysis.

There are two types of scenarios that may occur depend primarily on the experience level of the developer. With a developer that has not dealt with precision analysis and software to hardware mappings extensively, it may be that they wouldn't notice the unreasonable range information obtained by the range-finding analysis phase until the variable was suggested for optimization by the tool. For this case, the user would perform an appropriate simulation of the variable at that stage of the slack analysis and obtain tighter bounds. On the other hand, for a more experienced hardware designer that has encountered precision analysis before, they might look closely at the range-finding results prior to running the slack analysis. In this case, they would most likely run simulations and find more reasonable precision ranges for the variables in question, and constrain them before even beginning the slack analysis phase.

The results for these two scenarios are shown plotted together in Figure 11, normalized to the lowest bound among all three approaches. To differentiate the three methods, the first proposed method is shown as "simple", and is the same method used to plot the results for the wavelet benchmark. The "user guided" method refers to fixing the variables during slack analysis. Finally, the "start constrained" method denotes fixing the variables in question prior to starting slack analysis.

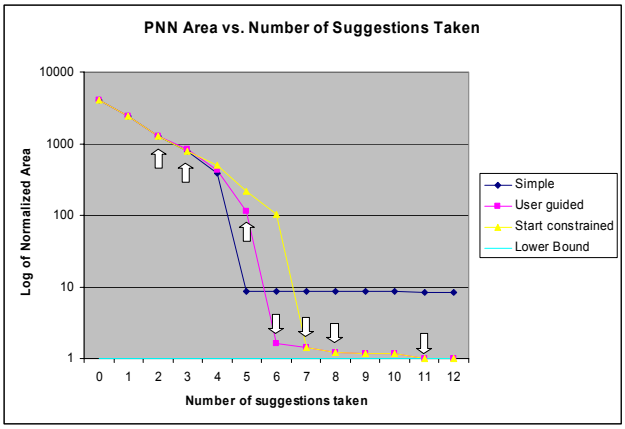

Figure 11. PNN area with user-defined variable precision ranges. Moves that had variables constrained to more reasonable ranges are highlighted with arrows.

At first glance, one can see that all three methods provide similar trends, approaching the lower bound within five to seven moves. This behavior is expected and is consistent with the results of the wavelet benchmark. However, one might expect that the start-constrained and user-guided methods would reach near the lower bound more quickly than the simple method. Instead, they take one or two additional moves to get near the lower bound compared to the simple method. This can be explained by understanding how the tool performs propagation across variables whose ranges are constrained by the user. By fixing the range of a variable, neither forward nor backward propagation will alter their precision ranges. In effect, we trust the user's decision when they fix a variable's precision range. The net effect is that any gains that might have been realized through back-propagation of smaller ranges will not be achieved if they must propagate through a variable whose range has been fixed. Finally, as the method used to compute the order of variables to constrain is greedy by nature, changing the order in which constraints are applied will alter the curve slightly.

6. Related work

While there have been several recent research efforts targeting precision analysis, none have approached it in such an interactive fashion. As mentioned in previous sections, Mark Stephenson and Jonathan Babb's work developing the Bitwise compiler at MIT [6] is an excellent foundation work regarding precision propagation techniques. They have applied their techniques in the SUIF compiler infrastructure and are targeting the C language for silicon compilation.

Anshuman Nayak's work at Northwestern University [9] is very relevant to our own research, as it is based upon the same MATCH compiler framework as our own. This work utilizes a similar propagation engine within the MATCH compiler as optimization phases and attempts to perform all analysis, including error, automatically, generating RTL VHDL suitable for synthesis.

Two other research efforts, one at the University of Southern California and one at Imperial College in London, approach the precision matter in an entirely different way. Kiran Bondalapati's work on dynamic precision management of loop computations [10] concentrates on developing a formal methodology for analyzing the precision requirements of loop structures. Finally, George A. Constantinides, et. al. have developed a Synoptix-based system for the analysis and automated generation of DSP applications[11].

7. Conclusions

In this paper we have demonstrated the need for precision analysis tools in the development cycle of software to hardware mapping. To direct the developer's efforts in hand-optimizing the precision of algorithms mapped to hardware, we have developed and demonstrated a tool, Précis, which allows the user to automate many tasks necessary for effective precision analysis. We have demonstrated how our tool can aid the developer in simulation of fixed-point math with automatic annotated MATLAB code generation. We have also developed MATLAB scripts that support range analysis of a user's MATLAB code in order to deduce a theoretical lower bound to the precision of selected variables. We have also presented a framework for propagation of precision range information over a MATLAB program. Finally, we have described our methodology of slack analysis, and have shown how the suggestions provided by this methodology can be helpful in guiding the user in their manual precision optimization on real-world benchmarks.

8. Acknowledgements

This research was supported by contracts with NASA and DARPA, and a grant from NSF. Scott Hauck was supported in part by an NSF CAREER award and a Sloan Research Fellowship.

9. References

[1] Synopsis CoCentric SystemC Compiler. http://www.synopsys.com/products/cocentric_systemC/cocentric_systemC_ds.html

[2] Celoxia Handel-C Compiler. http://www.celoxica.com/products/technical_papers/datasheets/DATHNC002_0.pdf

[3] P. Banerjee, N. Shenoy, A. Choudhary, S. Hauck, C. Bachmann, M. Chang, M. Haldar, P. Joisha, A. Jones, A. Kanhare, A. Nayak, S. Periyacheri, M. Walkden. "MATCH: A MATLAB Compiler for Configurable Computing Systems". Technical report CPDC-TR-

9908-013, submitted to IEEE Computer Magazine, August 1999.

[4] P. Banerjee, A. Choudhary, S. Hauck, N. Shenoy. "The MATCH Project Homepage". http://www.ece.nwu.edu/cpdc/Match/Match.html (1 Sept. 1999).

[5] AccelChip, info@accelchip.com, http://www.accelchip.com.

[6] Mark Stephenson. "Bitwise: Optimizing Bitwidths Using Data-Range Propagation". Master's thesis. Massachusetts Institute of Technology. May 2000.

[7] Mark L. Chang. "Adaptive Computing in NASA Multi-Spectral Image Processing". Master's Thesis. Northwestern University, Evanston, IL. December 1999.

[8] Thomas W. Fry. "Hyperspectral Image Compression on Reconfigurable Platforms". Master's Thesis. University of Washington, Seattle, IL. May 2001.

[9] A. Nayak, M. Haldar, A. Choudhary, P. Banerjee, "Precision And Error Analysis Of MATLAB Applications During Automated Hardware Synthesis for FPGAs", Proc. Design Automation and Test in Europe (DATE 2001), Berlin, Germany. Mar. 2001.

[10] Kiran Bondalapati and Viktor K. Prasanna, "Dynamic Precision Management for Loop Computations on Reconfigurable Architectures", IEEE Symposium on Field-Programmable Custom Computing Machines, April 1999.

[11] George A. Constantinides, Peter Y.K. Cheung, Wayne Luk, "The Multiple Wordlength Paradigm", IEEE Symposium on Field-Programmable Custom Computing Machines, April 2001.