Précis: A User-Centric Wordlength Optimization Tool

Mark L. Chang and Scott Hauck, University of Washington

***Abstract*— Currently, few tools exist to aid the hardware developer in translating an algorithm designed for a general-purpose processor into one that is precision-optimized for custom logic. 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.**

***Index Terms*— fixed-point arithmetic, wordlength optimization, MATLAB.**

I. Introduction

ONE of the most difficult tasks in implementing an algorithm in custom hardware is dealing with precision issues. Typical general-purpose processor concepts such as *word size* and *data type* are no longer valid in the world of custom logic where data paths can be custom-tailored to suit the needs of the algorithm. 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`, and `double`. These data types correspond to differently-sized data paths within the microprocessor. Most 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, custom and customizable hardware, such as an ASIC or FPGA, does not have predefined data widths for its data path. This allows the developer to tune the data paths to any width desired. Unfortunately, choosing the appropriate size for data paths is not a trivial task. Too many bits along a data path is wasteful, while too few may result in erroneous output.

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

II. Background

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.

A typical tool flow requires the developer to first convert a software prototyped algorithm into a hardware description. From this hardware description language (HDL) specification, various intermediate tools are used to perform simulation and generate target bitstreams which are then executed on reconfigurable logic.

A simple conversion without precision analysis would most likely yield an unreasonably large hardware implementation. For example, by choosing to emulate a general-purpose processor, or DSP, with 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, incorrect results, 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. While there are C-to-Verilog and C-to-VHDL tools in existence, such as the Synopsis CoCentric SystemC Compiler [1] and the Celoxia Handel-C Compiler [2], respectively, they do not offer such "designer aids" that would help with precision analysis of existing algorithms implemented in a high level language.

III. Related work

Most related work can be grouped into simulation-based, analytical, or a hybrid of the two techniques. They can be otherwise categorized by the amount of user interaction required to perform analysis, and the amount of feedback they can provide to the user.

Sung, et. al. [3] introduced a method and tool for wordlength optimization targeting custom VLSI implementations of digital signal processing algorithms. Purely simulation-based, they utilized first an internal and proprietary VHDL-based simulation environment [4]. This software was released as a commercial tool, "Fixed-Point Optimizer" [5], [6]. This release required the user to design a performance evaluation block in the description language. This block would return a positive value when the quantization effects on the output were within acceptable limits. Common blocks were signal-to-quantization-noise ratio (SQNR) computations. The system used basic hardware models from a commercial VLSI standard cell library to estimate the hardware cost of different implementations. Results were positive but required a lot of manual user intervention. While not inherently a drawback, the lack of optimization suggestions for the developer and the reliance on a programmatically determined "goodness function" differentiates it in motivation from our work.

In a closely related effort [7], operator overloading in C++ was utilized to perform range estimation of variables and fixed-point simulation. This work achieves the ability to simulate and estimate the ranges of non-linear and time-varying algorithms. However, it is still a completely manual optimization routine for the developer with only a simulation-based analysis and no hardware models to aid in area estimation.

A somewhat similar effort is described in [8]. It, too, uses standard general-purpose programming languages and custom libraries and data types to perform the fixed-point simulation. This work introduces the idea of interpolating ranges of intermediate variables without requiring the user to specify them explicitly. However, the steps toward efficient optimization are left for the user to deduce in an interactive optimization manner with no suggestions provided by the system.

[9]–[11] focus on developing algorithms for nearly fully-automatic wordlength optimization. These efforts still require a user-supplied criterion, either a latency target in [9] or a "goodness" function evaluator in [10], [11]. While this is very nearly an automatic process, the techniques employed limit the scope of the work to only linear time-invariant (LTI) systems. It should be noted that [12] extends the previous efforts to non-linear components in a data path and investigates the effect of precision optimization on power reduction.

Finally, [13], [14] introduce the Bitwise precision-analysis engine and the DeepC Silicon Compiler. These tools operate on C source code and provide a fully automatic static analysis approach to precision analysis and bitwidth reduction. This tool does not allow the developer to optimize bit-widths further while tolerating an error impact on the output, nor does it perform any suggestions to the user as to what directions to take for iterative optimization.

IV. User-Centric Automation

Much of the existing research focuses on fully-automated optimization techniques. While these methods have been shown to achieve good results, it is our belief that the developer should be kept close at hand during all design phases as they possess key information that an automatic optimization methodology simply cannot deduce or account for.

In order to guide an automatic precision optimization tool, a goodness function must be used to evaluate the performance of any optimization steps. In some cases, such as two-dimensional image processing, a simple signal-to-noise ratio (SNR) may be an appropriate goodness function. In other cases, the goodness function may be significantly more complex and therefore more difficult to develop. In either case, the developer still has the burden of implementing a goodness function within the framework of the automatic optimization tool.

By simulating a human developer's evaluation of what is an appropriate tradeoff between quality of result and hardware cost, the automatic optimization tool looses a crucial resource: the knowledgeable developer's greater sense of context in performing a goodness evaluation. Not only is this valuable resource lost, for many classes of applications a programmatically evaluated goodness function may be difficult or even impossible to implement. In other words, for many applications, a knowledgeable developer may be the best, and perhaps only, way to guide precision optimization. Therefore, there are many instances where a fully-automatic precision optimization tool should not or cannot be used.

In a departure from previous work utilizing fully-automatic methods, we approach this problem by providing a "design-time" precision analysis tool that interacts with the developer to guide the optimization of the hardware data path.

In performing manual data path optimization, one finds that the typical sequence of steps requires answering four questions regarding the algorithm and implementation:

1) What are the provable precision requirements of my algorithm?
2) What are the effects of fixed-precision on my results?
3) What are the actual precision requirements of my data sets?
4) Where along the data path should I optimize?

By repeatedly asking and answering these questions, hardware designers can perform effective wordlength optimization. The tradeoffs between area consumption and accumulated error within the computation need to be manually analyzed—a time consuming and error prone process with little tool support.

In order to fill the gap in *design-time, user-centric* tools that can aid in answering these precision questions, we introduce our prototyping tool, Précis.

V. Précis

Algorithms written in the MATLAB language serve as input to Précis. MATLAB is a very high-level programming language and prototyping environment that has found popularity particularly in the signal and image processing fields [15], [16]. More than just a language specification, MATLAB is an interactive tool with which developers can manipulate algorithms and data sets to quickly see the impact of changes on the output of an algorithm. The ease with which developers can explore the design space of their algorithms makes it a natural choice to pair with Précis to provide a design-time precision analysis environment.

Précis aids developers by automating many of the more mundane and error-prone tasks that are necessary to answer the four previously mentioned precision analysis questions. This is done by providing several integrated tools within a single application framework, including: constraint propagation, simulation support, range finding capabilities, and a slack analysis phase. It is designed to complement the existing tool flow at *design time*, coupling with the algorithm before it is translated into an HDL description and pushed through the vendor back-end bitstream generation tools. It is designed to provide a convenient way for the user to interact with the algorithm under consideration. The goal is for the knowledgeable user, after interacting with our tool and the algorithm, to have a much clearer idea of the precision requirements of the data paths within their algorithm.

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 Fig. 1. 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.

A. Propagation Engine

A core component of the Précis tool is a constraint propagation engine. The purpose of the constraint propagation engine is to answer the first of the four precision-analysis questions: *what are the provable precision requirements of my algorithm?* By learning how the data path of the algorithm under question grows in a worst-case sense, we can obtain a baseline for further optimization as well as easily pinpoint regions of interest—such as areas that explode in data path width—which may be important to highlight to the user.

The propagation engine, inspired in part by [13], [14], works by modeling the effects of using fixed-point numbers and fixed-point math in hardware. This is done by allowing the user

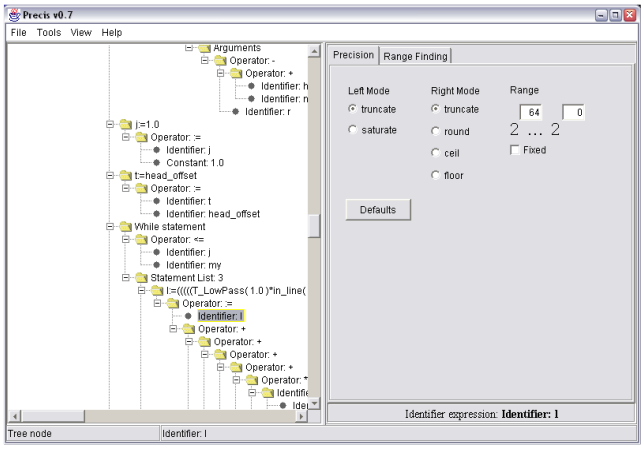

Fig. 1. Précis screenshot

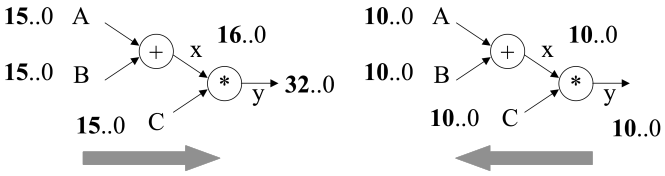

Fig. 2. Simple propagation example

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. This default width is chosen because it is the width of a double-precision floating-point number, the base number format used in the MATLAB environment. 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 MSB position of 2^{15} and a LSB position of 2^0, 16 bits, the resulting output range of variable a would have a range of $2^{16} - 1$ to $2^0 - 1$, 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. An example of forward and backward propagation is depicted in Fig. 2.

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 y having a range of [31, 0]. If, after

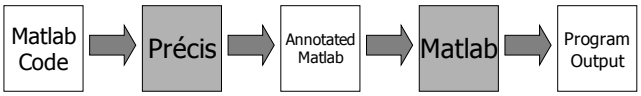

Fig. 3. Flow for code generation for simulation

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));`

Fig. 4. Sample output generated for simulation, with the range of a variable constrained

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.

The propagation engine is used to get a quick, macro-scale 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 allows the user to see a conservative estimate of how the input bit width affects the size of operations down stream. While the propagation engine provides some important insight into the effects of fixed-point operations on the resultant data path, it forms a conservative estimate. For example, in an addition, the propagation engine assumes that the operation requires the carry-out bit to be set. It would be appropriate to consider the data path widths determined from the propagation engine to be worst-case results, or in other words, an upper bound. This upper bound, as well as the propagation engine, will become useful in further analysis phases of Précis.

B. Simulation Support

To answer the second question during manual precision analysis: *what are the effects of fixed-precision on my results?*, the algorithm needs to be operated in a fixed-point environment. This is often done on a trial-and-error basis, as there are few, if any, structured, high-level fixed-point environments. To aid in performing fixed-point simulation, Précis easily produces annotated MATLAB code. 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, alleviating the need for the developer to construct custom fixed-point blocks. The simulation flow is shown in Fig. 3.

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 Fig. 4.

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 generates 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 width of the data paths used by some of the constrained variables.

During this manual phase of precision analysis, it is typically not sufficient to merely test whether the fixed precision results are identical to the unconstrained precision results, since this is likely 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 and executed, allowing the user to view the results and the impact of error without the tedious editing of algorithm source code.

C. 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. This leads to the third question that must be answered in order to perform effective data path optimization: *what are the actual precision requirements of my data sets?*

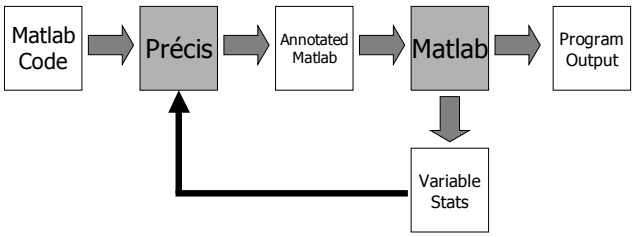

Fig. 5. Development cycle for range finding analysis

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');`

Fig. 6. Sample range finding output

Précis answers this question by providing a range finding capability that helps the user deduce the data path requirements of intermediate nodes whose ranges may not be obvious. The development cycle utilizing range finding is shown in Fig. 5.

After the MATLAB code is parsed, 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 for a further analysis phases. Example range finding output is shown in Fig. 6.

The user then loads the resultant range values discovered by `rangeFind` back into the Précis tool and (optionally) constrains the variables. The range finding phase has now given the user an accurate profile of what precision each variable requires for the data sets under test. Propagation can now be performed to conservatively estimate the effect these data path widths have on the rest of the system.

The propagation engine and the range finding tools work closely together to allow the user to build a more comprehensive picture of the precision requirements of the algorithm than either of the tools could do alone. The propagation engine, with user-knowledge of input and perhaps output variable constraints, achieves a first-order estimation of the data path widths of the algorithm. Using the range finding information allows for significant refinement of this estimation, as the discovered variable statistics allow for narrower data path widths to be realized that more closely reflect the true algorithmic precision requirements.

Another useful step that the user can perform is to constrain variables even further than suggested by the range finding phase. The user then performs subsequent simulations to see if these further refinements introduce an acceptable amount of into the result. These simulations, as before, are easily generated and executed within the Précis framework.

The results from this range finding method 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. Data sets that are precision representative of the common case as well as boundary and extreme cases should be used to allow the range finding phase to gather meaningful and robust statistics.

It is useful, therefore, to consider range-gathered precision information to be a lower bound on the precision required by the algorithm. As the data sets run by the user have been observed to exercise a known amount of data path width, any further reduction in the precision will likely incur error. Given that the precisions obtained from the propagation engine are conservative estimates, or an upper bound, manipulating the difference between these two bounds leads us to a novel method of user-guided precision analysis—slack analysis.

D. Slack Analysis

One of the goals of this work is to provide the user with "hints" as to where the developer's manual precision analysis and hardware tuning efforts should be focused. This is the subject of the fourth precision analysis question: *where along the data path should I optimize?*. 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. With this information, the developer could start a hardware implementation using more generic data path precision, such as a standard 64 or 32-bit data path, and iteratively optimize code sections that would yield the most benefit. Iteratively optimizing sections of code or hardware is a technique commonly used to efficiently meet constraints such as time, cost, area, performance, or power. We believe this type of "tuning list" would give a developer effective starting points for each iteration of their manual optimization, putting them on the most direct path to meeting their constraints.

Recall that 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 to maintain an error-free output. 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 17th bit position. For example, if $c = \lambda - b$, the range of values a could ever take on is governed by λ. To a person investigating this 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 outside of LTI systems as described and modeled in [10], [11]. 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 when performing iteration of optimization.

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. For this work we use a simple area model as our main metric. For example, an adder has an area model of x, indicating 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, which is in line with the implementations that are familiar to hardware designers. Using these parameters, our approach can effectively choose the nodes with the most possible gain to suggest to the user. We detail our methodology in the next section.

E. 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.

```
PERFORMSLACKANALYSIS
 1 constrain user-specified variables
 2 perform propagation
 3 baseArea ← calculateArea()
 4 load range data for some set of variables n
 5 listOfGains ← ∅
 6 foreach m in n
 7     reset all variables to baseline precision
 8     constrain range of m to the range analysis value
 9     perform forward and reverse propagation
10     newArea ← calculateArea()
11     if (newArea < baseArea) then
12         listOfGains ← (m, baseArea − newArea)
13 sort listOfGains by decreasing gain
```

Fig. 7. Slack analysis pseudo-code

The user can then decide to stop further investigation when the difference between the current and lower bound area 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 in terms of area 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 the procedure for the remaining nodes that have slack. We then sort the resultant list of gain values in decreasing order and present this information to the user in a dialog window. From the graphical user interface, the user can easily see how and which nodes to modify to achieve the highest gain. It is then up to the designer to consider these nodes and determine which, if any, should actually be more tightly constrained than suggested by Précis. Pseudo-code for the slack analysis procedure is shown in Fig. 7.

VI. BENCHMARKS

In order to determine the effectiveness of Précis, we utilized the tool to optimize a variety of image and signal processing benchmarks. 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.

A. 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 [17], [18]. A typical discrete wavelet transform runs a high-pass filter and low-pass filter over the input image in one dimension. The results are then downsampled by a factor of two and the process is repeated in the other dimension. Each pass results in a new image composed

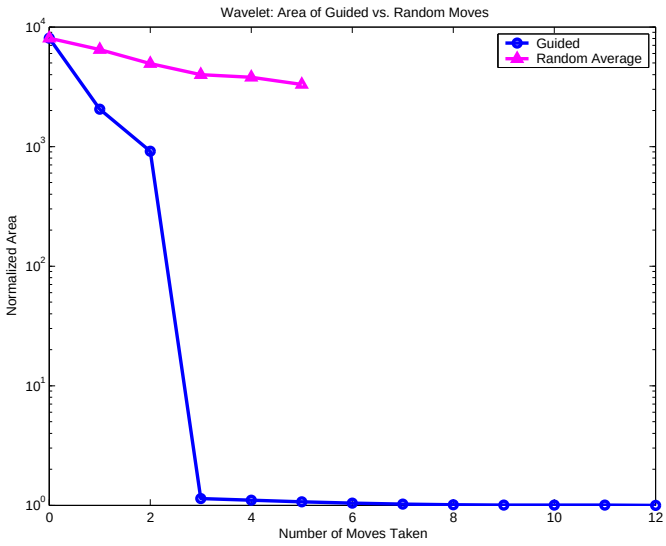

Fig. 8. Wavelet area vs. number of optimization steps implemented

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 [17]. 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 here, albeit by hand. 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 subsequently implemented in MATLAB and optimized in 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 Fig. 8.

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 four orders of magnitude. The slack analysis results suggested constraining the input 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 fifteen percent of the lower bound area of the system in three moves. By about seven 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.

To determine if this methodology is sound, we compared the suggested optimization steps to the performance if we were optimizing randomly, i.e. choosing randomly which variables would be constrained. We performed five optimization runs of five random optimization steps each using the same values for the upper and lower precision bounds as in the guided optimization scheme. The average area of these five random passes is plotted versus the guided slack-analysis approach in Fig. 8. As can be seen, the guided optimization route that is suggested by Précis reaches very close to the lower bound quicker than the random method. The random method, while still improving with each optimization step, does so much more slowly than the guided slack analysis approach. From this we conclude that our slack analysis approach is providing useful feedback to the user in terms of what nodes to optimize and in what order to make the largest gains in the fewest number of optimization moves.

The rather large gains in terms of area achieved by the optimization steps are an artifact of the testing methodology. In all of the benchmarks, the initial ranges for all of the nodes, input, output, and intermediate variables, were set to a worst-case 64-bits, the width of a double-precision word in MATLAB. We chose this as our starting point simply because the MATLAB environment defaults to this data type when no other type is defined. While moving to a narrower default data width would reduce the height of the y-axis in the resulting figures, it would not change the shape of the curves, which clearly indicate that there is a benefit to choosing the correct order of nodes to optimize. The selection of a 64-bit wide default word is therefore not arbitrary, but rather fair given the nature of MATLAB, the input specification language for Précis.

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.

B. CORDIC

The next benchmark investigates the CORDIC algorithm [19], an acronym that stands for **CO**ordinate **R**otation **DI**gital **C**omputer. The algorithm is novel in that it is an iterative solver

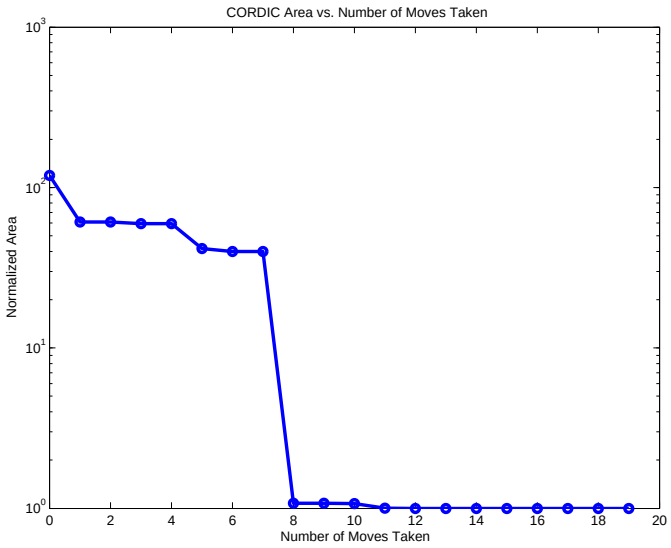

Fig. 9. CORDIC benchmark results

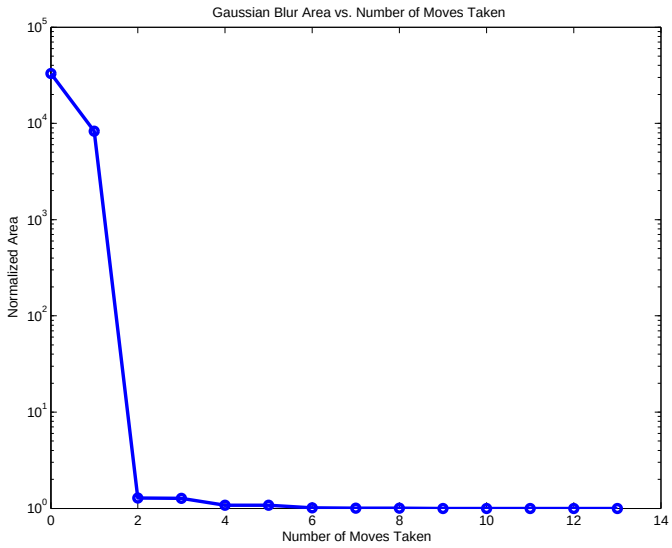

Fig. 10. Gaussian blur results

for trigonometric functions that requires only a simple network of shifts and adds and produces approximately one additional bit of accuracy for each iteration. A more complete discussion of the algorithm as well as a survey of FPGA implementations can be found in [20].

The CORDIC algorithm can be utilized in two modes, rotation mode and vectoring mode. For this benchmark, we utilized rotation mode which rotates an input vector by a specified angle, simultaneously computing the sine and cosine of the input angle. As in [20], the difference equations for rotation mode are:

$$\begin{aligned} x_{i+1} &= x_i - y_i * d_i * 2^{-i} \\ y_{i+1} &= y_i + x_i * d_i * 2^{-i} \\ z_{i+1} &= z_i - d_i * \tan^{-1}(2^{-i}) \end{aligned}$$

where

$$d_i = \begin{cases} -1 & \text{if } z_i < 0 \\ +1 & \text{otherwise} \end{cases}$$

The MATLAB implementation of CORDIC was unrolled into twelve stages. In order to obtain a variety of variable range information during the range finding phase of the analysis, a test harness was developed that swept the input angle through all integer angles between $0°$ and $90°$. The results were then passed into Précis and all 41 intermediate nodes were chosen for slack analysis. The results are consistent with those in the wavelet benchmark, and are shown in Fig. 9, truncated to the first 20 moves suggested by the tool.

The suggested moves do not converge upon the lower bound as quickly as the wavelet benchmark—reaching within eight percent in eight moves. This can be attributed to the fact that the slack analysis algorithm is greedy in nature. The first few proposed moves all originate at the outputs. Only after these are constrained does the slack analysis suggest moving to the input variables. This behavior is in part due to the depth of the adder tree present in the twelve-stage unrolling of the algorithm. The gain achieved by constraining the outputs is greater than the limited impact of constraining any one of the inputs because the output nodes are significantly larger. Shortly thereafter, though, all the input variables have been constrained, giving us the large improvement in area after the seventh suggested move, at which point the very linear data path of the CORDIC algorithm has been collapsed to near the lower bound.

C. Gaussian Blur

The third benchmark is a Gaussian blur implemented as a spatial convolution of a 3x3 Gaussian kernel with a 512x512 greyscale input image. We ignore rescaling of the blurred image for simplicity. The Gaussian blur algorithm was was passed into Précis and 14 intermediate nodes were chosen for the slack analysis phase. The results are shown in Fig. 10. The slack analysis prompted us to constrain first the Gaussian kernel followed by the input image. This led to the largest area improvement—within 29 percent of the lower bound in two moves, and within eight percent in 4 moves. Again, the tool selects clear choices for optimization and achieves performance near the lower bound in a few optimization steps.

D. 1-D Discrete Cosine Transform

The next benchmark is a one-dimensional discrete cosine transform. The DCT [21] is a frequency transform much like the discrete Fourier transform, but using only real numbers. It is widely used in image and video compression. Our implementation is based upon the work done by [22] as used by the Independent JPEG Group's JPEG software distribution [23]. This implementation is requires only 12 multiplications and 32 additions.

Our MATLAB implementation performed an 8-point 1-D DCT upon a 512x512 input image. Fig. 11 shows the area trend as moves are selected as suggested by Précis for the first 16 nodes of the total 25 nodes chosen for slack analysis. To get within a factor of two of the lower bound, the input

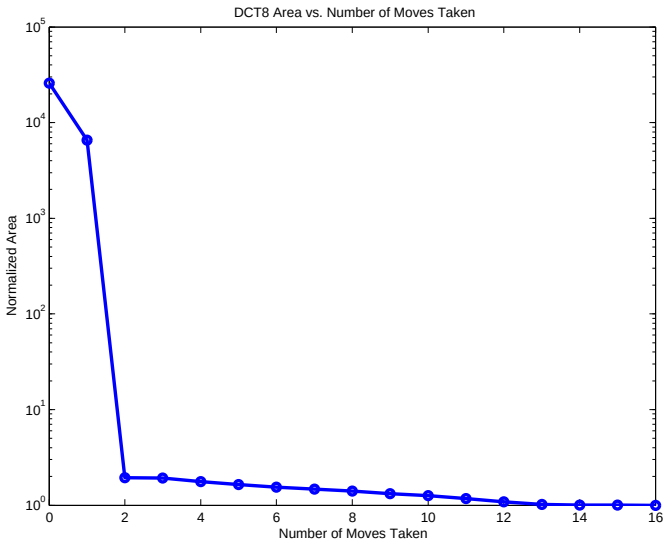

Fig. 11. 8-point 1-D DCT results

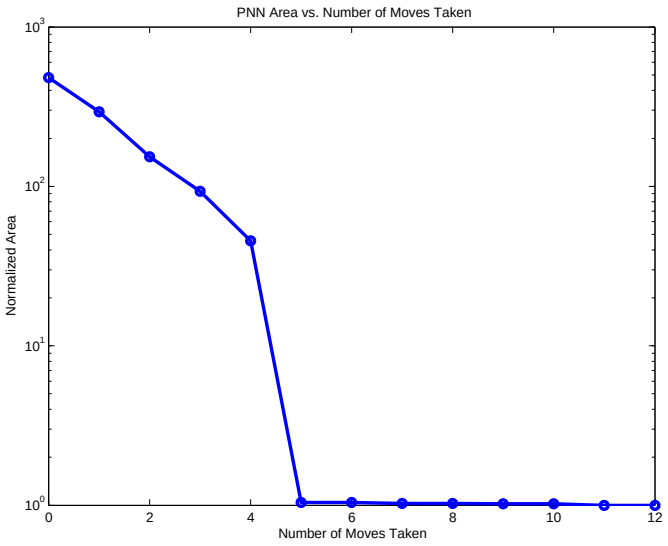

Fig. 12. PNN area vs. number of optimization steps implemented utilizing only range-analysis-discovered values

image and DCT input vector are constrained. The suggested moves achieve within 50 percent of the lower bound within seven moves, and within nine percent in 12 moves.

E. Probabilistic Neural Network

The final benchmark we investigated was a multi-spectral image-processing algorithm designed for NASA satellite imagery that is similar to clustering analysis or image compression. The goal of the algorithm is to use multiple spectral bands of instrument observation data to classify each image pixel into one of several classes. For this particular application, these classes define terrain types, such as urban, agricultural, rangeland, and barren. In other implementations, these classes could be any significant distinguishing attributes present in the underlying dataset. This class of algorithm transforms the multi-spectral image into a form that is more useful for analysis by humans.

One proposed scheme to perform this automatic classification is the Probabilistic Neural Network classifier [24]. In this implementation, 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 the formula depicted below.

$$f(\overrightarrow{X}|S_k) = \frac{1}{(2\pi)^{d/2}\sigma^d} * \frac{1}{P_k} * \sum_{i=1}^{P_k} \exp\left[-\frac{(\overrightarrow{X} - \overrightarrow{W_{ki}})^T(\overrightarrow{X} - \overrightarrow{W_{ki}})}{2\sigma^2}\right]$$

Here, $\overrightarrow{X}$ is the pixel vector under test, $\overrightarrow{W_{ki}}$ 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 $\overrightarrow{X}$ belongs in 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 [25]. 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 optimized with Précis. Twelve variables were selected and slack analysis was run as in the previous benchmarks. Again, all results were normalized to the lower bound area. As shown in Fig. 12, the tool behaved consistent with other benchmarks and was able to reach within five percent of the lower bound within six moves, whereafter additional moves serve to make only minor improvements in area. However, with the PNN algorithm, we take the time to demonstrate a method to provide even further refinement of the slack analysis approach with Précis.

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 fractional bits (bits 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

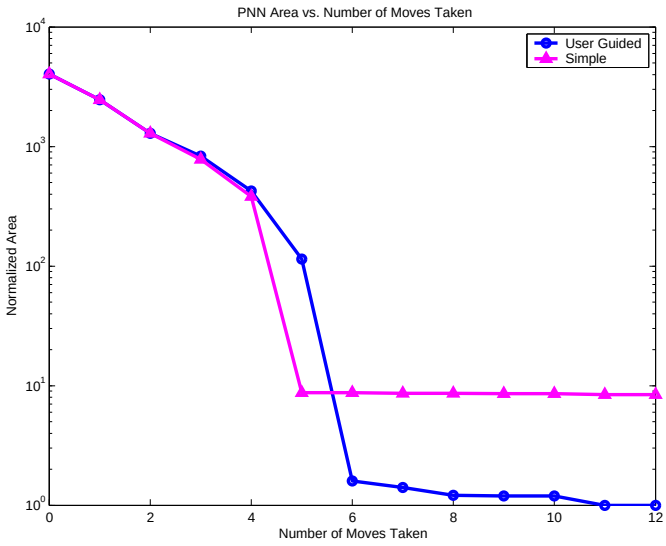

Fig. 13. PNN area with user-defined variable precision ranges

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 implement 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 would then be able to continue the slack analysis phase, optionally reconstraining variables through use of simulation as wider-than-expected precision ranges were encountered. We present the results from this experiment in Fig. 13, normalized to the lowest bound between the standard and "user-guided" approaches.

At first glance, one can see that both 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 other benchmarks. The results show that the user-guided approach, when reconstraining variables during slack analysis, achieves a lower bound eight times lower than the previously described method. As expected, the simpler method does not improve further as the number of optimization steps in increased, and remains fixed at ten times the lower bound.

The intuition of the hardware developer is used in this case to achieve a more area-efficient implementation than was possible with the unguided slack analysis optimization. The ability to keep the "user in the loop" for optimization is crucial to obtaining good implementations, something that Précis is clearly able to exploit.

VII. Conclusions

A tool for semi-automatic, user-centric, design-time precision analysis has been presented. Précis combines an automatic propagation engine, a fixed-point simulation environment with automatic MATLAB code generation, MATLAB routines also with automatic code generation for variable statistics gathering, and a slack analysis phase. Together, this toolchest addresses a major shortcoming of automated data path optimization techniques: leaving the developer out of the optimization. We have demonstrated an effective methodology for guiding the developer's eventual manual optimization toward those regions of the data path that will provide the largest area improvement.

Précis aids new and seasoned hardware developers in answering the four basic questions needed to perform data path optimization at a very high level, before HDL is generated. At this time, small design changes almost always lead to large differences in performance of the final implementation. Thus, it is crucial to have assistive tools from the very beginning of the design cycle, in particular, data path optimization. Unfortunately, there are few commercial and academic tools that provide this level of support, which highlights the importance of this contribution.

Ongoing research is focused on developing techniques for accurate area and quantization error estimation for precision analysis. Specifically, developing methodologies to select the least significant bit position along a data path subject to area and error constraints.

References

[1] Synopsys, "Synopsys CoCentric SystemC Compiler," September 2003. [Online]. Available: http://www.synopsys.com/

[2] Celoxia, "Handel-C Compiler," September 2003. [Online]. Available: http://www.celoxia.com/

[3] W. Sung and K.-I. Kum, "Simulation-based word-length optimization method for fixed-point digital signal processing systems," *IEEE transactions on Signal Processing*, vol. 43, no. 12, pp. 3087–3090, December 1995.

[4] K.-I. Kum and W. Sung, "VHDL basd fixed-point digital signal processing algorithm development software," in *Proceedings of the IEEE International Conference on VLSI CAD*, November 1993, pp. 257–260.

[5] W. Sung and K.-I. Kum, "Word-length determination and scaling software for signal flow block diagram," in *International Conference on Acoustic, Speech, and Signal Processing*, Adelaide, Australia, 1994, pp. 457–460.

[6] *Fixed-Point Optimizer User's Guide*, Alta Group of Cadence Design Systems, Inc., August 1994.

[7] S. Kim, K.-I. Kum, and W. Sung, "Fixed-point optimization utility for C and C++ based digital signal processing programs," in *Workshop on VLSI and Signal Processing*, Osaka, 1995.

[8] M. Willems, V. Bürsgens, H. Keding, T. Grötker, and H. Meyr, "System level fixed-point design based on an interpolative approach," in *Design Automation and Test in Europe*, 1997.

[9] G. A. Constantinides, P. Y. Cheung, and W. Luk, "Heuristic datapath allocation for multiple wordlength systems," in *Design Automation and Test in Europe*, 2001.

[10] G. A. Constantinides, P. Y. Cheung, and W. Luk, "Optimum wordlength allocation," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002.

[11] G. A. Constantinides, P. Y. Cheung, and W. Luk, "The multiple wordlength paradigm," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[12] G. Constantinides, "Perturbation analysis for word-length optimization," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2003.

[13] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth analysis with application to silicon compilation," in *Proceedings of the SIGPLAN conference on Programming Language Design and Implementation*, June 2000.
[14] M. W. Stephenson, "Bitwise: Optimizing bitwidths using data-range propagation," Master's thesis, Massachusetts Institute of Technology, May 2000.
[15] C. B. Moler, "MATLAB — an interactive matrix laboratory," University of New Mexico. Dept. of Computer Science, Tech. Rep. 369, 1980.
[16] C. B. Moler, "MATLAB user's guide," University of New Mexico. Dept. of Computer Science, Tech. Rep., Nov. 1980.
[17] T. W. Fry, "Hyperspectral image compression on reconfigurable platforms," Master's thesis, University of Washington, Seattle, WA, May 2001.
[18] T. W. Fry and S. Hauck, "Hyperspectral image compression on reconfigurable platforms," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2002, pp. 251–260.
[19] J. Volder, "The CORDIC trigonometric computing technique," *IRE Trans. Electronic Computing*, vol. EC-8, pp. 330–334, September 1959.
[20] R. Andraka, "A survey of CORDIC algorithms for FPGAs," in *ACM/SIGDA Sixth International Symposium on Field Programmable Gate Arrays*, 1998.
[21] N. Ahmed, T. Natarajan, and K. Rao, "Discrete cosine transform," *IEEE Transactions on Computer*, vol. C-23, pp. 90–93, January 1974.
[22] C. Loeffler, A. Ligtenberg, and G. S. Moschytz, "Practical fast 1-D DCT algorithms with 11 multiplications," in *Proc. Int'l. Conf. on Acoustics, Speech, and Signal processing*, 1989, pp. 988–991.
[23] T. Lane, P. Gladstone, L. C. Jim Boucher, J. Minguillon, L. Ortiz, G. Phillips, D. Rossi, G. Vollbeding, and G. Weijers, "The independent JPEG group's JPEG software library," http://www.ijg.org/files/jpegsrc.v6b.tar.gz, June 2004.
[24] S. R. Chettri, R. F. Cromp, and M. Birmingham, "Design of neural networks for classification of remotely sensed imagery," *Telematics and Informatics*, vol. 9, no. 3-4, pp. 145–156, 1992.
[25] M. L. Chang, "Adaptive computing in NASA multi-spectral image processing," Master's thesis, Northwestern University, Dept. of ECE, December 1999.