MODEL-BASED PRECISION ANALYSIS AND OPTIMIZATION FOR DIGITAL SIGNAL PROCESSORS

Soujanya Kedilaya[1], *William Plishker*[2], *Aleksandar Purkovic*[1], *Brian Johnson*[1], *and Shuvra S. Bhattacharyya*[2]

[1]Texas Instruments
Germantown, MD 20874, USA
{kedilaya, apurkovic, bfjohnson}@ti.com

[2]University of Maryland
College Park, MD 20742, USA
{plishker, ssb}@umd.edu

ABSTRACT

Embedded signal processing has witnessed explosive growth in recent years in both scientific and consumer applications, driving the need for complex, high-performance signal processing systems that are largely application driven. In order to efficiently implement these systems on programmable platforms such as digital signal processors (DSPs), it is important to analyze and optimize the application design from early stages of the design process. A key performance concern for designers is choosing the data format. In this work, we propose a systematic and efficient design flow involving model-based design to analyze application data sets and precision requirements.

We demonstrate this design flow with an exploration study into the required precision for eigenvalue decomposition (EVD) using the Jacobi algorithm. We demonstrate that with a high degree of structured analysis and automation, we are able to analyze the data set to derive an efficient data format, and optimize important parts of the algorithm with respect to precision.

1. INTRODUCTION

With the increased need for the design and development of complex signal processing systems in scientific fields such as wireless communications, medical imaging, and high energy physics, which deal with large unpredictable data sets, a key design decision to make for efficient implementation is the choice of data format. In general, floating point digital signal processors (DSPs) achieve much greater precision and dynamic range at the expense of speed, since they require multiple cycles for each operation. On the other hand, fixed-point DSPs are favored for high-volume applications where unit manufacturing costs need to be kept low. But with the evolution of semiconductor technology, cost issues relating to size of the DSP core are no longer as significant. Overall, fixed-point DSPs still have an edge in cost and floating-point DSPs in ease of use, but the gap has narrowed to the point where the choice of using a fixed- or floating-point data format comes down to whether floating-point math is needed by the application data set [1].

With new Texas Instruments (TI) DSPs such as the TMS320C674x series supporting a superset of both fixed- and floating-point instruction sets, it has become possible to implement an application in fixed-point with only a subset of instructions requiring higher precision in floating-point. The challenge arises in identifying the precision bottlenecks in any given application in a systematic and highly automated manner.

In this work, we propose and demonstrate a methodology using model-based design for the analysis and optimized application of data precision in a signal processing system. Model-based design has proven to be an effective and efficient method for designing signal processing systems (e.g., see [2]). Predominantly model-based design is used to link designs directly to requirements, integrate testing with design, help isolate domain experts from the need to understand low-level hardware and software details, and use a common design abstraction across project teams. Additionally, model-based design is also effective for application exploration. In such exploration, a designer typically starts with a platform-independent description of the application. This structured, formal application description is a useful starting point for capturing concurrency and optimizing and analyzing the application.

The Dataflow Interchange Format (DIF) [3] provides one such tool for model-based design and implementation of signal processing systems. The DIF environment employs the methodology of dataflow-based application modeling. DIF allows for high-level application specification, software simulation, and synthesis for hardware or software implementation.

Dataflow modeling has often been used in precision analysis, most commonly in automatic floating-to-fixed-point conversion of DSP code. Since high level languages such as C do not have built-in fixed-point datatypes, it is common practice to develop DSP algorithms with floating point datatypes, and then implement them on fixed point architectures. Since the manual transformation of floating-point data to fixed-point data is time consuming and error prone, a significant volume of research has been focused on the automatic conversion of floating-point to fixed-point code (e.g., see [4, 5]). Some of these research works, such as those reported in [4, 6, 7], employ fine-grained dataflow graphs as an intermediate representation between the floating- and fixed-point programs. In this kind of intermediate representation, the dataflow graph has vertices representing relatively low level ("fine grained") operations, and the edges represent data variables. Using such a dataflow graph as the backbone, several statistical and/or analytical methods are applied at every node to compute and annotate vertices with their respective dynamic ranges, binary point positions, and ultimately, bit widths.

We adopt some of these methods for precision analysis and optimization. However, in most of these works, the starting point is not a high level application specification, instead it is floating point code for the targeted algorithm. The dataflow graph is a fine-grained, general purpose intermediate representation rather than a high-level representation that is in terms of coarse grain building blocks (e.g., digital filters and FFT modules) and formal dataflow models, such as synchronous or cyclo-static dataflow [8, 9]. Although fine-grained, general purpose representations are suitable for converting floating point code to fixed point code, they do not expose high level application structure effectively as formal models do, and therefore are significantly limited in terms of their capabilities to support high-level, cross-module application analysis and design exploration [2].

The *DSPCAD Integrative Command Line Environment* (*DICE*) [10] is a framework for facilitating efficient management of design and test of cross-platform software projects. DICE includes features, such as stream-oriented and dataflow module testing support, that are well suited for DSP applications. Although DIF and DICE are orthogonal (one can be employed without the other), there are useful synergies between the two tools, such as improving FPGA design processes for high performance signal processing [11].

In this work, we use DICE as a framework to simulate a DSP system that is modeled in DIF, and to evaluate component inter-

© EURASIP, 2011 - ISSN 2076-1465

actions and system-level metrics for this system. We demonstrate that this approach leads to an efficient methodology for precision analysis of DSP computations. We demonstrate this methodology with an exploration into the internal precision of computation for Jacobi Eigenvalue Decomposition (EVD) [12]. In this analysis, the mathematically intensive Jacobi EVD is modeled as a mixed-grain dataflow graph in DIF. We exploit the synergy between DIF and DICE when evaluating the precision requirements of the different operations. Based on this analysis, we are able to identify parts of the application that require higher precision, optimize these parts, and provide useful feedback to designers about the formulation of the algorithm. The approach provides a structured, highly automated solution to precision analysis problems, which helps to improve the quality of the derived precision configurations, and the efficiency and reliability of their derivation.

2. DATAFLOW MODELING

Modeling DSP applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models have been developed for dataflow-based design. A growing set of DSP design tools support such dataflow semantics. Designers are expected to be able to find a match between their application and one of the well-studied models, including cyclo-static dataflow (CSDF), synchronous dataflow (SDF) [8], single-rate dataflow, homogeneous synchronous dataflow (HSDF), or a more complicated model such as boolean dataflow (BDF) [13].

Common to each of these modeling paradigms is the representation of computational behavior as a dataflow graph. A dataflow graph G is an ordered pair (V,E), where V is a set of vertices (or nodes), and E is a set of directed edges. A directed edge $e = (v_1, v_2) \in E$ is an ordered pair of a source vertex $v_1 \in V$ and a sink vertex $v_2 \in V$. Vertices (*actors*) represent computations while edges represent FIFO communication links between actors.

3. MODEL-BASED PRECISION ANALYSIS

3.1 High-level Application Specification

System development often involves an initial application description in a design environment, which is then transcoded and tuned to target the implementation platform. We use DIF as our dataflow analysis engine and as our model-based development environment. The functionality of the actors in the application graph is implemented using *functional DIF*, which is a plug-in to the DIF package that provides functional simulation capabilities for signal processing systems that are represented as dataflow graphs [14].

3.2 Proposed Methodology — Dynamic Range Analysis

The required precision for any data variable can be computed by estimating the required integer wordlength (*iwl*) and the required fractional wordlength (*fwl*). There exist both analytical and statistical methods to determine these wordlengths [15]. In general, *iwl* is estimated by computing the dynamic range of the data variable. In this work, we focus on the analysis of the dynamic range of different data variables. We extrapolate the information obtained from the computed dynamic ranges to understand the precision required in both the integer and fractional parts of the data representation.

Data dynamic range can be computed through floating point simulation of the application and statistical estimation of the data variable ranges. This gives a more accurate estimate, but since it is simulation-based, some possible cases can be overlooked leading to overflow problems. The second method is an analytical approach where the dynamic range of a particular output variable is expressed in terms of dynamic ranges of the inputs to that variable. In this method, dataflow modeling is useful for dynamic range analysis. This method guarantees the prevention of overflow, but is a worst-case estimate, thereby being more conservative. We adopt the latter approach so that all possible cases are taken into account.

Interval arithmetic theory [16] can be used to determine data dynamic range in this method. The dynamic range of each data item is obtained during traversal of the application dataflow graph with the help of propagation rules defined by interval arithmetic theory. Each operator or computation node has a defined propagation rule. One drawback of interval arithmetic is its susceptibility to over-estimation problems. In recent years, researchers have developed methods to overcome this problem (e.g., see [17, 18]). The flexibility of our proposed precision optimization methodology and the underlying DICE framework allows such methods to be incorporated and experimented with. Development of such extensions is a useful direction for further investigation.

3.3 DICE Unit Testing Framework

DICE includes a framework for implementation and execution of tests for software projects [10]. A major goal of the testing capabilities in DICE is to provide a lightweight and flexible unit testing environment. It is lightweight in that it requires minimal learning of new syntax or specialized languages, and flexible in that it can be used to test source code in any language, including C, Java, Verilog, and VHDL. This is useful in heterogeneous development environments so that a common framework can be used to test across all of the relevant platforms. However, the DICE unit testing framework is not restricted to unit testing or functional verification alone, but is adaptable for use in a wide range of simulation-based application exploration scenarios. This feature makes DICE well suited to studying implementation issues for signal processing systems.

The basic component of the DICE unit testing framework is a directory referred to as an *Individual Test Subdirectory* (*ITS*). A test suite in DICE consists of an arbitrary number of ITSs that test the different behaviors of the module under test (MUT).

An ITS includes the following required files, which provide a standard interface for testing that is independent of the underlying design language, target platform or development tools.

- A `makeme` script that contains all compilation steps required before running the test or analysis (e.g., a driver program that supplies the MUT with inputs and prints its outputs).
- A `runme` script that runs the test or analysis. The contents of `runme` may vary depending on the type of the MUT. Through appropriate programming of the `runme` file, the standard output of `runme` is in general highly configurable by the person who develops the test. Creative design of `runme` files can help to make more powerful and convenient test organizations within the DICE testing framework.
- A `correct-output.txt` file that contains the correct standard output that has to be produced by the test (i.e., after running the `runme` file).
- An `expected-errors.txt` file that contains the error messages that the test is expected to produce on standard error. This file is useful when the ITS checks for error conditions that the MUT should be detecting and reporting.

DICE encourages designers to think of a module interface in terms of streaming data primitives. Inputs and outputs are specified in the form of simple text files with streams of tokens. The basic DICE utility that makes use of the required files and exercises the test suite, called `dxtest`, recursively traverses all subdirectories executing `makeme`, followed by `runme`. It then compares the actual output generated after running `runme` with `correct-output.txt` and the actual standard error output with `expected-errors.txt`, finally producing a summary of successful and failed tests. This latter step is useful while testing, and is not used in precision analysis. In this work, we use the DICE unit testing framework as a simulation framework for precision analysis.

3.4 Dynamic Range Simulation with Functional DIF and DICE

In our approach to precision analysis, a library of functional DIF actors is created, which collectively compute the dynamic ranges of the different operations (vertices) in the application graph. These actors can be tested through the DICE unit testing framework. Since we are using model-based principles by describing the application

as a dataflow graph, we are able to reuse the same top-level representation for any model-based application analysis with or without simulation. By reusing the application graph, we save significant time in design exploration as the need for re-specifying the graph or rewriting the DIF file is avoided. Similarly, the library of functional DIF actors that compute dynamic ranges for different arithmetic operations can be used across different applications. This helps significantly to streamline the overall effort and design infrastructure that are devoted to precision analysis.

For dynamic range analysis in our approach, the ranges of values that can be assumed by the inputs to the application is specified in the same manner as the input test patterns of a unit test. This is done by inserting *File Readers*, which are functional DIF actors that read input values from text files to provide input samples for a simulation. Similarly, sink nodes are connected to *File Writers*, which are functional DIF actors that write outputs to text files. The dynamic range is computed by each actor based on the dynamic ranges of the inputs using interval arithmetic theory. Since the data ranges at all nodes have to be analyzed and not just at the sink nodes, we connect File Writers to the intermediate nodes that record this information. The input files are added to the ITS, and `dxtest` executes the simulation. The dynamic range at each node is captured in text files in the ITS. The `runme` script in the ITS can be configured for any post-processing on the outputs as demonstrated in the ensuing case study.

4. CASE STUDY — PRECISION ANALYSIS FOR JACOBI EIGENVALUE DECOMPOSITION

Eigenvalue decomposition (EVD) is used in a wide range of modern signal processing and communication applications, such as MIMO wireless communication, image recognition technologies, and direction of wave arrival estimation algorithms. In this case study, the EVD algorithm is implemented as part of a beamforming application inherent to MIMO wireless technology. The matrix of interest is a small and dense Hermitian matrix that characterizes the channel between each pair of transmit and receive antennas. The matrix sizes under consideration are 2×2, 4×4 and 8×8.

The focus of the study is to conduct a preliminary exploration of various bit precisions for EVD in order to provide a benchmark for system designers to help decide on the internal precision of their system given signal and noise variances and required output signal to noise ratio (SNR). The objective is to obtain the minimum required SNR in EVD by reducing the internal precision of the computation, thereby leading the way towards more efficient implementations (e.g., in terms of power consumption or cost).

4.1 Jacobi Eigenvalue Decomposition

Let $\mathbf{A}$ be a square $(N \times N)$ matrix with N linearly independent eigenvectors. Then $\mathbf{A}$ can be factorized as:

$$\mathbf{A} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$$

Here, $\mathbf{V}$ is the square $(N \times N)$ matrix whose ith column is the eigenvector q_i of $\mathbf{A}$, and $\mathbf{D}$ is the diagonal matrix whose diagonal elements are the corresponding eigenvalues, i.e., $\mathbf{D}_{ii} = \lambda_i$. This is known as the eigenvalue decomposition (EVD) or eigendecomposition of the matrix $\mathbf{A}$.

The Jacobi method is explored due to its efficiency with respect to small, dense matrices and its inherent parallelism. The Jacobi method diagonalizes the given matrix by systematically reducing the norm of the off-diagonal elements of the matrix. This is achieved by performing a series of Jacobi rotations, where each transformation is just a plane rotation designed to annihilate one of the off-diagonal matrix elements. This algorithm overwrites $\mathbf{A}$ with $\mathbf{V}^{\mathbf{T}}\mathbf{A}\mathbf{V}$, where $\mathbf{V}$ is orthogonal and $\mathbf{A}$ is increasingly diagonal.

4.2 Motivation for Precision Analysis

The Jacobi EVD algorithm is an iterative algorithm with each sweep of the algorithm propagating the errors of previous stages. Thus, a high degree of precision in both the signal and coefficients is required to minimize the effects of these propagated errors. In such an application, a full analysis of the data set is important to decide which type of data format to use, as this is essential to both the convergence of the algorithm, and the accurate computation of the eigenvalues and eigenvectors.

Algorithm 1 Pseudocode for the Jacobi EVD [12].

while offset(A) $> \varepsilon$ **do**
  for $p = 1$ to $n-1$ **do**
    for $q = p+1$ to n **do**
      Compute (v_1, v_2, θ)
      $A = J(p,q,\theta)^T A J(p,q,\theta)$
      $V = VJ(p,q,\theta)$
    end for
  end for
  Recalculate offset(A)
end while

Precision	2×2	4×4	8×8
Double	✓	✓	✓
Single	✓	×	×
PF (31,10)	✓	×	×
PF (31,8)	✓	×	×
PF (24,10)	✓	×	×
PF (24,8)	✓	×	×
PF (16,10)	✓	×	×
PF (16,8)	✓	×	×

Table 1: Convergence of Jacobi EVD implementation for different precisions.

Preliminary simulations test for the convergence of the Jacobi EVD for various precisions. The results are documented in Table 1. Here, the *pseudo floating point* (*PF*) format is a generalized floating point format that we define, where any real number can be represented as $Mantissa \times 2^{Exponent}$. The number of bits for the mantissa and exponent are given by I and E, respectively. By using this pseudo floating point format, all computations in the program are performed within the precision given by (I,E), thereby simulating a system with the given precision. In this work, the precisions of interest are all combinations from within $I = [16,31]$ and $E = [6,10]$.

Theoretically, the Jacobi EVD always converges [12]. However, due to insufficient precision leading to rounding and other errors, it is possible that the algorithm may not always converge with finite arithmetic implementations as observed in Table 1. This warrants a detailed analysis of the required precision at every step of the algorithm when mapping into an implementation.

4.3 Dataflow Model of the Jacobi EVD

In order to identify the parts of the algorithm that lead to non-convergence of an implementation, we first create a mixed-grained dataflow graph for the application using DIF. In this dataflow graph, the more precision-sensitive parts are represented with finer granularity actors (e.g. parts involving division or square root operations). These parts are decomposed into detailed dataflow representations of their internal computations, thereby exposing their computational structure in more depth. All of the graph vertices are implemented as *actors* within the functional DIF package. By performing appropriate analysis on the data propagating through this graph at each node, we can estimate the required precision at every node.

The Jacobi algorithm has two bounded loops to iterate over the rows and columns of the matrix, and one unbounded loop to execute the algorithm until a suitable solution within specified error bounds has been obtained. Since the base graph structure remains the same for all the iterations, we simulate the graph behavior for the required number of iterations by setting the iteration count in the simulator. We use a statistical estimate for the number of iterations of the unbounded loop. The dataflow graph for the Jacobi EVD for

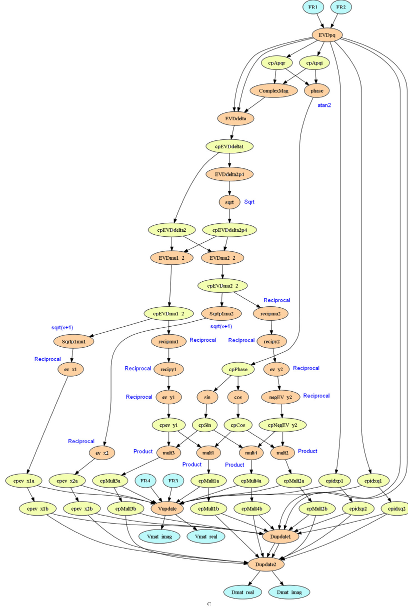

Figure 1: Dataflow graph for the 2x2 Jacobi EVD.

one iteration of the algorithm is shown in Figure 1.

4.4 Simulation with Functional DIF and DICE

In dataflow terminology, all of the actors in our Jacobi EVD model conform to *synchronous dataflow* (SDF) semantics, which means that on each execution, each actor produces constant amounts of data from its input ports and produces constant amounts of data on its output ports [8]. Only one iteration of the graph is required for a 2×2 matrix. Hence, the graph in Figure 1 with (p,q) as $(0,1)$ is the dataflow graph for the 2×2 Jacobi EVD. However, for the 4×4 and 8×8 matrices, the numbers of required iterations are set in the functional DIF simulator, and the reconfigurability of the `runme` file in DICE is applied to facilitate the feedback of output to input for successive iterations.

For example, the input files for data ranges for the input $\mathbf{A}$ matrix are `in-Areal.txt` and `in-Aimag.txt`. The output files for the $\mathbf{V}$ and $\mathbf{D}$ matrices are `out-Vreal.txt`, `out-Vimag.txt`, `out-Dreal.txt` and `out-Dimag.txt`. In order to create a feedback loop from the output to the input, the File Readers and File Writers read and write from intermediate files `Vreal.txt`, `Vimag.txt`, `Dreal.txt` and `Dimag.txt`. The `runme` file is programmed in such a way that the input files are first copied to the intermediate files, followed by the actual simulation before finally copying the intermediate files to the output files. This way the input files remain intact and are not overwritten by any intermediate results. By exercising model-based design integrated with appropriate simulation capabilities, the development process is largely automated and simplified, and the development time is significantly reduced. Ultimately, each actor has an associated text file consisting of the corresponding dynamic ranges.

4.5 Results and Discussion

The dynamic range simulation for an input matrix of size 2×2 is consistent with the results documented in Table 1. However, for the 4×4 matrix, Table 2 indicates multiple nodes with infinite dynamic range. The first actors that correspond to the infinite range are `recipmu1` and `recipmu2`, which calculate the dynamic range of a reciprocal operation on the outputs from `EVDmu12` and `EVDmu22`. Since the minimum possible values at `EVDmu12` and `EVDmu22` are identically 0, the values at `recipmu1` and `recipmu2` are unbounded.

Node	Computation	Dynamic Range
ComplexMag	$\sqrt{a^2+b^2}$	$[1.49\times10^{-8}, 4.78\times10^{14}]$
EVDdelta	$(a-b)/c$	$[-4.65\times10^{22}, 4.65\times10^{22}]$
EVDdelta2p4	a^2+4	$[4, 2.17\times10^{45}]$
sqrt	$\sqrt{x}$	$[2, 4.65\times10^{22}]$
EVDmu12	$(\sqrt{\delta^2+4}-\delta)^2$	$[0, 2.16\times10^{45}]$
EVDmu22	$(\sqrt{\delta^2+4}+\delta)^2$	$[0, 2.16\times10^{45}]$
Sqrtp1mu1	$\sqrt{x+1}$	$[1, 4.65\times10^{22}]$
Sqrtp1mu2	$\sqrt{x+1}$	$[1, 4.65\times10^{22}]$
ev-x1, ev-x2	$1/x$	$[2.15\times10^{-23}, 1]$
recipmu1	$1/x$	$[0, \infty]$
recipmu2	$1/x$	$[0, \infty]$
recipy1	$\sqrt{x+1}$	$[0, \infty]$
recipy2	$\sqrt{x+1}$	$[0, \infty]$
ev-y1, ev-y2	$1/x$	$[0, 1]$
negev-y2	$-x$	$[-1, 0]$
mult1 - mult4	$a*b$	$[-1, 1]$
Vupdate	$\mathbf{AB}$	$[-3.51\times10^{6}, 1.76\times10^{6}]$
Dupdate1, 2	$\mathbf{AB}$	$[-1.04\times10^{15}, 1.04\times10^{15}]$

Table 2: Dynamic ranges for computations in 4×4 Jacobi EVD.

The actor `EVDmu12` computes the dynamic range of the operation $(\sqrt{\delta^2+4}-\delta)^2$. Mathematically speaking, this expression should always be greater than 0 because it is a squaring operation, and $\sqrt{\delta^2+4}\neq\delta$. Because the minimum value at this node is 0 when it should not be so, further operations in the application yield values of ∞, thereby leading to many incorrect computations and loss of convergence.

On closely observing the flow of the data in this part of the graph, and the respective dynamic ranges, it can be seen that this happens when δ assumes very high values. These operations correspond to equations (2) and (3). As the number of iterations increases in the Jacobi EVD algorithm, the off-diagonal elements (parameter b in the equation for δ) tend to 0. Therefore, as the number of iterations increases, $\delta\rightarrow\infty$. In turn, as $\delta\rightarrow\infty$, we observe that $\sqrt{\delta^2+4}\approx\delta$ and $\sqrt{\delta^2+4}-\delta\rightarrow0$. In case of insufficient precision, this difference becomes exactly 0, leading to incorrect computations further on. The same happens when $\delta<0$ with `EVDmu22`, which computes $(\sqrt{\delta^2+4}+\delta)^2$.

$$v_1=\begin{bmatrix}\frac{e^{j\theta}}{\sqrt{1+\frac{1}{|\mu_1|^2}}}\\ \frac{1}{\sqrt{1+|\mu_1|^2}}\end{bmatrix}\qquad v_2=\begin{bmatrix}\frac{-e^{j\theta}}{\sqrt{1+\frac{1}{|\mu_2|^2}}}\\ \frac{1}{\sqrt{1+|\mu_2|^2}}\end{bmatrix}\tag{1}$$

$$\mu_1=\frac{2}{\sqrt{\delta^2+4}-\delta}\qquad \mu_2=\frac{2}{\sqrt{\delta^2+4}+\delta}\tag{2}$$

$$\delta=\frac{a-c}{|b|}\qquad \theta=\tan^{-1}\left[\frac{Im(b)}{Re(b)}\right]\tag{3}$$

By using our application exploration framework and carefully adopting fundamental principles of precision analysis, we have identified the source of precision loss in the Jacobi EVD. These operations require higher precision for accurate behavior across the entire data set. However, by identifying solutions to the source of the precision loss problem, and verifying them, we can provide useful feedback to DSP software designers to optimize the implementation. We achieve this through reformulation of the equation for μ_1 in (2) in such a way as to avoid the difference operation. When $\delta>0$, μ_2 can be computed without any precision loss due to the presence of the addition operation. On close inspection of the equations in (2), it can be seen that μ_1 can be expressed in terms of μ_2, thereby avoiding the difference operation.

$$\mu_1\mu_2=\left(\frac{2}{\sqrt{\delta^2+4}-\delta}\right)\left(\frac{2}{\sqrt{\delta^2+4}+\delta}\right)=\frac{4}{(\sqrt{\delta^2+4})^2-\delta^2}=1$$

Node	Original Dyn Range	New Dynamic Range
EVDmu12	$[0, 2.16 \times 10^{45}]$	$[10^{-45}, 2.16 \times 10^{45}]$
recipmu1, recipmu2	$[0, \infty]$	$[4.6 \times 10^{-46}, 10^{-45}]$
recipy1, recipy2	$[1, \infty]$	$[1, 4.65 \times 10^{22}]$
ev-y1, ev-y2	$[0, 1]$	$[2.15 \times 10^{-23}, 1]$
negev-y2	$[-1, 0]$	$[-1, -2.15 \times 10^{-23}]$

Table 3: Dynamic ranges for 4×4 Jacobi EVD with reformulation.

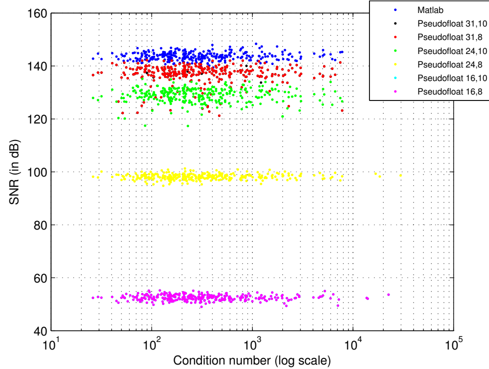

Figure 2: SNR vs. condition number for 4×4 Hermitian matrix.

$$\mu_1 = 1/\mu_2 = (\sqrt{\delta^2 + 4} + \delta)/2$$

This theoretically eliminates the root of the precision problem, and is a useful optimization to the algorithm. In order to verify this new formulation, the actors for computing the dynamic range of μ_1 and μ_2 were accordingly rewritten and the dynamic range simulation with functional DIF was repeated. The new ranges obtained are all within $[-10^{46}, 10^{46}]$ and can thus be implemented with pseudo floating point with at least $E = 10$. However, since this analysis is conservative, it may be possible to implement this algorithm with $E < 10$, which we confirm with our C-based implementation.

This analysis was verified with the C-based implementation by rewriting the code segment corresponding to the computation of $\mu_{1,2}$. The new implementation converged for all of the precisions under consideration and produced valid results for all configurations with $E \geq 7$. The plot in Figure 2 shows the SNR of the reconstructed matrix after EVD as a function of the independent parameter, the matrix condition number, for both MATLAB and C for various data formats. By using DIF to model the EVD and functional DIF to prototype the dynamic range analysis of this application in the DICE framework, we have demonstrated how dataflow modeling and DICE synergistically facilitate high level application exploration. DICE's highly flexible and reconfigurable framework enables it to be used in various stages of application development, and is especially well-suited for dataflow based implementations.

5. CONCLUSION

In this work, we have demonstrated model-based precision analysis with an exploration study into the performance versus precision metrics for Jacobi Eigenvalue Decomposition. We modeled the application graph and the precision analysis with DIF and functional DIF, and executed the entire analysis by appropriately configuring the DICE unit testing framework. Although the aim was to perform an analysis and not testing or verification per se, simulation of the application graph was still required with external inputs, making DICE a convenient framework for this exploration study. DICE enjoys a high level of synergy with DIF, our model-based development environment, in high level application exploration and in the seamless integration of testing with design.

We were able to analyze the dynamic ranges at different nodes in the application graph, and hence identify the nodes that required higher precision than what was available. By reformulating the mathematical expressions for these operations, we reliably circumvented this problem and provided relevant feedback to the algorithm developers. This case study is a demonstration of the use of dataflow modeling in early stage application exploration, and the use of DICE in the overall design flow. With this case study, the integration of DICE with a model-based approach was highlighted to make the application design process more efficient, yet still rigorous and evolvable.

REFERENCES

[1] G. Frantz and R. Simar, "Comparing fixed and floating point DSPs," Tech. Rep. SPRY061, Texas Instruments, 2004.

[2] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*, Springer, 2010.

[3] C. Hsu and S. S. Bhattacharyya, "Porting DSP applications across design tools using the dataflow interchange format," in *Proc. RSP 2005*, Montreal, Canada, June 2005, pp. 40–46.

[4] D. Menard, D. Chillet, F. Charot, and O. Sentieys, "Automatic floating-point to fixed-point conversion for DSP code generation," in *Proc. CASES 2002*, Grenoble, France, October 2002, pp. 270–276.

[5] H. Keding, M. Willems, M. Coors, and H. Meyr, "FRIDGE: A fixed-point design and simulation environment," *DATE 1998*, p. 429, 1998.

[6] P. Belanovic and M. Rupp, "Automated floating-point to fixed-point conversion with the fixify environment," in *Proc. RSP 2005*, Washington DC, USA, 2005, pp. 172–178.

[7] A. A. Gaffar, W. Luk, P. Y. K. Cheung, and N. Shirazi, "Customising floating-point designs," in *Proc. FCCM 2002*, Washington DC, USA, 2002, p. 315.

[8] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Transactions on Computers*, February 1987.

[9] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[10] S. S. Bhattacharyya, S. Kedilaya, W. Plishker, N. Sane, C. Shen, and G. Zaki, "The DSPCAD integrative command line environment: Introduction to DICE version 1," Tech. Rep. UMIACS-TR-2009-13, Institute for Advanced Computer Studies, University of Maryland at College Park, August 2009.

[11] W. Plishker et al., "Model-based DSP implementation on FPGAs," in *Proc. RSP 2010*, Fairfax, Virginia, June 2010, pp. 8–11.

[12] G. H. Golub and C. F. Van Loan, *Matrix Computations*, The Johns Hopkins University Press, 3rd edition, 1996.

[13] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *Proc. ICASSP 1993*, April 1993.

[14] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proc. RSP 2008*, Monterey, California, June 2008, pp. 17–23.

[15] K. Han and B. L. Evans, "Optimum wordlength search using sensitivity information," *EURASIP Journal on Applied Signal Processing*, vol. 2006, pp. 76, 2006.

[16] R. B. Kearfott, "Interval computations: Introduction, Uses and Resources," *Euromath Bulletin*, vol. 2, pp. 95–112, 1996.

[17] L. H. de Figueiredo and J. Stolfi, "Affine arithmetic: Concepts and Applications," *Numerical Algorithms*, vol. 37, no. 1, pp. 147–158, 2004.

[18] K. Makino and M. Berz, "Efficient control of the dependency problem based on Taylor model methods," *Reliable Computing*, vol. 5, pp. 3–12, 1999.