DESIGN METHODOLOGIES FOR HIGH-PERFORMANCE SIGNAL PROCESSING SYSTEMS

by

Soujanya A. Kedilaya

Advisor: Professor Shuvra S. Bhattacharyya

University of Maryland, College Park

Report submitted to the Salzburg University of Applied Sciences, Salzburg, Austria in partial fulfillment of the requirements for the Marshall Plan Scholarship
2010

Acknowledgments[1]

Foremost, I would like to thank my advisor, Professor Shuvra Bhattacharyya for his inspiration and guidance through out the course of my research.

I am indebted to the Austrian Marshall Plan Foundation and the Salzburg University of Applied Sciences (FHS) in Salzburg, Austria for providing me with a truly wonderful opportunity to spend a semester in Salzburg where I was exposed to a whole new school of thought. I wish to thank Professor Bhattacharyya and Professor Gabriele Abermann of FHS for facilitating this exchange. I would like to acknowledge Professor Gerhard Jöchtl and Simon Kranzer for the interesting discussions and the Deutsch-English translations, and the wonderful company of Bernadette, Andreas, Peter and the other staff members of the University who made my stay in Austria unforgettable. Danke schön!

I am also grateful to the Department of Electrical and Computer Engineering at University of Maryland College Park, and Texas Instruments, Germantown for selecting me as a Texas Instruments Scholar for the year 2009-2010. At TI I had the invaluable experience of working on excellent industry projects with the best of minds. I would like to thank Mr. Brian Johnson, Mr. Aleksander Purkovic and Ms. Mingjian Yan for their support and mentoring, which have been crucial to my research.

My special thanks go out to Dr. William Plishker for his thoughtful advice, which often served as a sense of direction during my research work.

[1]This research was supported in part by the Austrian Marshall Plan Foundation and Texas Instruments.

Table of Contents

Chapter 1

Introduction

System development for digital signal processing (DSP) applications often involves an initial application description in a design environment, which is then manually transcoded and tuned to target the final design platform. Often separated by languages, tools, and even different teams, going from an initial application description to a final implementation tends to be a manual, error-prone, and time-consuming problem. To improve the quality and performance while reducing development time, a cross platform design environment is needed that accommodates both early design exploration and final implementation tuning.

The DSPCAD Integrative Command Line Environment (DICE) [1] [2] is a realization of managing these enhancements to the design flow. It is a framework for facilitating efficient management of design and test of cross-platform software projects. DICE defines platform and language independent conventions for describing and organizing tests, facilitating high portability of tests for cross-platform operation. Although DICE can be used for any type of software development, it makes a natural fit with dataflow models due to the streaming nature of inputs and outputs supported by DICE.

We use the Dataflow Interchange Format (DIF) [3] as our dataflow analysis engine which can leverage the extracted dataflow models, and as our model-based

development environment. Although DIF and DICE are orthogonal to each other (one can exist without the other), our research explores novel synergies between them, such as integrating testing with design to continuously identify and correct errors; generating automatic testbenches for improving the ease with which cross-platform tests are created; and using DICE as a framework to simulate systems modeled in DIF, and explore design trade-offs, component interactions, and system-level metrics. We have demonstrated the novel test framework of DICE for a cross-platform project in [4].

In this work we present a case study which is a demonstration of the use of dataflow modeling in early stage application exploration and the use of DICE in the overall design flow. We do an exploration study into the internal precision of computation for the Jacobi Eigenvalue Decomposition (EVD) [5]. Due to the mathematically intensive nature of the computations in this algorithm, it becomes important to comprehensively analyze the required precision at every step of the algorithm. We do this analysis by modeling the Jacobi EVD as a mixed-grain dataflow graph in DIF. We not only verify the functional correctness of the EVD algorithm, but also further demonstrate the synergy between DIF and DICE when analyzing the data dynamic range of the intrinsic computations by reusing the same application graph. Based on this analysis, we are able to provide useful feedback to the low-level designers about the formulation of some parts of this algorithm.

Chapter 2

Background

To give context to our model based testing approach, this section covers the dataflow models that we base our technique on, as well as the dataflow modeling tool we utilize for application description.

2.1 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) [6], single-rate dataflow, homogeneous synchronous dataflow (HSDF), or a more complicated model such as boolean dataflow (BDF) [7].

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 $v_2 \in V$. Nodes or *actors* represent computations while edges represent

a FIFO communication links between them.

2.2 Dataflow Interchange Format

To describe dataflow applications for this wide range of dataflow models, application developers can use the Dataflow Interchange Format (DIF) [3], a standard language founded in dataflow semantics and tailored for DSP system design. DIF is suitable as an interchange format for different dataflow-based DSP design tools because it provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification [8]. From a dataflow point of view, DIF is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related and actor-specific information.

The dataflow semantic specification is based on dataflow modeling theory and independent of any design tool. Therefore, the dataflow semantics of a DSP application is unique in DIF regardless of any design tool used to originally enter the application specification. Moreover, DIF also provides syntax to specify design-tool-specific information, which is captured within the data structures associated with DIF intermediate representations.

2.2.1 The DIF Package (TDP)

To utilize the semantics captured by describing applications in the DIF language, the DIF package was created. An overview is illustrated in Figure 2.1 (for a

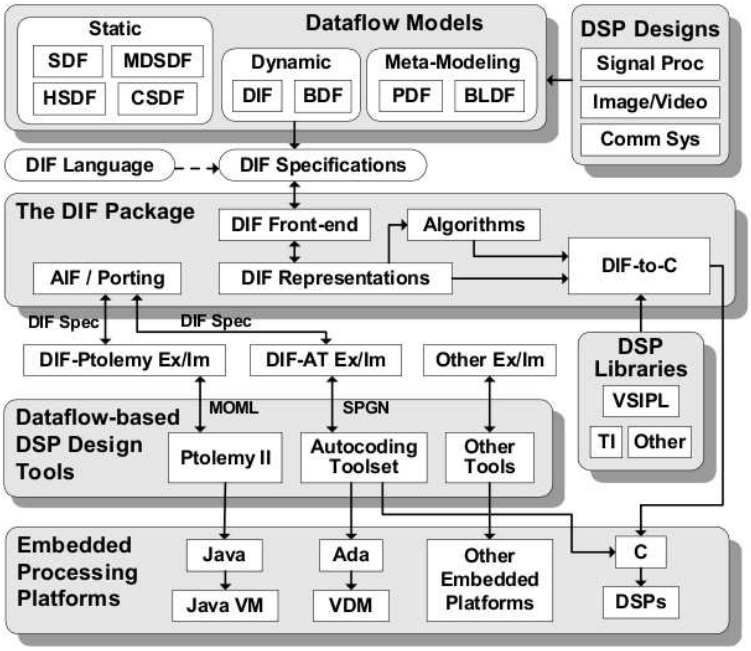

Figure 2.1: DIF-based design flow.[3]

full explanation of it, see [3]). Along with the ability to transform a DIF description into a manipulatable internal representation, the DIF package contains graph utilities, optimization engines, and algorithms that can prove useful properties of an application. These facilities make the DIF package an effective environment for modeling dataflow applications, providing interoperability with other design environments and developing new tools.

2.3 Functional DIF

To quickly arrive at quality prototypes, designers must be able to describe their complex applications in a single environment. In the context of dataflow programming, this involves describing not only the top level connectivity and hierarchy

of the application graph, but also the functionality of the graph actors (the functional modules that correspond to the non-hierarchical graph vertices), preferably in a natural way that integrates with the semantics of the dataflow model they are embedded in. Once the application is appropriately captured, designers need to be able to evaluate static schedules (for high performance) alongside dynamic behavior without losing semantic ground. With a properly-constructed schedule and a fully-described application, designers should be able to verify the functionality of a dataflow-based system. With such a feature set, designers should arrive at quality prototypes faster.

The functional DIF [9] (DIF with functional designs) package enables fast simulation and prototyping of scheduling strategies. Prototyping in functional DIF is useful because it not only allows one to rapidly validate the overall functionality and high level dataflow architecture of a design, but also allows for a much faster simulation of complete system functionality.

In this work, we would like to not only model the application description but also have functional simulation for which we utilize functional DIF. The semantic foundation of functional DIF is *core functional dataflow* (CFDF) [9], which is capable of expressing deterministic, dynamic dataflow applications. In this formalism, each actor $a \in V$ has a set of *modes*, M_a, in which it can execute. Each mode, when executed, consumes and produces a fixed number of tokens. In the context of the work presented in this thesis, we use only the SDF model of computation in which the actors have only one mode. We use this structured representation of functionality to derive the appropriate dataflow testbench for each actor.

Chapter 3

DSPCAD Integrative Command Line Environment (DICE)

3.1 Introduction

DICE (the DSPCAD Integrative Command Line Environment) [1] is a package of utilities that facilitates efficient management of software projects. The objective of DICE is to provide a flexible, light-weight environment for the research, development, testing, and integration of software projects, particularly those that employ heterogeneous programming languages or models of computation. DICE is not meant to replace existing software development tools. Instead it is a command line solution to utilize the existing tools more effectively, especially for cross-platform design.

DICE is implemented as a collection of utilities that are in the form of bash scripts, C programs, and python scripts. The package is intended for cross- platform operation, and is currently being developed and used actively on the Windows (equipped with Cygwin), Solaris, and Linux platforms.

DICE includes a variety of utilities to help improve productivity while working in a command-line or shell-based project development environment. Since navigation and relocating files and directories inside or across complex project directory structures can be tedious and prone to errors, DICE provides a set of utilities for efficient navigation through directories and to easily move files and folders between

different directories.

3.2 DICE Unit Testing Framework

DICE includes a framework for implementation and execution of tests for software projects. Although the emphasis in this framework is on unit tests, and therefore, it is often referred to as the DICE unit testing framework, the framework can also be applied to testing at higher levels of abstraction, including subsystem- and system-level testing.

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.

The basic component of the DICE unit testing framework is a directory referred to as an *Individual Test Subdirectory* (ITS). The test suite consists of several ITSs that test the different behaviors of the MUT. Every ITS name must start with the prefix "test" (e.g., test01, test02, test-square-matrix-1, test-square-matrix-2, etc.). By doing so, changing the ITS prefix to any word other than "test" will exclude it from the test suite.

An ITS consists of the following required files:

- A *readme.txt* file that contains an explanation of what part of the MUT func-

tionality this ITS tests. This is useful for the proper documentation of all the tests.

- A *makeme* script that contains all compilation steps required before running the test. It is important to note that *makeme* does not compile the source code of the MUT, but it compiles any additional code required for the test (e.g., a driver program that supplies the MUT with inputs and prints its outputs).

- A *runme* script that runs the test. The contents of *runme* may vary depending on the type of the MUT. For example, when testing a C program, one may need to just run an object file, but for a Verilog module, a hardware simulator such as ModelSim may need to be run. Also the *runme* file may contain a call to other executables that perform different post processing on the MUT output before doing the comparison with correct-output and expected-error files.

- 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 the standard error. This file is useful when the ITS checks for the errors that the MUT should be catching.

The basic DICE utility that makes use of the required files and exercises the test suite is called *dxtest*. By running *dxtest* from a certain directory, it recursively traverses all subdirectories that begin with the prefix "test". A subdirectory that

contains a *runme* file is considered as an ITS. When *dxtest* traverses an ITS, it first executes the *makeme*, followed by the *runme*. It then compares the actual output generated after running *runme* with the *correct-output.txt* and the actual standard error output with the *expected-errors.txt*. After traversing all the subdirectories, a summary of successful and failed tests is produced.

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. We demonstrate this in Chapter 4.

Chapter 4

Case Study - Precision Analysis of the 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, direction of wave arrival estimation algorithms etc. In the context of this work, the EVD algorithm is being implemented as part of a beamforming application inherent to MIMO wireless technology.

4.1 EVD in MIMO wireless technology

With the wireless community engaged in the research and development of the fourth generation (4G) of wireless cellular systems, various schemes are being explored to achieve the data rate requirements for 4G. Multiple-input multiple-output (MIMO) wireless communication has been one of the most promising technologies for improving the spectrum efficiency of wireless systems. MIMO schemes enable a variety of functions including multi-stream transmission for high spectrum efficiency, improved link quality through diversity mechanisms, and adaptation of radiation patterns for signal gain and interference mitigation through adaptive beamforming [10].

When a signal $\mathbf{x}$ is transmitted through a MIMO channel with channel gain

matrix $\mathbf{H}$, the received signal $\mathbf{y}$ can be modeled as,

$$\mathbf{y} = \mathbf{H}\mathbf{x} + \mathbf{n}$$

where $\mathbf{n}$ is the noise experienced by the receivers.

When omni-directional antennas are used at the basestation, the transmission/reception of each user's signal becomes a source of interference to other users located in the same cell, making the overall system interference limited. Beamforming is a technique where each user's signal is multiplied with a beamforming vector with complex weights that adjusts the magnitude and phase of the signal to and from each antenna. This causes the output from the array of antennas to form a transmit/receive beam in the desired direction and minimizes the output in other directions. By transmitting in the direction of the eigenvector corresponding to the largest eigenvalue of the positive semi-definite matrix $\mathbf{H}^{\dagger}\mathbf{H}$, the signal-to-noise ratio (SNR) at the receiver is maximized [11]. More generally, vector information could be sent along all of the eigenchannels of $\mathbf{H}^{\dagger}\mathbf{H}$ as described in [12], resulting in increased spectral efficiency. [13] proposes the methodology of *eigenbeamforming* where the transmit beamforming vector is chosen as the eigenvector corresponding to the largest eigenvalue of the matrix given by $\frac{1}{K}\sum_{k=1}^{K}\mathbf{H}^{\dagger}(k)\mathbf{H}(k)$, where K is the number of sub-carriers. Eigenvalue decomposition is thus used in beamforming and MIMO systems to compute the eigenvalues and corresponding eigenvectors of $\mathbf{H}^{\dagger}\mathbf{H}$.

4.2 Eigenvalue Decomposition

A (non-zero) vector $\mathbf{x}$ of dimension N is an eigenvector of a square $(N \times N)$ matrix $\mathbf{A}$ if and only if it satisfies the linear equation

$$\mathbf{Ax} = \lambda\mathbf{x}$$

where λ is a scalar, termed the eigenvalue corresponding to $\mathbf{x}$.

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

$$\mathbf{A} = \mathbf{VDV^{-1}} \tag{4.1}$$

$\mathbf{V}$ is the square $(N \times N)$ matrix whose i-th 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 matrix of interest in this case is a 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. In this method, we explore the Jacobi method due to its efficiency with respect to small, dense matrices and its inherent parallelism.

4.3 The Jacobi idea

The Jacobi method diagonalizes the given matrix by systematically reducing the norm of the off-diagonal elements of the matrix given by the quantity,

$$off(\mathbf{A}) = \sqrt{\sum_{i=0}^{n} \sum_{j=1, j\neq i}^{n} a_{ij}^2}$$

This is achieved by performing a series of Jacobi Rotations (similar to the Givens rotations) where each transformation (a Jacobi rotation) is just a plane rotation designed to annihilate one of the off-diagonal matrix elements. Successive transformations undo previously set zeros, but the off-diagonal elements nevertheless get smaller and smaller, until the matrix is diagonal to machine precision. The Jacobi rotation matrix is given by the following equation.

$$J(p,q,\theta) = \begin{array}{c} \\ \\ p \\ \\ q \\ \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & p & & q & \\ \end{array} \\ \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & v_{11} & \dots & v_{21} & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & v_{12} & \dots & v_{22} & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix} \end{array}$$

For a matrix $\begin{pmatrix} a & b \\ b^* & c \end{pmatrix}$ where a and c are positive real-valued, (v_1, v_2) can be computed as shown in equation (4.2).

Algorithm 1 Pseudocode for the Jacobi EVD

```
while offset(A) > ε do
  for p = 1 to n − 1 do
    for q = p + 1 to n do
      Compute (v1, v2)
      A = J(p, q, θ)^T A J(p, q, θ)
      V = V J(p, q, θ)
    end for
  end for
  Recalculate offset(A)
end while
```

$$v_1 = \begin{bmatrix} \frac{e^{j\theta}}{\sqrt{1+\frac{1}{|\mu_1|^2}}} \\ \frac{1}{\sqrt{1+|\mu_1|^2}} \end{bmatrix} \qquad v_2 = \begin{bmatrix} \frac{-e^{j\theta}}{\sqrt{1+\frac{1}{|\mu_2|^2}}} \\ \frac{1}{\sqrt{1+|\mu_2|^2}} \end{bmatrix} \tag{4.2}$$

$$\mu_1 = \frac{2}{\sqrt{\delta^2+4}-\delta} \qquad \mu_2 = \frac{2}{\sqrt{\delta^2+4}+\delta} \tag{4.3}$$

$$\delta = \frac{a-c}{|b|} \qquad \theta = \tan^{-1}\left[\frac{Im(b)}{Re(b)}\right] \tag{4.4}$$

Every iteration results in $off(\mathbf{A})^2 = off(\mathbf{A})^2 - 2a_{pq}^2$. In this sense, $\mathbf{A}$ moves closer to the diagonal form with each Jacobi step. This algorithm overwrites $\mathbf{A}$ with $\mathbf{V^T AV}$ with $\mathbf{V}$ being orthogonal and $\mathbf{A}$ being increasingly diagonal.

4.4 Precision Analysis

4.4.1 Motivation for Precision Analysis

The goal is to conduct an initial exploration study of various bit precisions for eigenvalue decomposition 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 SNR. The focus of the study is to obtain the minimum required signal to noise ratio (SNR) in eigenvalue decomposition by reducing the internal precision of the computation.

Due to the mathematically sensitive nature of the Jacobi EVD algorithm, the need was identified for a more thorough analysis on the data precision at every step of the algorithm. This naturally led to the representation of the algorithm as a fine-grained data flow graph where each node represented a basic block of computation (Sec. 4.5). We use DIF to model the dataflow of the Jacobi EVD and DICE as the framework within which the precision analysis is carried out.

4.4.2 Functional Simulation in C

The Jacobi EVD was implemented in double precision, single precision and pseudo floating point formats to analyze the performance of the algorithm as a function of precision. Here the pseudo floating point 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 appropriately setting the values of I and E, the internal bit precision of the numbers can be controlled. In this work, precisions of interest are all combinations from within $I = 16, 24, 31$ and $E = 6, 8, 10$. Preliminary simulations test for the convergence of the Jacobi EVD for all precisions. The results are documented in Table 4.1.

The double precision floating point implementation of the Jacobi EVD converged for all required matrix sizes, and the implementations for all precisions con-

Precision	2×2	4×4	8×8
Double	Converges	Converges	Converges
Single	Converges	Does not converge	Does not converge
Pf (31,10)	Converges	Does not converge	Does not converge
Pf (31,8)	Converges	Does not converge	Does not converge
Pf (31,6)	Converges	Does not converge	Does not converge
Pf (24,10)	Converges	Does not converge	Does not converge
Pf (24,8)	Converges	Does not converge	Does not converge
Pf (24,6)	Converges	Does not converge	Does not converge
Pf (16,10)	Converges	Does not converge	Does not converge
Pf (16,8)	Converges	Does not converge	Does not converge
Pf (16,6)	Converges	Does not converge	Does not converge

Table 4.1: Convergence of Jacobi EVD implementation for all precisions

verged for matrix size of 2×2. However, the overall results were well below expectations as the implementation did not converge for any precision configuration other than double floating point for 4×4 and 8×8 matrices. This was indeed largely unsatisfactory as some of the considered precisions offer large dynamic ranges and fractional word lengths sufficient for most sensitive applications. This warranted a much more detailed analysis of the required precision at every step of the algorithm.

4.5 Dataflow model of the Jacobi EVD

Following the convergence issues with the initial implementation, there arises a need to identify the parts of the algorithm that leads to the non-convergence of the implementation. An intuitive way to do this would be to cleverly partition the algorithm into smaller computation nodes and represent the algorithm as a dataflow graph. By doing appropriate analysis at every node on the data propagating through

this graph, we can estimate the required precision at every node.

4.5.1 Related Work

Dataflow modeling has often been used in such precision analysis, most commonly in automatic floating to fixed point conversion of programs ([14], [15], [16]). Some of these research works like [14], [17], [18] use fine-grained dataflow graphs as an intermediate representation between the floating- and fixed-point programs. In this intermediate representation, the dataflow graph has nodes representing the operations and the variables as edges. Using this dataflow graph as the backbone, several statistical and/or analytical methods are applied at every node to compute and annotate the nodes with their respective dynamic ranges, binary point positions, and ultimately bit widths. We adopt some of these methods to analyze the data set of the Jacobi EVD, and identify the computations in the algorithm that require more precision.

4.5.2 Dataflow model

DIF has been used to model the dataflow graph for the Jacobi EVD. In constructing this graph, we identify the operations in the algorithm that are more sensitive to precision and make them individual nodes in the graph. Such operations typically include square root, division etc. There are many such occurrences in the Jacobi algorithm for eigenvalue computation. All the nodes are implemented as *actors* within the functional DIF package.

An important point of consideration in constructing the dataflow graph is the

presence of unbounded and bounded loops in the algorithm. Normally the graphs can be unrolled for bounded loops. However, since the base graph structure remains the same for all the iterations, we make use of functional DIF's CFDF simulator capabilities in simulating the graph behavior for the required number of iterations. The dataflow graph for the Jacobi EVD for one iteration of the algorithm is shown in Figure 4.1.

Only one iteration of the graph is required for a 2×2 matrix. Hence, the graph in Figure 4.1 with (p, q) as $(0, 1)$ is the dataflow graph for the 2×2 Jacobi EVD. For 4×4 and 8×8 matrices, the graph in Figure 4.1 is iteratively simulated 24 and 140 times respectively, with each iteration having a different (p, q) index. The output matrices of each iteration will be the input of the next iteration. DICE is used to facilitate this configuration, by correspondingly programming the *runme* file.

4.6 Dynamic Range Analysis

In our work, we concentrate 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 part of the data representation without directly calculating the wordlengths.

An analytical approach is employed where the dynamic range of a particular output variable is expressed in terms of dynamic ranges of the inputs to that node. In this method, dataflow modeling is useful for dynamic range analysis. This method guarantees no overflow, but is a worst-case estimate thereby being more conservative.

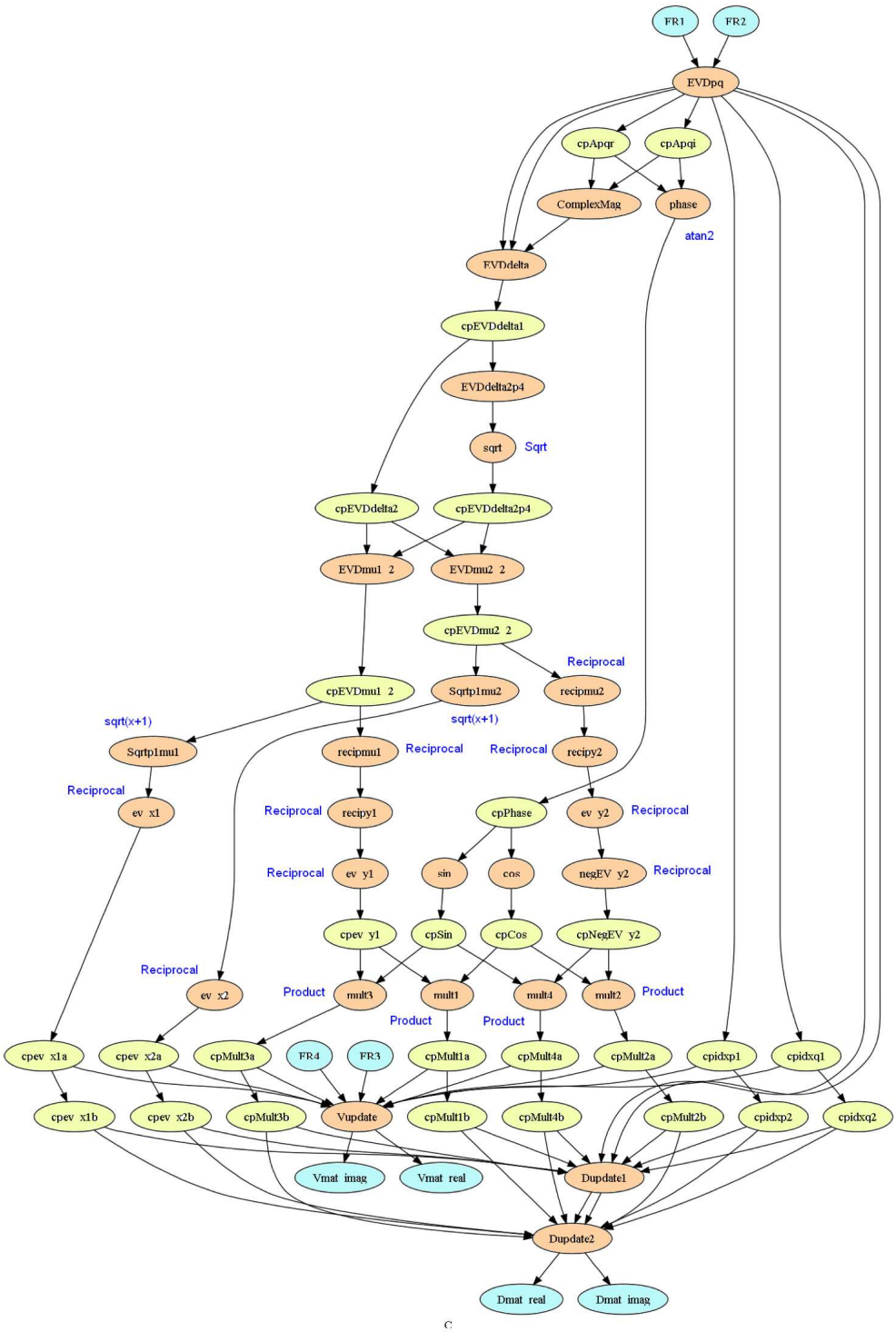

Figure 4.1: Dataflow graph for the 2x2 Jacobi EVD

For our application, it is more suitable to adopt this approach so that all possible cases are taken into account.

Interval Arithmetic theory [19] can be used to determine data dynamic range in this method. Interval arithmetic is an arithmetic defined on sets of intervals, rather than sets of real numbers. The dynamic range of each data is obtained during the traversal of the application graph with the help of propagation rules defined by interval arithmetic theory. Each operator or computation node has a defined propagation rule. Table 4.2 enlists the interval computations for the basic arithmetic operations most commonly used operations in the Jacobi EVD.

Operation	**Interval Computation**
Addition	$[a,b]+[c,d]=[a+c,b+d]$
Subtraction	$[a,b]-[c,d]=[a-d,b-c]$
Multiplication	$[a,b]\times[c,d]=[min(ac,ad,bc,bd),max(ac,ad,bc,bd)]$
Division	$[a,b]\div[c,d]=[min(a/c,a/d,b/c,b/d),max(a/c,a/d,b/c,b/d)]$, $0\notin[c,d]$
Squaring	$[a,b]^2=[a^2,b^2]$, if $a\geq 0$
	$[a,b]^2=[b^2,a^2]$, if $b<0$
	$[a,b]^2=[0,max(a^2,b^2)]$ otherwise
Square root	$[a,b]^{1/2}=[\sqrt{a},\sqrt{b}]$

Table 4.2: Interval Arithmetic

4.6.1 Dynamic range simulation with functional DIF and DICE

A library of functional DIF actors are written corresponding to the nodes in the application graph. All these actors have a single *mode* and are therefore SDF

with constant production and consumption rates. They are tested through the DICE unit testing framework. The application graph shown in Figure 4.1 is verified for functional correctness using the CFDF simulator and sample test patterns with appropriate *correct-output.txt* files.

For dynamic range analysis, the same application graph is used but a parallel library of actors is created corresponding to each node. This time each actor calculates the dynamic range of the corresponding operation using interval arithmetic's propagation rules. All the actor properties remain the same in terms of their models but the production and consumption rates double wherever applicable because for each data variable, there are now two values - the minimum and maximum values of the range.

The DICE unit testing framework is not restricted to unit testing or functional verification alone, but is flexible to be used for any simulation-based application exploration. For the dynamic range analysis, the range of values of the input matrix is specified in a similar fashion as the input test patterns of a unit test by hooking in *File Readers* and the final outputs from the **V** and **D** matrices are hooked into *File Writers*. *File Readers* and *File Writers* are also functional DIF actors that read and write input and output tokens respectively from text files.

The dynamic range is computed by each actor based on the dynamic ranges of the inputs. For the functional verification of the whole application, outputs from the sink nodes alone are required. For dynamic range analysis, outputs from all intermediate nodes are required as well. To facilitate this, *File Writers* are hooked into each computation node. Ultimately each node has an associated output file

Data Format	Number of bits	Approx. Dynamic Range
Double precision	I=53, E=11	-10^{308} to 10^{308}
Single precision	I=24, E=8	-10^{38} to 10^{38}
Pseudo float	I=31/24/16, E=10	-10^{154} to 10^{154}
	I=31/24/16, E=8	-10^{38} to 10^{38}
	I=31/24/16, E=6	-10^{9} to 10^{9}

Table 4.3: Dynamic ranges for various data formats

consisting of the corresponding dynamic range from every iteration.

4.7 Results and Discussion

From the dynamic range simulation for input matrix of size 2×2, it is observed that the results from the computations do not exceed 10^{10}. From Table 4.3, it can be inferred that the Jacobi EVD for a 2×2 matrix should produce valid results for all precisions under consideration except when $E = 6$ in the pseudo floating-point representation. This is in agreement with the results obtained in Sec. 4.4.2.

However, for the 4×4 matrix, Table 4.4 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 value at *EVDmu12* and *EVDmu22* is 0, the maximum value at *recipmu1* and *recipmu2* come out to be ∞. *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 a squaring operation and $\sqrt{\delta^2 + 4} \neq \delta$. The fact that the minimum value

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	$1/x$	$[2.15\times10^{-23}, 1]$
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	$1/x$	$[0, 1]$
ev-y2	$1/x$	$[0, 1]$
negev-y2	$-x$	$[-1, 0]$
mult1	$a*b$	$[-1, 1]$
mult2	$a*b$	$[-1, 1]$
mult3	$a*b$	$[-1, 1]$
mult4	$a*b$	$[-1, 1]$
Vupdate	$\mathbf{AB}$	$[-3.51\times10^{6}, 1.76\times10^{6}]$
Dupdate1	$\mathbf{AB}$	$[-1.04\times10^{15}, 1.04\times10^{15}]$
Dupdate2	$\mathbf{AB}$	$[-1.04\times10^{15}, 1.04\times10^{15}]$

Table 4.4: Dynamic ranges for computations in 4×4 Jacobi EVD

at this node is 0, when it should not be so is the reason why further operations in the application become ∞ thereby leading to many incorrect computations and the 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 (4.3) and (4.4) which are restated here for convenience. As the number of iterations increase 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, $\delta \rightarrow \infty$. As $\delta \rightarrow \infty$, $\sqrt{\delta^2+4} \approx \delta$ and $\sqrt{\delta^2+4} - \delta \rightarrow 0$). 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 $(\sqrt{\delta^2+4} + \delta)^2$

$$\mu_1 = \frac{2}{\sqrt{\delta^2+4} - \delta} \qquad \mu_2 = \frac{2}{\sqrt{\delta^2+4} + \delta} \tag{4.5}$$

$$\delta = \frac{a-c}{|b|} \qquad \theta = \tan^{-1}\left[\frac{Im(b)}{Re(b)}\right] \tag{4.6}$$

Using our application exploration framework and adopting basic principles of precision analysis, we have identified the source of precision loss in the Jacobi eigenvalue decomposition. By identifying solutions to this problem, and verifying them, we can provide useful feedback to the low-level designers regarding the implementation.

We aim to reformulate the equation for μ_1 in (4.5) in such a way as to avoid the difference operation. Note that when $\delta > 0$, μ_2 can be computed without any precision loss due to the presence of the addition operation. Obviously, if reformulating the expression for $\mu_{1,2}$ is feasible, it would be a more foolproof solution

to confirm the convergence of the algorithm.

On close inspection of the equations in (4.5), it can be seen that μ_1 can be expressed in terms of μ_2 thereby avoiding the difference operation.

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

$$\mu_1 = \frac{1}{\mu_2} = \frac{\sqrt{\delta^2+4}+\delta}{2}$$

This theoretically eliminates the root of the precision problem, and is useful feedback to the algorithm developers. 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 4.5 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 even with $E = 8$. We confirm this with C simulation.

This analysis can also be verified with the C-based implementation by rewriting the code segment corresponding to $\mu_{1,2}$'s computation. The new implementation converged for all the precisions under consideration and produced valid results for all configurations with $E \geq 8$. The SNRs were expectedly higher for higher precisions, but in all cases were above the minimum required SNR of 50 dB. Thus the minimum required internal precision for computation for the Jacobi eigenvalue decompositon among the considered precisions is $I = 16, E = 8$.

By using DIF to model the eigenvalue decompositon and functional DIF to prototype the dynamic range analysis of this application in the DICE framework, we

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	$[4, 2.16\times10^{45}]$
sqrt	$\sqrt{x}$	$[2, 4.65\times10^{22}]$
EVDmu12	$(\sqrt{\delta^2+4}+\delta)^2$	$[10^{-45}, 2.16\times10^{45}]$
Sqrtp1mu1	$\sqrt{x+1}$	$[1, 4.65\times10^{22}]$
ev-x1	$1/x$	$[2.15\times10^{-23}, 1]$
ev-x2	$1/x$	$[2.15\times10^{-23}, 1]$
recipmu1	$1/x$	$[4.6\times10^{-46}, 10^{-45}]$
recipy1	$\sqrt{x+1}$	$[1, 4.65\times10^{22}]$
ev-y1	$1/x$	$[2.15\times10^{-23}, 1]$
negev-y2	$-x$	$[-1, -2.15\times10^{-23}]$
ev-y2	$1/x$	$[2.15\times10^{-23}, 1]$
mult1	$a*b$	$[-1, 1]$
mult2	$a*b$	$[-1, 1]$
mult3	$a*b$	$[-1, 1]$
mult4	$a*b$	$[-1, 1]$
Vupdate	**AB**	$[-3.51\times10^{6}, 1.76\times10^{6}]$
Dupdate1	**AB**	$[-1.04\times10^{15}, 1.04\times10^{15}]$
Dupdate2	**AB**	$[-1.04\times10^{15}, 1.04\times10^{15}]$

Table 4.5: Dynamic ranges 4×4 Jacobi EVD with reformulation

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 in model-based or dataflow based implementations.

Chapter 5
Conclusion

In this work, we proposed enhancements to existing design flows that utilize model-based design to extract dataflow behavior and to verify cross-platform correctness of individual actors. We introduced the DSPCAD Integrative Command Line Environment (DICE) as a realization of managing these enhancements. DICE enjoys a high level of synergy with Dataflow Interchange Format (DIF), our model-based development environment, in high level application exploration and in the seamless integration of testing with design.

We demonstrated this with an exploration study into the required precision of computations in the Jacobi Eigenvalue Decomposition (EVD). We modeled the application graph and the precision analysis with DIF and functional DIF and executed the entire analysis by slightly reconfiguring the DICE unit testing framework. We were able to analyze the required precisions at different nodes in the application graph and hence identify the nodes that required higher precision than available. By reformulating the mathematical expressions for these operations, we circumvented this problem and provided 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.

Bibliography

[1] 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. Technical Report UMIACS-TR-2009-13, Institute for Advanced Computer Studies, University of Maryland at College Park, August 2009.

[2] S. S. Bhattacharyya, S. Kedilaya, W. Plishker, N. Sane, C. Shen, and G. Zaki. Using the DSPCAD integrative command-line environment: User's guide for DICE version 1.0. Technical Report DSPCAD-TR-2009-01, Maryland DSPCAD Research Group, Department of Electrical and Computer Engineering, University of Maryland at College Park, 2009.

[3] C Hsu, I. Corretjer, M. Ko., W. Plishker, and S. S. Bhattacharyya. Dataflow interchange format: Language reference for DIF language version 1.0, user's guide for DIF package version 1.0. Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2007. Also Computer Science Technical Report CS-TR-4871.

[4] W. Plishker et al. Model-based DSP implementation on FPGAs. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 8–11, Fairfax, Virginia, June 2010.

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

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

[7] J. T. Buck and E. A. Lee. Scheduling dynamic dataflow graphs using the token flow model. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

[8] C. Hsu and S. S. Bhattacharyya. Porting DSP applications across design tools using the dataflow interchange format. In *Proceedings of the International Workshop on Rapid System Prototyping*, pages 40–46, Montreal, Canada, June 2005.

[9] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for rapid prototyping. In *Proceedings of the International Symposium on Rapid System Prototyping*, pages 17–23, Monterey, California, June 2008.

[10] D. Gerlach and A. Paulraj. Adaptive transmitting antenna arrays with feedback. *IEEE Signal Processing Letters*, 1(10):150–152, oct 1994.

[11] A. J. Grant. Performance analysis of transmit beamforming. *IEEE Transactions on Communications*, Vol. 53:738–744, 2005.

[12] I. E. Telatar. Capacity of multi-antenna gaussian channels. *European Transactions on Telecommunications*, 10:585–595, 1999.

[13] F. W. Vook, T. A. Thomas, and Z. Xiangyang. Transmit diversity and transmit adaptive arrays for broadband mobile OFDM systems. In *IEEE Wireless Communications and Networking*, volume 1, pages 44 –49, 2003.

[14] D. Menard, D. Chillet, F. Charot, and O. Sentieys. Automatic floating-point to fixed-point conversion for DSP code generation. In *Proceedings of International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES 2002)*, pages 270–276, Grenoble, France, October 2002.

[15] H. Keding, M. Willems, M. Coors, and H. Meyr. FRIDGE: A fixed-point design and simulation environment. *Design, Automation and Test in Europe Conference and Exhibition*, page 429, 1998.

[16] K. Kum, J. Kang, and W. Sung. AUTOSCALER for C: an optimizing floating-point to integer C program converter for fixed-point digital signal processors. *IEEE Transaction on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):840–848, September 2000.

[17] P. Belanovic and M. Rupp. Automated floating-point to fixed-point conversion with the fixify environment. In *RSP 2005: Proceedings of the 16th IEEE International Workshop on Rapid System Prototyping*, pages 172–178, Washington, DC, USA, 2005. IEEE Computer Society.

[18] A. A. Gaffar, W. Luk, P. Y. K. Cheung, and N. Shirazi. Customising floating-point designs. In *FCCM 2002: Proceedings of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, page 315, Washington, DC, USA, 2002. IEEE Computer Society.

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