ABSTRACT

Title of thesis: DESIGN AND TESTING METHODOLOGIES FOR SIGNAL PROCESSING SYSTEMS USING DICE

Soujanya A. Kedilaya, Master of Science, 2010

Thesis directed by: Professor Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering

Embedded computing has witnessed explosive growth in recent years in both scientific and consumer applications, driving the need for high-performance complex digital systems. The design and integration of embedded systems in heterogeneous programming environments is still largely done in an ad hoc fashion, and is especially sluggish in large collaborative projects with globally-distributed design teams, making the overall development process more complicated, error-prone and tedious. This has led to the increased need for systematic and efficient design flows.

In this work, we propose enhancements to existing design flows that utilize model-based design to extract dataflow behavior and to verify cross-platform correctness of individual actors. The DSPCAD Integrative Command Line Environment (DICE) is a realization of managing these enhancements to the design flow. DICE, with its platform independent conventions, facilitates the efficient management of design and test of cross-platform software projects, and enjoys a high level of synergy with the Dataflow Interchange Format (DIF), a model-based development

environment for signal processing systems.

We demonstrate this design flow with two case studies. We use DICE's novel test framework on modules of a triggering system in the Compact Muon Solenoid of the Large Hadron Collider (LHC), and demonstrate how the cross-platform model-based approach, automatic testbench creation and integration of testing in the design process alleviate the rigors of developing such a complex digital system.

The second case study is an exploration study into the required precision for eigenvalue decomposition (EVD) using the Jacobi algorithm. 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.

DESIGN AND TESTING METHODOLOGIES FOR SIGNAL PROCESSING SYSTEMS USING DICE

by

Soujanya A. Kedilaya

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2010

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Associate Professor Emeritus Steve Tretter
Associate Professor Gang Qu

Acknowledgments[1]

Foremost, I would like to thank my advisor, Professor Shuvra Bhattacharyya for his inspiration and guidance. This thesis and the two years of work that went into it would not have been a thoroughly enjoyable experience if not for his enthusiasm, sound advice and encouragement.

I am indebted 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. I look forward to spending more time at TI as I take on a full-time position there.

The highlight of the last year has been the five months I spent as a visiting student at the Salzburg University of Applied Sciences (FHS) in Salzburg, Austria where I was exposed to a whole new school of thought. I wish to thank Professor Bhattacharyya, Professor Gabriele Abermann of FHS, and the Austrian Marshall Plan Foundation 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,

[1]This research was supported in part by the Austrian Marshall Plan Foundation, the US National Science Foundation (Award Number 0823989), and Texas Instruments.

Peter and the other staff members of the University who made my stay in Austria unforgettable. Danke schön!

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. The DSPCAD group has been a cornerstone in my two years at UMD, and I would like to thank Chung-Ching, Nimish, George, Wu, Kishan, Ruirui, Hojin and Inkeun for providing great company and ideas.

I would like to express my sincere gratitude to Dr. Steve Tretter and Dr. Gang Qu for serving on my Masters Thesis committee.

I wish to thank all my friends at College Park, who helped me make the best out of Graduate School life and the residents of 4301-102, Rowalt Drive for all the emotional support, camaraderie and entertainment.

Lastly and most importantly, I wish to thank my parents and my sister who have always stood by me and encouraged all my endeavours in life. To them I dedicate this thesis.

Table of Contents

List of Tables vi

List of Figures vii

1 Introduction 1
1.1 Contributions of this thesis . . . 3
1.2 Outline of the thesis . . . 5

2 Background 7
2.1 Dataflow Modeling . . . 7
2.2 Dataflow Interchange Format . . . 8
2.2.1 The DIF Package (TDP) . . . 8
2.3 Functional DIF . . . 9

3 DSPCAD Integrative Command Line Environment (DICE) 12
3.1 Introduction . . . 12
3.2 DICE Unit Testing Framework . . . 13
3.3 Related Work . . . 15

4 Cross-Platform Model-Based Design Approach 17
4.1 High-level Application Specification . . . 19
4.2 Interface Specification for Tests . . . 20
4.3 Model-based Testing . . . 21
4.4 Automatic Testbench Creation . . . 23

5 Case Study - Trigger System of the Compact Muon Solenoid 27
5.1 CMS Calorimeter Trigger system . . . 27
5.2 Motivation for Cross-platform Model-based approach in CMS . . . 28
5.3 Demonstration . . . 29

6 Case Study - Precision Analysis of the Jacobi Eigenvalue Decomposition 33
6.1 EVD in MIMO wireless technology . . . 33
6.2 Eigenvalue Decomposition . . . 35
6.2.1 Existing EVD algorithms . . . 36
6.3 The Jacobi idea . . . 38
6.4 Motivation for Precision Analysis . . . 40
6.5 Functional Simulation and Performance Evaluation . . . 43
6.5.1 Simulation Framework . . . 44
6.5.2 Performance Evaluation . . . 44
6.5.3 Simulation Parameters . . . 46
6.5.4 Results and Discussion: Part I . . . 46
6.6 Dataflow model of the Jacobi EVD . . . 47
6.6.1 Introduction . . . 47

6.6.2 Related Work 48
6.6.3 Dataflow model for Jacobi EVD 49
6.6.3.1 2×2 Matrix 50
6.6.3.2 4×4 and 8×8 Matrices 50
6.7 Dynamic Range Analysis 52
6.7.1 Interval arithmetic for Jacobi EVD 53
6.7.2 Dynamic range simulation with functional DIF and DICE . . 54
6.8 Results and Discussion: Part II 58

7 Conclusion 65
7.1 Future Work 67

Bibliography 68

List of Tables

5.1 DICE testbenches automatically generated from dataflow models of CMS actors 30

6.1 Convergence of Jacobi EVD implementation for all precisions 47
6.2 Number of iterations of the base graph for Jacobi EVD 50
6.3 Interval Arithmetic 54
6.4 *runme* file for multiple iterations in Jacobi EVD 57
6.5 Dynamic ranges for various data formats 58
6.6 Dynamic ranges for computations in 4×4 Jacobi EVD 59
6.7 Dynamic ranges 4×4 Jacobi EVD with reformulation 62

List of Figures

2.1 DIF-based design flow. 10

4.1 Traditional design flow with proposed model-based features. 18
4.2 DICE file and directory structure of a cross-platform project. 22
4.3 Automatically generated testbench - DIF graph 25
4.4 Automatically generated testbench - DIF file 26

5.1 Automatically generated testbench for the ClusterWeight actor. . . . 31

6.1 MIMO Channel 34
6.2 Dataflow graph for the 2x2 Jacobi EVD 51
6.3 SNR in dB vs condition number for 2×2 Hermitian matrix 63
6.4 SNR in dB vs condition number for 4×4 Hermitian matrix 64
6.5 SNR in dB vs condition number for 8×8 Hermitian matrix 64

Chapter 1

Introduction

As the demands for functionality and performance are increasing in embedded software development, more diverse sets of target platforms are being used to satisfy these demands including digital signal processors (DSPs), microcontrollers, and field programmable gate arrays (FPGAs). This can be a challenging task as programmability for high performance platforms is inherently low level and is not tailored for a particular application. This is the fundamental reason why programming approaches for high performance implementations are proliferating where the different approaches have their own programming models and development environments and are often developed with their own design teams. Also, due to the increased use of and need for the design and development of complex digital systems in scientific fields like wireless communications, medical imaging, high energy physics, there arises a need for application development techniques more accessible to scientists and engineers specialized in the application domain, but not in high performance hardware design. Thus the original application description also often has its own programming environment to facilitate fast development of the platform-independent algorithm, which can range from general imperative languages like C, to object oriented ones like C++ or Java, to domain specific approaches like MATLAB. This naturally leads to the need for having multiple implementations for algorithm

development and hardware designs.

This multitude of languages and programming environments make the design flow for modern embedded systems time consuming and error prone as developers are often manually transcoding between them. With no end in sight for this problem in the near future, developers would need tools that are nimble enough to deal with this uncertainty of hopping from one platform to the other with nearly unportable code.

Many best practices are utilized in industrial and academic environments to help this process, such as automatically generating documentation, auto-configuration, adherence to interface specifications, and unit testing. In particular, unit testing facilitates productive design by integrating testing early into the design flow to catch erroneous or unexpected behavior in a module earlier in the design cycle. Such techniques have proven effective for many languages and platforms individually, but for systems that employ more than one of these into a single design process, these tools still leave many manual, error prone steps possible, leading to longer design times with lower quality implementations. In today's signal processing and control system based applications, model-based design is especially useful to help isolate domain experts from the need to understand low-level hardware and software details. Model-based design improves efficiency by using a common design abstraction across project teams and by linking designs directly to requirements.

1.1 Contributions of this thesis

In this work, we propose to enhance existing design flows with model-based design that extracts dataflow behavior and verifies cross-platform correctness of individual actors. Furthermore, with model-based development, automatic test-bench creation is possible, improving the ease with which designers can create cross-platform tests.

The DSPCAD Integrative Command Line Environment (DICE) [1] 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. Not only do designers want build and test structures to port as much as possible, but equally important, and something that is ignored by other tools, is the ability to quickly hook in the inevitable platform specific tools shipped with the new platform (such as the compiler, synthesizer, and documentation engine). DICE fits in this requirement by being a grounded, yet light-weight tool that will be able to change with a shifting low level programming landscape. 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. DICE runs and analyzes tests using shell scripts and programs written in high-level languages and is an open access resource available for download [2] [3].

We use the Dataflow Interchange Format (DIF) [4] 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), we explore 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.

As part of this work, we present two case studies. First we demonstrate this novel test framework on modules of a triggering system for the Large Hadron Collider (LHC) [5], which includes algorithm development modules in C++ and implementation modules developed in Verilog. The LHC is a true example of a complex digital system, and is a collaborative project among geographically distributed teams each using diverse target platforms and different programming paradigms. Through the integration of model-based design with DICE, we highlight the use of good practices in system testing and integration to decrease overall development time, and in the process validate equivalence of modules and expose concurrency in the application.

The second part of this work deals with an exploration study into the internal precision of computation for the Jacobi Eigenvalue Decomposition (EVD) [6]. EVD is a matrix decomposition where a given square matrix is factorized into a canonical form containing the eigenvalues and eigenvectors of the matrix. EVD is used in a wide range of modern signal processing and communication applications such as Multiple-Input and Multiple-Output (MIMO) wireless communication, image recognition technologies, and direction of wave arrival estimation algorithms. In

the context of this work, the EVD algorithm is being studied as part of a beamforming application inherent to MIMO wireless technology. The primary objective of this work is to determine the least precision required for DSP implementation of EVD in order to obtain the minimum desired performance. But also 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.

With these two case studies, we aim to show that using this model-based approach and the integration of DICE, we are able to make model-based design easier on the application designer while still being rigorous, agile, and evolvable.

1.2 Outline of the thesis

The outline of the thesis is as follows. Chapter 2 provides a background on the dataflow model of computation for DSP applications. Chapter 3 introduces DICE and the DICE Unit Testing Framework. In Chapter 4, we describe the cross-platform model-based design approach with DIF and DICE, and an illustration of model-based testing, before providing a demonstration of the same in Chapter 5

with the triggering system of the Large Hadron Collider as an example application. Chapter 6 presents a case study on the precision analysis of the Jacobi Eigenvalue Decomposition, this time demonstrating the use of DIF and DICE in application exploration. Conclusions and scope for future work are discussed in Chapter 7.

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) [7], single-rate dataflow, homogeneous synchronous dataflow (HSDF), or a more complicated model such as boolean dataflow (BDF) [8].

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) [4], 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 [9]. 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

full explanation of it, see [4]). 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. To promote reuse, DIF provides common dataflow features so that developers and users of design tools can focus on the novel features and unique constraints associated with their design problems. Beyond these features, DIF is also suitable as a design environment for implementing dataflow-based application representations. Developer productivity benefits from the tailored semantics and the dataflow tool suite. The internal representation can be turned into functional implementation with the DIF-to-C tool [10], which is an efficient and optimized code synthesis tool for SDF.

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

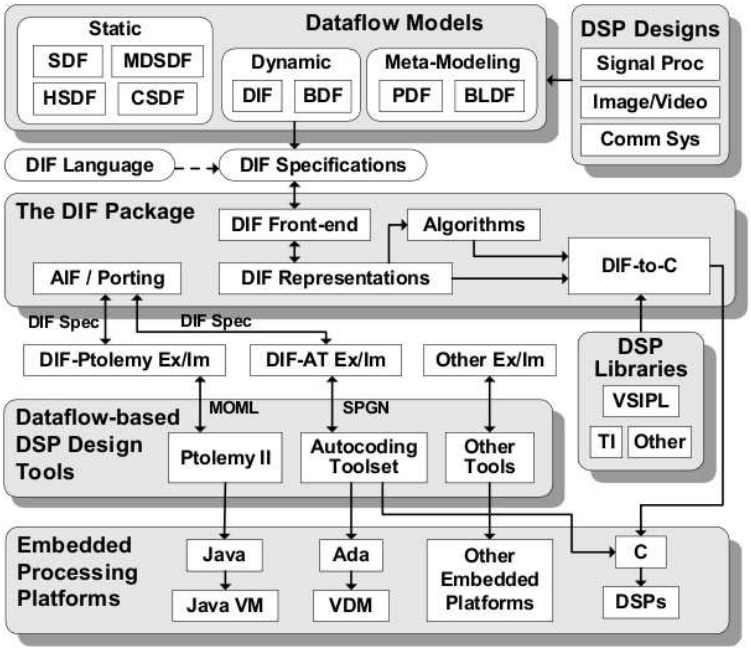

Figure 2.1: DIF-based design flow.[4]

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 [11] (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.

Once the functional DIF prototype has been completed and validated, the designer can proceed with greater confidence to tackling the lower level implemen-

tation details required for the targeted HDL implementation. At the same time, the designer has a valuable reference implementation for functional validation of the HDL design as it evolves.

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) [11], 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 6.

The use of the unit testing framework for cross-platform model based design is elucidated in Sec. 5.3.

3.3 Related Work

Typically the tools designers employ for design and verification are language specific [12], [13]. More than just a syntactic customization, such frameworks are often tied to fundamental constructs of the language. For example, in CppUnit, a unit test inherits from a base class defined by CppUnit. A test writer then overloads various methods of the base class to put the specific unit test in this framework. Tests requiring the specific features that leverage the constructs of a language (e.g. in an object oriented language, checking that method exhibits the proper form of polymorphism) are well served by these approaches. Furthermore, these language-

specific approaches work well when designers are using only a single language or a single platform for their final implementation. But when designers must move between languages with different constructs (like C++ to Verilog), the existing tests must be rewritten. This creates extra design effort and creates a new verification challenge to ensure unit tests between these two languages are in fact performing the same test.

Probably the most related framework to DICE is the Test Anything Protocol (TAP) [14]. Like DICE, TAP achieves language independence by defining the protocol that manages the communication between unit tests and a test harness. Individual tests (TAP producers) communicate test results to the testing harness (TAP consumers). TAP enables multi-platform and multi-language design, but only at the communication boundary. Unit tests need only adhere to the communication design, leaving test writers with no specific language independent mechanism for writing the tests themselves. Indeed, many language specific unit tests have TAP compatible outputs so they may be hooked into a larger multilanguage testing environment. In contrast with these other efforts, we provide a cross-platform approach to utilizing model-based utilities and unit test writing by inferring dataflow models and leveraging primitive datatypes with DICE. Some unit test frameworks have data generators, but DICE encourages designers to think of module interface in terms of streaming data primitives. DICE captures these input/output sequences in files and then ensures the output files match with a structured build and run framework. These assumptions allow test writers in DICE to build more complete solutions than a test communication protocol alone.

Chapter 4

Cross-Platform Model-Based Design Approach

Figure 4.1 illustrates the traditional design flow of first performing application exploration in a high-level development environment to achieve correct functionality and do preliminary planning for implementation. Once the design is finalized, the application is either synthesized or transcoded to the programming environment of the target platform. But in cross-platform design flows, developers need a description that will have meaning for future platforms. Therefore we require a formal specification with structure and formalisms as defined by the application area and not by any platform, so that even as the target platform shifts, there is still a good starting point to take on whatever platform specific issues arise with the next target. This is a more suitable paradigm than C, because it provides a more structured description with customizable programming restrictions (i.e. model selection). In order to incorporate model-based design in a cross-platform design flow, we proposed an approach [15] that augments this design flow by detecting the models of the actors used in the application. By using this information it is possible to provide more analysis to the application exploration phase and improve testing. In this thesis, we present DICE a cross platform based design tool which is able to verify the functionality of the final implementation and the original high level description of it.

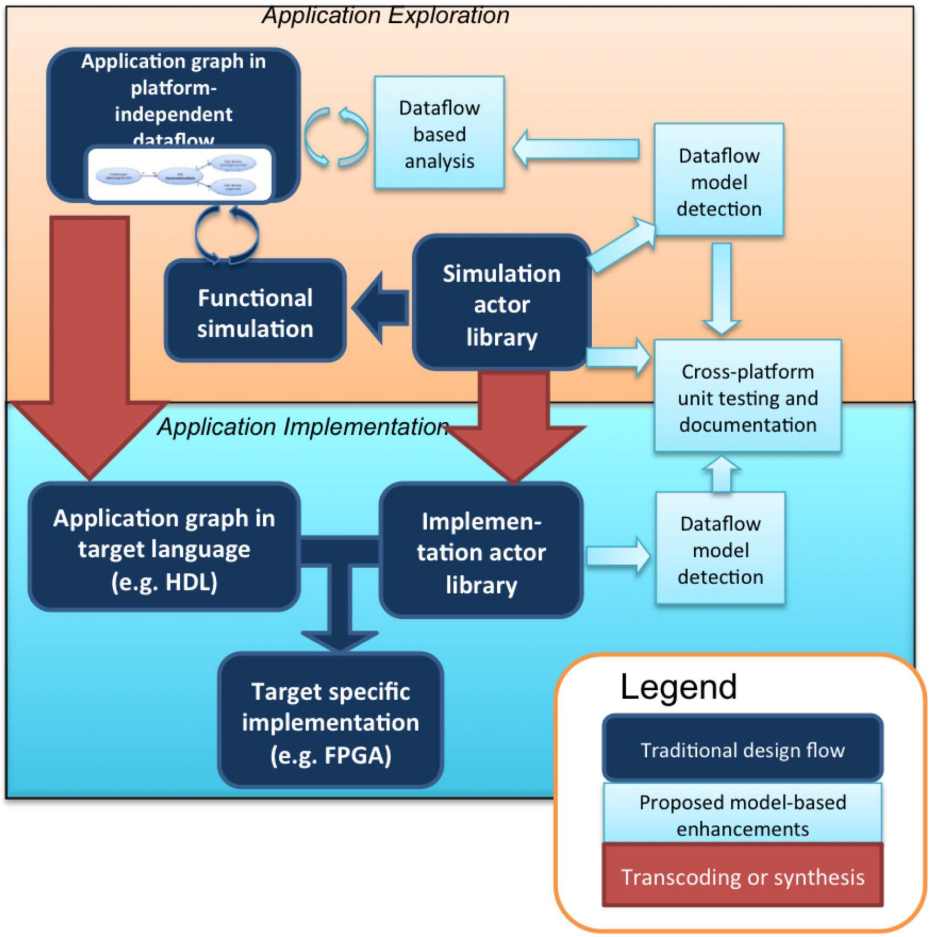

Figure 4.1: Traditional design flow augmented with proposed model-based features.[15]

4.1 High-level Application Specification

System development 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 initial higher level application specification can also be effectively used for testing purposes. Such a description allows testing the functionality of the application specification for a valid set of inputs. This functionally correct implementation could then be used as a benchmark as the development of underlying subsystem(s) in the application proceeds on various platforms using different tools.

DIF provides one such tool for model-based design and implementation of signal processing systems using dataflow graphs. A designer starts with a platform-independent description of the application in DIF. This structured, formal application description is an ideal starting point for capturing concurrency and optimizing and analyzing the application. After settling on the DIF description, a designer can refine this description to a high-performance implementation by employing platform specific tools including compilers, debuggers, and simulators. Any transcoding or platform specific enhancements are accommodated by DICE via its flexible but standardized build and test framework. This allows designers to utilize the same de-

sign framework at inception as they do at final implementation. Software developed jointly with DIF and DICE enjoys a single, cross platform software management framework, where verification of modules is handled consistently throughout each phase of development. If DIF is used as the reference description, transcoding effort is saved by having a formal, unambiguous application description to base the implementation on. Quality is controlled with a high degree of automation through the direct reuse of unit tests in DICE.

The DICE framework can be applied for testing each of the individual modules, subsystems, or even an entire application. In case of testing an individual module, we specify valid inputs and expected correct outputs for that module using concepts mentioned in Sec. 4.2. We create wrapper modules consisting of an individual module, its valid input interface, and the output interface. Such a wrapper module can then be tested independently of other modules. This functionally correct module description can then be used to develop platform or language specific implementations of that module. We could use the same test suite for testing and verifying the correctness of such modules using the features of DICE, as explained in Section 4.3.

4.2 Interface Specification for Tests

Most of the tools available for unit testing require the test inputs and outputs to be specified in a way that is platform or application language dependent. Such dependence makes it difficult to use tests designed for a particular platform or ap-

plication language across other platforms or languages. Our framework provides a solution to this problem by allowing test inputs and outputs to be specified in a manner that is platform and application language independent. While using DICE framework, test inputs and outputs are of primitive data types. The valid sets of such inputs and the corresponding correct outputs are solely dependent on the functionality of the module being tested. A given set of valid inputs for a module should produce a corresponding expected correct output depending upon the functionality of that module and is independent of the application language or implementation platform. DICE framework has platform independent test features and utilities that help running tests with test suites so that tests may be uniformly created and aggregated. It has scripts to build the source code using plugins that call the language-specific compiler. This built code is tested for a valid set of inputs to generate the corresponding output. The resultant output is then compared with the expected correct output as determined by the functionality of the module being tested.

4.3 Model-based Testing

In order to accommodate cross platform operation, the DICE engine consists of a collection of utilities implemented as Bash scripts, C programs, and python scripts. By writing DICE based on these open-source command line interfaces and languages, DICE is able to operate on different platforms such as Windows (equipped with Cygwin), OS-X, Solaris, and Linux. This gives DICE a wide base from which to integrate specific design flows. From this base, we have architected a testing

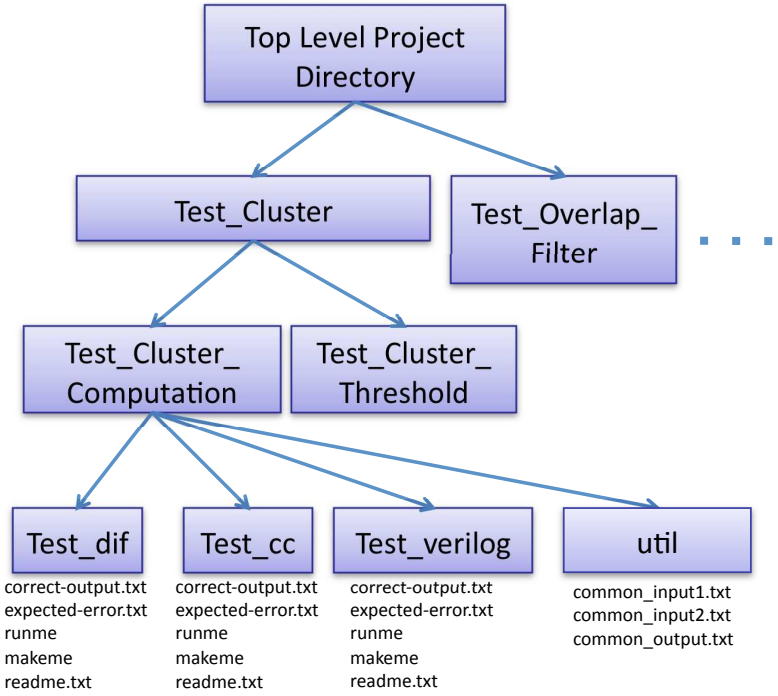

Figure 4.2: DICE file and directory structure of a cross-platform project using our testing approach.[15]

approach that is applicable across design flows.

To implement a unit testing suite, the developer provides a test reference for the functional behavior of the module under test (MUT). This reference consists of a set of inputs and the set of outputs that are produced as a result of correct processing of those inputs. Unit testing is performed by executing the module and comparing the actual output with the correct expected behavior. In the event of a module throwing an expected error (e.g. a designer is trying to see that when an input condition is violated, the application returns the proper error), this can be added to the test reference as well.

In the example in Figure 4.2, there are three implementations under test for the same module: DIF, C++, and Verilog. The input and output patterns are common

to each of these tests and reside in the util directory. While each language has customized build and simulation scripts in *makeme* and *runme*, and *correct-output.txt* and *expected-error.txt* tailored to their simulation environments, the fundamental inputs and outputs are directly shared between these platforms. By using such a framework that automates the process of test verification, any change to the basic MUT can be verified not only for the new functional correctness, but also to ensure that it does not ruin a previous correct behavior. This also enables an incremental code development, and reduces the development and verification time.

4.4 Automatic Testbench Creation

With model-based development, automatic testbench creation is possible, improving the ease with which designers can create cross-platform tests. We use the dataflow interchange format (DIF) as our model-based development environment to create automatic testbench using a testbench creator and improve the ease with which designers can create cross-platform tests.

For the high-level application specification, the formal description of the functionality of each module is done using functional DIF by implementing the system modules or graph nodes as *actors*. Each actor contains in its description, the names of its input and output ports and other related properties. A given module-under-test is provided as an input to the testbench creator. The testbench creator extracts information about the input and output ports of the given MUT from the description file of the corresponding actor, and attaches *File Readers* and *File Writers* to

each input and output port respectively. *File Readers* and *File Writers* are also functional DIF based actors that read and write input and output samples from input and output text files respectively. The testbench generated is a dataflow graph specified in the form of a DIF file (with .dif file extension), which can be simulated with appropriate input files containing the test patterns for the MUT.

Figures 4.3 and 4.4 demonstrate the automatic testbench generation for the MUT *ComplexMag*. *ComplexMag* is an actor that computes the magnitude of a complex number. Hence this actor has two input ports corresponding to the real and imaginary parts of a complex number and one output port which is the computed complex magnitude. The testbench creator extracts the port-related information from the actor file and automatically assigns two *File Readers*, *input1* and *input2* to read the inputs from the files *input1.txt* and *input2.txt*. The file names and the names of the *File Reader* actors are directly adapted from the port names in the actor file. Similarly a *File Writer* is also assigned for the sole output port *output*. The testbench creator also automatically creates the edges connecting the *File Readers* and the *File Writer* to the MUT *ComplexMag*. The input edges are named *in1*, *in2* etc. and the output edges are named *out1*, *out2* and so on. In this manner, testbench creation is automated for any MUT and the MUT can be tested straightway for functionality.

When the total number of IO ports for the MUT is very high, the automatic generation of the testbench saves a developer the time of writing the testbench. This aspect has been demonstrated in Sec. 5.3 for modules of the trigger system of the Compact Muon Solenoid, some of which have dozens of IO ports.

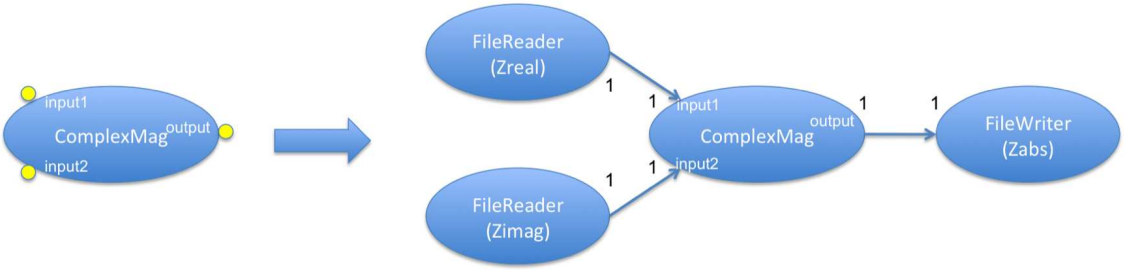

Figure 4.3: Automatically generated testbench for the *ComplexMag* actor that computes the magnitude of a complex number.

```
cfdf C {
    topology {
        nodes = input1, input2, ComplexMag, output;
        edges =
        in1(input1, ComplexMag),
        in2(input2, ComplexMag),
        out1(ComplexMag, output);
    }
    actor input1 {
    computation = "mapss.dif.actors.File_Reader";
    FileName = "input1.txt";
    output = in1;
    }
    actor input2 {
    computation = "mapss.dif.actors.File_Reader";
    FileName = "input2.txt";
    output = in2;
    }
    actor ComplexMag {
    computation = "mapss.dif.evd.bin.ComplexMag";
    input1 = in1;
    input2 = in2;
    output = out1;
    }
    actor output {
    computation = "mapss.dif.actors.File_Writer";
    FileName = "output.txt";
    input = out1;
    }
}
```

Figure 4.4: Automatically generated testbench in the form of a DIF file for the *ComplexMag* actor that computes the magnitude of a complex number.

Chapter 5

Case Study - Trigger System of the Compact Muon Solenoid

DICE is actively being used as a test framework for the Trigger system of the Compact Muon Solenoid (CMS) Detector of the Large Hadron Collider (LHC) at CERN. The LHC [5] is the world's largest particle accelerator built for physicists to study the most fundamental questions in particle physics. Along with this collider, there are multiple particle detectors to track the motion and measure the energy and charge of the new particles thrown out in all directions from the collisions. The Compact Muon Solenoid (CMS) [16] being one of these is designed as a general-purpose detector, capable of studying many aspects of proton collisions at energies in the TeV range. It contains subsystems which are designed to measure the energy and momentum of photons, electrons, muons, and other products of the collisions.

5.1 CMS Calorimeter Trigger system

Although each beam consists of thousands of billions of particles, the particles are so tiny that the chance of any two colliding is very small. To maximize the probability of observing interesting events in the detector, a very large number of collisions are required. However, the amount of raw data from each collision to store and process is massive, and well beyond the maximum rate that can be archived by the online computer farm. Thus, a trigger system in the CMS is used to perform

the initial filtering by passing on only interactions of interest. The Level-1 (L1) trigger system [17] first reduces the proton-proton interaction rate and then a High Level Trigger (HLT), using an on-line computer farm, handles the remaining rate reduction. The L1 trigger is designed to do a fast detection of signatures of isolated and non-isolated electrons, photons, jets, muons, and missing and total transverse energy by comparing the measured energies to certain thresholds. The identified events of interest are sent to the HLT where a more detailed analysis takes place before archiving the obtained data.

5.2 Motivation for Cross-platform Model-based approach in CMS

Much of the logic in the trigger system is contained in custom and semi-custom Application Specific Integrated Circuits (ASICs) and Field Programmable Gate Arrays (FPGAs). In general, systems for high energy physics (HEP) applications like the CMS trigger increasingly depend on FPGAs for data processing and communication. Increased use of and need for complex FPGA-based designs in scientific fields like HEP necessitates application development techniques more accessible to scientists and engineers specialized in the application area, but not in high performance hardware design. This naturally leads to the need for having multiple implementations for algorithm development and hardware designs for an application such as the CMS trigger.

The design of the CMS trigger, being a complex digital system, is collaborative between multiple geographically distributed research groups, each of which

create their designs independently, often using different styles and techniques. On the whole, over 130 institutions all over the world collaborate on the design and working of the LHC. This complicates system testing and integration, and hinders the use of good practices important to decrease the development time. For example, with the current CMS trigger system, dozens of teams from different institutions have contributed to the design of hundreds of boards. Individual teams use different design tools, hardware description languages, and FPGA platforms or ASICs in their designs. This non-uniform method of design has made the digital systems in the current CMS trigger difficult to maintain, test, and enhance. DICE as a platform independent framework supports such projects that involve such heterogeneous programming languages and offers a unified framework for testing and integration.

By having access to a common test suite through DICE, designers can verify any changes they make to the design incrementally and independently. Furthermore because of the emphasis placed on adding tests in the test suite as an integral part of the development process, designers can have high confidence that their changes to the code are being tested comprehensively with respect to the rest of the code base.

5.3 Demonstration

Building on our integration of testing considerations into this project, we have explored novel synergies between dataflow-based system design and unit testing methodologies. We use the evolving test suites in the project to help ensure consis-

Actor	Input	Output	# samples
Cluster Threshold	12	12	1333
Cluster Computation	12	6	1333
Cluster Overlap Filter	8	4	10248
Cluster Weight	4	2	10248
Jet Reconstruction	1	2	128

Table 5.1: DICE testbenches automatically generated from dataflow models of CMS actors

tency between multiple implementations of the same actor in a design. For example, a given functional block in the CMS trigger has a Java-based version for the DIF simulation, a C++ based version for software emulation, and a Verilog version for FPGA implementation. We have created a library of actors corresponding to each module in the L1 trigger. The common DIF representation to which these actors interface ensures that the actors are designed with a standard, precise, and efficient method of communication to the enclosing application by standardizing its input and output interfaces. Unit testing provides a complementary form of integration support by validating equivalent input/output functionality across multiple versions of the same actor in an extensive and highly automated way. Based on software emulation, the algorithm developers provide a set of test patterns with inputs and corresponding outputs for correct functionality. All three actor versions are tested with the same tests thereby ensuring that tests need not be rewritten for different languages and that the multiple implementations are consistent with each other.

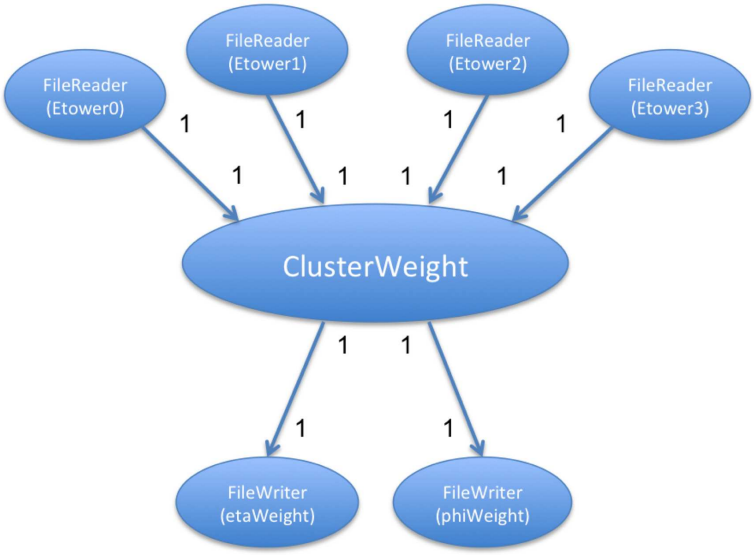

Figure 5.1: Test bench automatically generated from the model description of the Cluster Weight actor, which reads a cluster of four towers and calculates the η and ϕ weights based on the concentration of energies in the η and ϕ axes.

Through the integration offered by DICE, the application development has been systematic and development time has been reduced by identifying and fixing implementation and programming bugs early in the development cycle. Inconsistencies in versioning, data representation, and deviations from design specifications and requirements are some of the errors that were identified by utilizing the unified testing framework for the CMS trigger.

We have applied model based design with DICE testing to CMS actors which are summarized in Table 5.1. From the formal description of the functionality of a model, a DIF graph of the resulting testbench is generated (figure 5.1) that hooks into input text file readers and writes to the appropriate output text file writers. When the total number of ports for these actors is as high as 24 ports, the automatic

generation of the testbench saves a developer the time of writing the testbench. Due to this automatic generation of testbench and streaming input data, the actors can be seamlessly hooked into the framework making the testing process largely automated and hassle-free, thus making the integration of testing into the design process easy and very efficient. With more efficient dataflow scheduling techniques, the simulation time is expected to improve further.

Chapter 6

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.

6.1 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 along with orthogonal frequency division multiplexing (OFDM) will serve as the physical layer of two key technologies for mobile communication systems: LTE and WiMax. LTE is the 4G evolution of cellular systems, while WiMax seeks to deliver last mile wireless broadband access. Both LTE and WiMax technologies make extensive use of MIMO. MIMO schemes enable a variety of functions including multi-stream transmission for

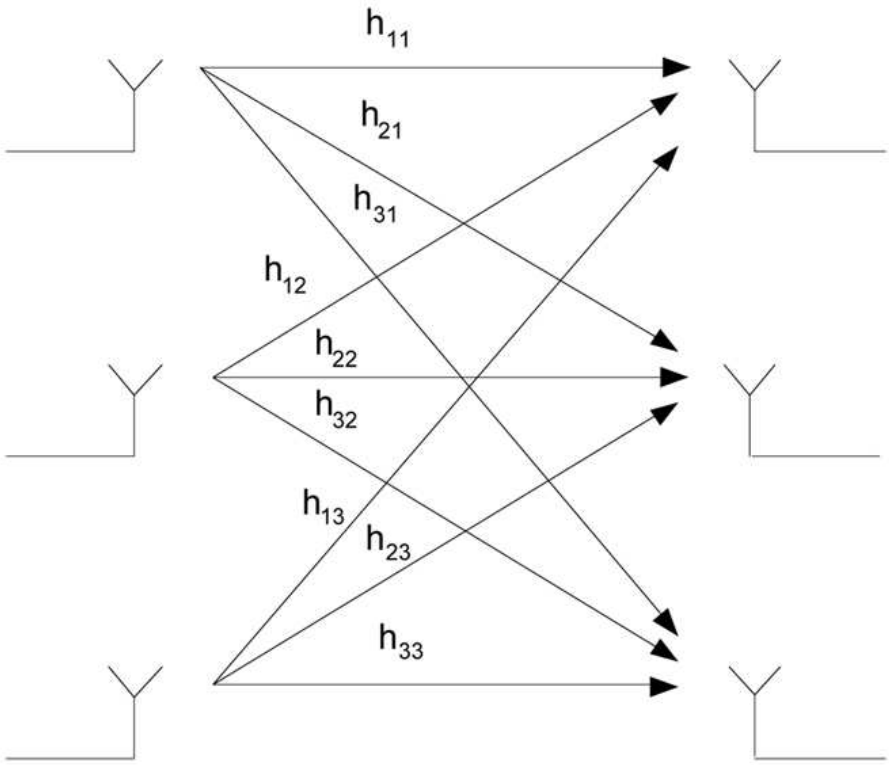

Figure 6.1: MIMO Channel. Figure taken from Intel Technology Journal [19].

high spectrum efficiency, improved link quality through diversity mechanisms, and adaptation of radiation patterns for signal gain and interference mitigation through adaptive beamforming [18].

Figure 6.1 shows a model for a MIMO channel. 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 [20]. More generally, vector information could be sent along all of the eigenchannels of $\mathbf{H}^{\dagger}\mathbf{H}$ as described in [21], resulting in increased spectral efficiency. [22] 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 $\mathbf{H}^{\dagger}\mathbf{H}$.

6.2 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

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

where λ is a scalar, termed the eigenvalue corresponding to $\mathbf{x}$ and the eigenvalues are roots of the equation

$$det(\mathbf{A} - \lambda\mathbf{I}) = 0$$

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

$$\mathbf{A} = \mathbf{VDV^{-1}} \tag{6.1}$$

$\mathbf{V}$ 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 $\mathbf{A}$.

All eigenvalue algorithms are iterative by Abel's theorem [23]. Abel's theorem shows that there are no direct methods for solving the general eigenvalue problem, for the existence of a finite, pre-specified procedure would imply the existence of a complicated formula for the solutions of an arbitrary polynomial equation. Due to the iterative nature of the EVD algorithms, a key aspect of any EVD algorithm is whether the algorithm always converges and if so, the rate of convergence.

6.2.1 Existing EVD algorithms

Many algorithms exist for the calculation of eigenvectors and eigenvalues of matrices. The choice of the algorithm will depend on the application, nature of the input matrix, required number of eigenvalues/eigenvectors and also computational constraints like speed, accuracy etc.

- Power Method: The power iteration is a very simple algorithm. It does not compute a matrix decomposition, and hence it can be used when **A** is a very large sparse matrix. However, it will find only one eigenvalue (the one with the greatest absolute value) and it may converge only slowly.

- QR Iterations: For dense, non-symmetric eigenvalue problems, QR method is shown to be the best algorithm [23]. However in the symmetric case, QR by itself is not the most efficient algorithm.

- Block Lanczos Method: Typically, the Block Lanczos method works well for large, sparse matrices, but is notorious for its instability with respect to rounding errors.

- Tridiagonal Methods: This involves reducing the given matrix **A** to tridiagonal form before applying other methods to do the final computation of the eigenvalues and vectors.

- Jacobi Method: This is one of the oldest known methods for EVD. It has good convergence properties, especially for small, dense matrices. It rose to prominence of late due to its inherent parallelism.

In the context of this work, the EVD algorithm is being implemented as part of a beamforming application inherent to MIMO wireless technology. The matrix of interest in this case is a Hermitian matrix that characterizes the channel between each pair of transmit and receive antennas. This channel matrix typically has the nature of being small and dense. 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.

6.3 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

$$J(p,q,\theta) = \begin{array}{c} \\ \\ p \\ \\ q \\ \\ \end{array} \begin{array}{c} \begin{array}{ccccccc} & & & p & & q & \end{array} \\ \begin{pmatrix} 1 & \dots & 0 & \dots & 0 & \dots & 0 \\ \vdots & \ddots & \vdots & & \vdots & & \vdots \\ 0 & \dots & c & \dots & s & \dots & 0 \\ \vdots & & \vdots & \ddots & \vdots & & \vdots \\ 0 & \dots & -s & \dots & c & \dots & 0 \\ \vdots & & \vdots & & \vdots & \ddots & \vdots \\ 0 & \dots & 0 & \dots & 0 & \dots & 1 \end{pmatrix} \end{array}$$

where $c = \cos(\theta)$ and $s = \sin(\theta)$ for some θ which is the rotation angle.

Algorithm 1 Pseudocode for the Jacobi EVD [6]

```
while offset(A) > ε do
  for p = 1 to n − 1 do
    for q = p + 1 to n do
      (c, s) = sym.schur2(A, p, q)
      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
```

In the pseudocode for Jacobi EVD, *sym.schur2(A,p,q)* represents a 2×2 symmetric Schur decomposition that computes the (c, s) pair that forces $A(p, q)$ to 0. 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.

However, the Jacobi method shown in the pseudo code does not apply to complex-valued Hermitian symmetric positive definite matrix. Instead it is possible to derive a closed-form expression of the EVD of a 2×2 complex-valued Hermitian symmetric matrix. This is then used to substitute the 2×2 Schur decomposition for each Givens rotation.

For such matrix $\begin{pmatrix} a & b \\ b^* & c \end{pmatrix}$ where a and c are positive real-valued, the eigenvalues can be written as:

$$\lambda_{1,2} = \frac{(a+c) \pm \sqrt{(a-c)^2 + 4|b|^2}}{2} \tag{6.2}$$

It can also be derived that the two eigenvectors can be written as:

$$v_{1,2} = \frac{1}{\sqrt{1 + |\mu_{1,2}|^2}} \begin{bmatrix} \mu_{1,2} \\ 1 \end{bmatrix} \tag{6.3}$$

$$\mu_{1,2} = \frac{b}{\lambda_{1,2} - a} = \frac{2b}{-(a-c) \pm \sqrt{(a-c)^2 + 4|b|^2}} \tag{6.4}$$

The above formulation albeit correct, suffers from numerical instability issues as the parameter b appears both in the numerator and denominator in equation (6.4), and with b being an off-diagonal element, the goal is to make it as close as possible to zero. With some algebraic manipulation, the eigenvector formulation in equations (6.3) and (6.4) can be rewritten as shown in equation (6.5). This formulation for 2×2 EVD can be used to replace the 2×2 real-valued Schur decomposition to extend the Jacobi method for any complex-valued Hermitian symmetric matrix.

$$v_1 = \begin{bmatrix} \frac{e^{j\theta}}{\sqrt{1+\frac{1}{|\mu_1|^2}}} \\ \frac{1}{\sqrt{1+|\mu_1|^2}} \end{bmatrix} \qquad v_2 = \begin{bmatrix} \frac{-e^{j\theta}}{\sqrt{1+\frac{1}{|\mu_2|^2}}} \\ \frac{1}{\sqrt{1+|\mu_2|^2}} \end{bmatrix} \tag{6.5}$$

$$\mu_1 = \frac{2}{\sqrt{\delta^2+4} - \delta} \qquad \mu_2 = \frac{2}{\sqrt{\delta^2+4} + \delta} \tag{6.6}$$

$$\delta = \frac{a-c}{|b|} \qquad \theta = \tan^{-1}\left[\frac{Im(b)}{Re(b)}\right] \tag{6.7}$$

6.4 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.

However, it may be optimistic to assume that the Jacobi EVD for Hermitian matrices can be fully implemented with wordlengths lesser than that of full 32-bit fixed-point or floating-point formats due to the nature of mathematical operations used in the algorithm.

For example, the Jacobi algorithm for eigenvalue decomposition is an iterative algorithm with each sweep of the algorithm propagating the errors of previous stages. So a high degree of precision in both the signal and coefficients are required to minimize the effects of these propagated errors. Second, as the number of sweeps increase, a number of off-diagonal elements start tending to zero and data accuracy must be maintained, even as it approaches zero due to the presence of division operations. In such an application it becomes important to fully analyze the data set 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.

In general, floating point DSPs achieve much greater precision and dynamic range at the expense of speed, since it requires multiple cycles for each operation. In the past, fixed-point DSPs were favored for high-volume applications where unit manufacturing costs had to be kept low and floating-point DSPs were adopted for low-volume applications where the time and cost of software development were of greater concern. This is because floating-point DSPs were easily programmable

through higher level languages like C and by coding real arithmetic directly onto the hardware, unlike fixed-point DSPs which had to be coded at assembly level making development time longer. However, this ease of use of floating point DSPs was offset by other factors like more internal circuitry, wider data buses, larger die area and packaging that resulted in a significant cost premium.

As semiconductor technology has evolved with transistor sizes having drastically reduced, cost issues relating to size of the DSP core is no longer as significant. Programming fixed-point DSPs has also evolved through the development of advanced tools for compiling, developing and debugging embedded applications. Overall, fixed-point DSPs still have an edge in cost and floating-point DSPs in ease of use, but the edge has narrowed to the point where the choice of using a fixed- or floating-point DSP boils down to whether floating-point math is needed by the application data set.

Clearly the floating-point format provides greater accuracy with larger mantissa word widths and the exponentiation vastly increasing the dynamic range available for the application. A wide dynamic range is important in dealing with extremely large data sets and with data sets where the range cannot be easily predicted.

With a new family of TI DSPs like 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 in floating-point. Keeping this in mind, the precision analysis was executed in a step-by-step manner. First a functional simulation of the Jacobi EVD was implemented in C

language in double precision, single precision and pseudo floating point formats for all matrix sizes of interest and the performance was analyzed, in terms of convergence and accurate computation of the eigensystems. Based on the results from this initial simulation (Sec. 6.5.4), 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. 6.6). This facilitated an analysis methodology where the data at the output edge of every node could be recorded, and its range over the entire duration of the program could be studied, leading to a better understanding of the required precision for computation and data representation (Sec. 6.8). We use DIF to model the dataflow of the Jacobi EVD and DICE as the framework within which the precision analysis is carried out.

6.5 Functional Simulation and Performance Evaluation

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. 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, precisions

of interest are all combinations from within $I = 16, 24, 31$ and $E = 6, 8, 10$.

6.5.1 Simulation Framework

The channel matrix $\mathbf{H}$ is generated with a Gaussian random number generator, such that $\mathbf{H}$ is Hermitian with real and imaginary parts of each element being 16-bit fixed-point numbers. The product of $\mathbf{H}$ and its conjugate transpose $\mathbf{H}^{\dagger}$ provides the input matrix $\mathbf{A}$ where each complex number entry has 32-bits each for real and imaginary parts. This matrix can be converted to single or double floating point, or custom pseudo floating point precision using the respective conversion routines.

The Jacobi EVD routines written in C compute the eigenvalue decomposition of $\mathbf{A}$. The resulting $\mathbf{D}$ and $\mathbf{V}$ matrices contain the eigenvalues and corresponding eigenvectors. In order to evaluate the correctness of the decomposition, MATLAB® is used as a golden reference. MATLAB's built-in function *eig(A)* computes the EVD of a given matrix $\mathbf{A}$ using any one of a series of library routines.

6.5.2 Performance Evaluation

The performance is measured in terms of the Signal to Noise Ratio (SNR), where the noise part refers to the amount of deviation of the obtained decomposition from an ideal case. With the decompositions obtained using the C-based Jacobi EVD and MATLAB the original matrix is reconstructed as given in equation (6.1). Using the input matrix $\mathbf{A}$ as the ideal case, SNRs for the C and Matlab based implementations are computed using the formulae given in equations (6.8) and (6.9).

$$SNR_M = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}|\mathbf{A}(i,j)|^2}{\sum_{i=1}^{N}\sum_{j=1}^{N}|\mathbf{A}(i,j) - \mathbf{A}_{r,M}(i,j)|^2} \tag{6.8}$$

$$SNR_C = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}|\mathbf{A}(i,j)|^2}{\sum_{i=1}^{N}\sum_{j=1}^{N}|\mathbf{A}(i,j) - \mathbf{A}_{r,C}(i,j)|^2} \tag{6.9}$$

Similarly, the SNRs for the $\mathbf{D}$ and $\mathbf{V}$ matrices are computed as well, with MATLAB as the ideal case.

$$SNR_V = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N}|\mathbf{V}_M(i,j)|^2}{\sum_{i=1}^{N}\sum_{j=1}^{N}|\mathbf{V}_M(i,j) - \mathbf{V}_C(i,j)|^2} \tag{6.10}$$

$$SNR_D = \frac{\sum_{i=1}^{N}|\mathbf{D}_{i,M}|^2}{\sum_{i=1}^{N}|\mathbf{D}_{i,M} - \mathbf{D}_{i,C}|^2} \tag{6.11}$$

We plot the obtained SNR against the condition number while comparing the SNRs obtained for different precisions. Condition number of a given matrix $\mathbf{A}$ in $\mathbf{Ax} = \mathbf{b}$ measures the sensitivity of the solution of this system of linear equations $\mathbf{x}$ to errors or changes in $\mathbf{b}$. $\mathbf{A}$ is said to be well-conditioned, and hence has a high condition number if changes in $\mathbf{x}$ due to changes in $\mathbf{b}$ are low and $\mathbf{A}$ is ill-conditioned or has a low condition number otherwise. Condition number of $\mathbf{A}$ can be computed by MATLAB using $cond(\mathbf{A})$. Since condition number is independent of machine precision, it provides a common ground for comparing performance results for various precisions.

6.5.3 Simulation Parameters

The parameters used during the simulations are:

- Number of receive antennas N_R
- Number of transmit antennas N_T: Note that $N_R = N_T$ since the matrices of interest are purely Hermitian which are square matrices. $N_R \times N_T$ is the size of the input matrix $\mathbf{H}$.
- Signal variance σ^2
- Number of realizations of matrix $\mathbf{H}$
- Precision parameters: In case of pseudo floating point, values for I and E are specified.

6.5.4 Results and Discussion: Part I

Simulation was carried out for double, single and pseudo floating point formats for matrix sizes of 2×2, 4×4 and 8×8. As described in section 6.2, all eigenvalue algorithms are iterative by Abel's theorem and due to the iterative nature of the EVD algorithms, a key aspect of concern is the convergence of the algorithm. Theoretically, the Jacobi EVD always converges [6]. However, due to insufficient precision leading to rounding errors, it may be possible that the algorithm may not always converge with finite arithmetic implementations. Preliminary simulations test for the convergence of the Jacobi EVD for all precisions. The results are documented in Table 6.1.

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 6.1: Convergence of Jacobi EVD implementation for all precisions

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

6.6 Dataflow model of the Jacobi EVD

6.6.1 Introduction

As described earlier in section 6.4, the overall objective of the Jacobi EVD project is to identify the minimum required internal precision of computation in order to obtain the required SNR. However, 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.

6.6.2 Related Work

Dataflow modeling has often been used in such precision analysis, most commonly in automatic floating to fixed point conversion of programs. Since high level languages like 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 lot of research has been focused on the automatic conversion of floating-point to fixed-point code ([24], [25], [26]). Some of these research works like [24], [27], [28] 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.

6.6.3 Dataflow model for Jacobi EVD

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. The computation of v_1 and v_2 given by the equations (6.5), (6.6) and (6.7) clearly indicate the presence of multiple square root, reciprocal and division operations. These are all represented as individual nodes in the graph as they tend to be more sensitive to precision. Each of the matrix multiplication steps in the Jacobi pseudocode (Algorithm 1) that are composed of multiply-and-add operations are also represented as nodes in the graph and are of higher granularity relative to the rest of the nodes. 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. As Algorithm 1 indicates, 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 till a suitable solution within specified error bounds has been obtained. 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. We use a statistical estimate for the number of iterations of the

Matrix size	Statistical estimate for unbounded loop iterations	No. of (p,q) index combinations	Total number of iterations of base graph
2×2	1	1	1
4×4	4	6	24
8×8	5	28	140

Table 6.2: Number of iterations of the base graph for Jacobi EVD

unbounded loop by simulating the double precision floating point implementation of Jacobi EVD over thousands of realizations and estimating the maximum number of iterations of the unbounded loop required for all matrix sizes. Using this information (documented in Table 6.2), the iteration count is set accordingly for the simulation of the graph behavior. The dataflow graph for the Jacobi EVD for one iteration of the algorithm is shown in Figure 6.2.

6.6.3.1 2×2 Matrix

As Table 6.2 indicates, only one iteration of the graph is required for a 2×2 matrix. Hence, the graph in Figure 6.2 with (p, q) as $(0, 1)$ is the dataflow graph for the 2×2 Jacobi EVD.

6.6.3.2 4×4 and 8×8 Matrices

For 4×4 and 8×8 matrices, the graph in Figure 6.2 is iteratively simulated 24 and 140 times respectively (from Table 6.2), 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

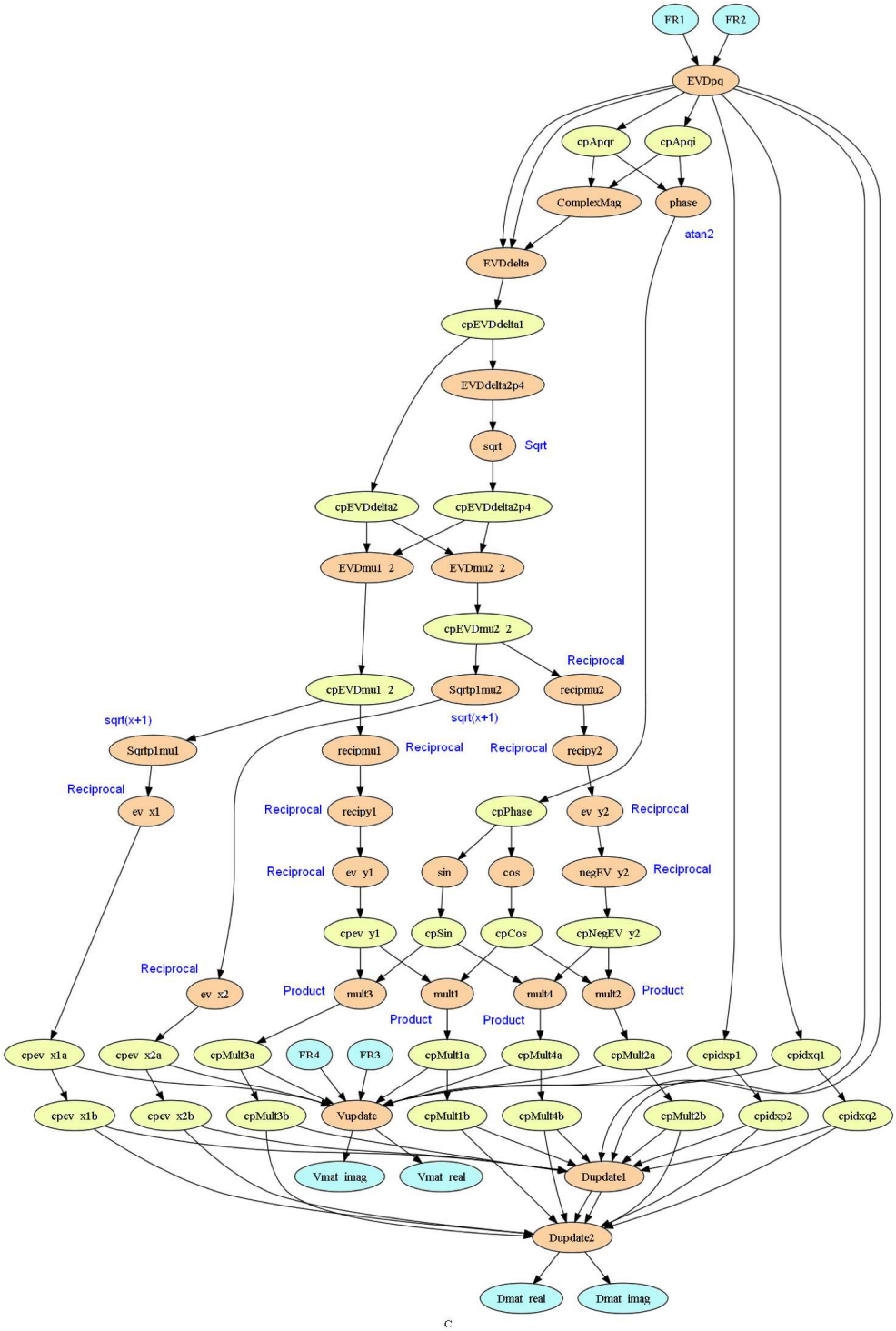

Figure 6.2: Dataflow graph for the 2x2 Jacobi EVD

programming the *runme* file.

6.7 Dynamic Range Analysis

The number of bits required to represent a data variable in a fixed-point format is the sum of the integer and fractional wordlengths. 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. [29] discusses some of these methods. In general, the *iwl* is estimated by computing the dynamic range of the data variable. The *fwl* can be obtained by simulating the program for all the possible *fwl*s and determining the SNR in each case and comparing it to the required SNR. Or, when using analytical methods, the computation error at every node can be expressed in terms of the *fwl* of the corresponding variable. The total error at the output can thus be expressed in terms of all the *fwl*s. Using optimization techniques the *fwl*s can be estimated such that the total error is less than a threshold determined by the required SNR.

In our work, we concentrate on the analysis of the dynamic range of different data variables. The goal is not to come up with exact numbers for the *iwl* and *fwl* for the different data, but instead to identify and understand the precision needs for different computations. 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.

Two methods can be used for evaluating the data dynamic range of an application. One method involves doing a floating point simulation of the application and statistically estimate the ranges of the data variables. This in general tends to be a more real estimate, but since it is simulation-based, some possible cases could be overlooked leading to overflow. 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 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. For our application, it is more suitable to adopt the latter approach so that all possible cases are taken into account.

Interval Arithmetic theory [30] can be used to determine data dynamic range in this method. 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.

6.7.1 Interval arithmetic for Jacobi EVD

Interval arithmetic is an arithmetic defined on sets of intervals, rather than sets of real numbers. Table 6.3 enlists the interval computations for the basic arithmetic operations of addition, subtraction, multiplication, division, squaring and square root. These are the most commonly used operations in the Jacobi EVD. Some of the other operations in Jacobi EVD that do not have straightforward formulae for

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 6.3: Interval Arithmetic

interval computation are *sine*,*cosine* and *atan2*. Instead, worst case ranges are used for these three functions. *sine* and *cosine* values always lie in the interval $[-1,1]$ and similarly *atan2* outputs angles in the range $[-\pi,\pi]$.

6.7.2 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 6.2 is verified for functional correctness using the CFDF simulator and sample test patterns with

appropriate *correct-output.txt* files.

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 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. The DIF file to describe this application is over 500 lines of code. It becomes even more invaluable when applied to larger applications. 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*. The dynamic range is computed by each actor based on the dynamic ranges of the inputs. Since the data ranges at all the nodes have to be analyzed and not just at the outputs, we also hook in *File Writers* to the intermediate nodes that record this information.

For the 2×2 matrix, this process is straightforward as only one iteration of the

graph is required. However for the 4×4 and 8×8 matrices, the number of required iterations is set in the CFDF simulator and the reconfigurability of the *runme* file is made use of to facilitate the feedback of output to input for successive iterations.

For example, the input files for data ranges for the input **A** matrix is written in *input-Areal.txt* and *input-Aimag.txt* (for the real and the imaginary parts of the complex number respectively). The output files for the **V** and **D** matrices are *output-Vreal.txt*, *output-Vimag.txt*, *output-Dreal.txt* and *output-Dimag.txt*. However, since we require 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*, instead of the above specified input and output files. The *runme* file shown in Table 6.4 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. If not for the availability of such a provision where the same base graph is used iteratively, there would have been a forcible need to unroll the loops. This is an incredibly tedious task for the 4×4 matrix with its 24 loops, leave alone the 8×8 matrix with 140. Thus, by exercising model-based design with simulation capabilities, development process is largely automated and simplified, and the development time has significantly reduced.

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

```
cp input-Areal.txt Dreal.txt

cp input-Aimag.txt Dimag.txt

cp input-Vreal.txt Vreal.txt

cp input-Vimag.txt Vimag.txt

../util/runtest

cp Dreal.txt output-Dreal.txt

cp Dimag.txt output-Dimag.txt

cp Vreal.txt output-Vreal.txt

cp Vimag.txt output-Vimag.txt
```

Table 6.4: *runme* file for multiple iterations in Jacobi EVD

computation node. Ultimately each node has an associated output file consisting of the corresponding dynamic range from every iteration. These files can be manually examined to make inferences on the required precisions, but it can become another tedious or even error-prone process in the case of a large application with numerous computational nodes, especially when there are too many iterations. Hence we automate this process by using an additional actor to read from these files and to generate the minimum and maximum values computed by any node over the course of the entire simulation.

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 6.5: Dynamic ranges for various data formats

6.8 Results and Discussion: Part II

Table 6.5 enlists the dynamic ranges offered by the different data formats under consideration in this work. It can be seen that double precision floating point offers very high dynamic range with the capability to express numbers as high as 10^{308}. As the number of exponent bits decrease, the dynamic range also drops off exponentially, with the exponent bit width of 6 offering a much more limited range of $[-10^9, 10^9]$.

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 6.5, 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. 6.5.4.

However, for the 4×4 matrix, Table 6.6 indicates multiple nodes with infinite dynamic range. Although this is a conservative estimate even for a worst-case

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 6.6: Dynamic ranges for computations in 4×4 Jacobi EVD

scenario, it is still an obvious indication that the dynamic range of some of these computations are dangerously high. Indeed, it is no accident that in the simulations carried out in Sec. 6.5.4, the 4×4 and 8×8 implementations converged only for double precision floating point which has a much larger 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 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 (6.6) and (6.7) 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 $(\sqrt{\delta^2+4}+\delta)^2$

$$\mu_1 = \frac{2}{\sqrt{\delta^2+4}-\delta} \qquad \mu_2 = \frac{2}{\sqrt{\delta^2+4}+\delta} \tag{6.12}$$

$$\delta = \frac{a-c}{|b|} \qquad \theta = \tan^{-1}\left[\frac{Im(b)}{Re(b)}\right] \tag{6.13}$$

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. *Prima facie* one of two methods can be used to overcome this problem. One option is to confirm by a more accurate analysis if double precision floating point's precision is sufficient to accurately compute $\sqrt{\delta^2+4}-\delta$ for all possible values of δ. The second option is to reformulate the equation for μ_1 in (6.12) 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 (6.12), 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

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 6.7: Dynamic ranges 4×4 Jacobi EVD with reformulation

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 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 $E \geq 8$. As was shown in the dataflow-based dynamic range

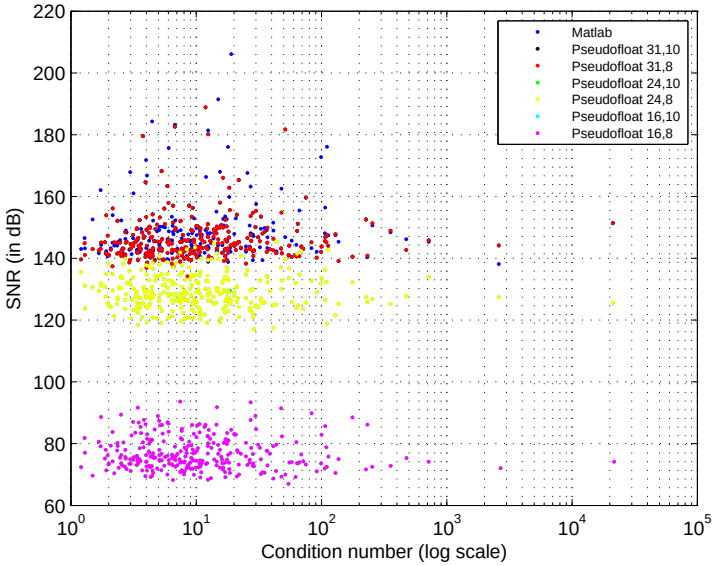

Figure 6.3: SNR in dB vs condition number for 2×2 Hermitian matrix

analysis, $E = 6$ did not provide sufficient range. The plots in figures 6.3, 6.4 and 6.5. show the SNR of the reconstructed matrix after eigenvalue decomposition as a function of the independent parameter, the condition number, for both MATLAB and C. The SNRs are expectedly higher for higher precisions, but in all cases are above the minimum required SNR of 50 dB. Thus the minimum required internal precision for computation for the Jacobi eigenvalue decompositon 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 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.

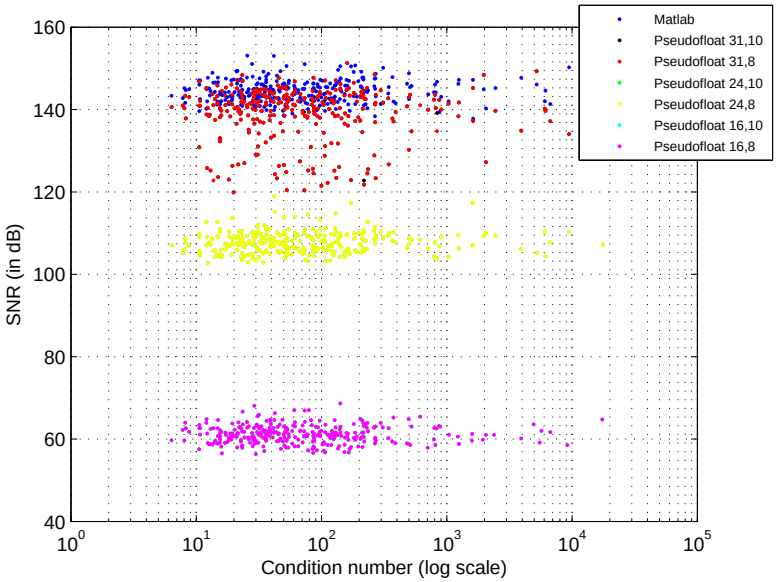

Figure 6.4: SNR in dB vs condition number for 4×4 Hermitian matrix

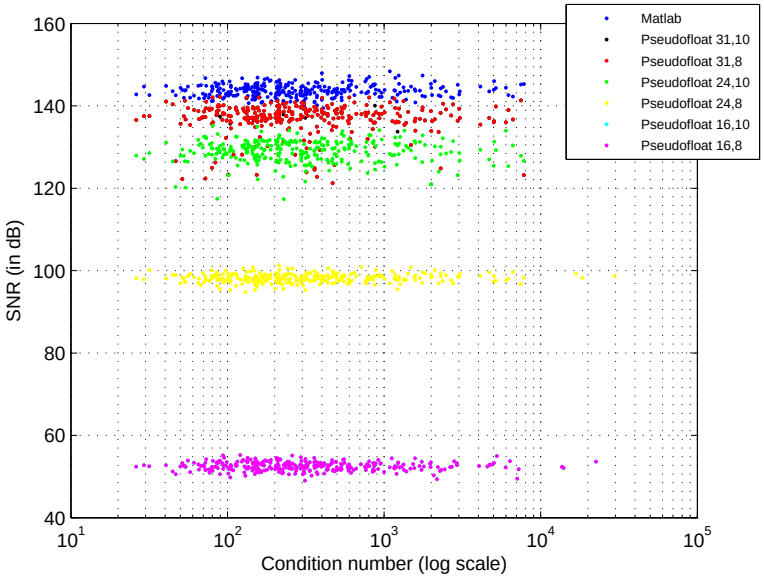

Figure 6.5: SNR in dB vs condition number for 8×8 Hermitian matrix

Chapter 7

Conclusion

With the rapidly growing number of application domains in modern embedded computing, many sophisticated applications requiring complex digital systems are emerging. This has led to the increased need for systematic and efficient design flows facilitating the use of heterogeneous programming environments, languages and target platforms making the overall development process more complicated, error-prone and tedious. The design and development of embedded systems is still largely done in an ad hoc fashion, and is especially sluggish in large collaborative projects with globally-distributed design teams. Best practices used do include model-based design which improves efficiency by using a common design environment across project teams and by linking designs directly to requirements.

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 introduce the DSPCAD Integrative Command Line Environment (DICE) as a realization of managing these enhancements to the design flow. DICE with its platform independent conventions facilitates the efficient management of design and test of cross-platform software projects, and 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 the use of this enhanced design flow with two case studies. The development of electronic systems in the Compact Muon Solenoid of the Large Hadron Collider is a collaborative project with multiple geographically distributed teams, and with each sub-system having separate teams for algorithm development, firmware and hardware development and so on. We actively use DICE's novel test framework on modules of a triggering system in the CMS, and demonstrate how the cross-platform model-based approach, automatic testbench creation and integration of testing in the design process alleviate the rigors of developing such a complex digital system. The use of a common framework like DICE for all the implementations helped standardize design specifications, communication interfaces, and ensured uniformity in data representation, apart from facilitating the reuse of tests.

In our second case study, we began with an exploration study into the performance versus precision metrics for the Jacobi Eigenvalue Decomposition (EVD). With the initial implementation in C leading to significant convergence issues, we identified the need to perform a fine-grained analysis on the precision for all the computations in the application. 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. Although the aim was to do 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. 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 this 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.

With these two case studies, the integration of DICE with model-based approach was highlighted to make the application design process easier, yet still rigorous and evolvable.

7.1 Future Work

In this thesis, we have presented a model-based design flow with a language independent software development framework. We have highlighted the benefits of using dataflow models to do analysis and verification of system modules. Future work along the same lines would involve using DICE for automating aspects of testing like test case generation using formal specification and derivation of hardware description language (HDL) test components from functional test structures developed at higher levels of abstraction using dataflow methods.

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] DICE for download, http://www.ece.umd.edu/DSPCAD/home/software.htm.

[3] 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.

[4] 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.

[5] C Lefevre. LHC: the guide. CERN-Brochure-2008-001-Eng, January 2008.

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

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

[8] 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.

[9] 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.

[10] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, Dallas, Texas, September 2005.

[11] 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.

[12] T. Dohmke and H. Gollee. Test-driven development of a PID controller. *IEEE Software*, 24(3):44–50, 2007.

[13] P. Hamill. *Unit test frameworks.* O'Reilly, 2004.

[14] S. Cozens. *Advanced Perl Programming, 2nd edition*, chapter 8. O'Reilly, 2005.

[15] 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.

[16] D. Acosta, M. Della Negra, L. Fo, A. Herv, and A. Petrilli. CMS physics: Technical design report. CMS Technical Design Report CERN-LHCC-2006-001, CERN, Geneva, Switzerland, 2006. Also CMS-TDR-008-1.

[17] P. Chumney, S. Dasu, J. Lackey, M. Jaworski, P. Robl, and W. H. Smith. Level-1 regional calorimeter trigger system for CMS. In *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics*, La Jolla, California, March 2003.

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

[19] I. Hen. MIMO architecture for wireless communication. *Intel Technology Journal*, 10(2), 2006.

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

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

[22] 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.

[23] D. S. Watkins. *Fundamentals of Matrix Computations.* John Wiley and Sons, Inc., 2nd edition, 2002.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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