ABSTRACT

Title of dissertation: MODEL-BASED DESIGN AND IMPLEMENTATION OF DEEP WAVEFORM ANALYSIS SYSTEMS

Yanzhou Liu
Doctor of Philosophy, 2018

Dissertation directed by: Professor Shuvra S. Bhattacharyya
Dept. of Electrical and Computer Engr., and
Institute for Advanced Computer Studies

Analysis of signals of relatively long duration, an area that is referred to as deep waveform analysis, is of increasing importance in instrumentation systems for wireless communications. For example, jitter measurement of deep waveforms must be performed during design and manufacturing tests for complex communications circuitry or equipment. As requirements for bit error rate performance become more stringent and data volumes increase, it becomes increasingly important and interesting to perform deep waveform analysis computations in long, or even temporally unbounded, waveforms.

Real-time response and limited hardware resources challenge the design methods of deep waveform analysis systems. Previous methods for deep waveform analysis required storage and computation across all samples of the waveform at once. However, as the amount of data in the waveform grows, and especially if the waveform is unbounded, storage of the waveform in its entirety becomes impractical.

The need to satisfy stringent real-time constraints, handle large volumes of data at high sample rates, and operate on resource-constrained platforms result in challenging problems in the development of advanced systems for deep waveform analysis. In this thesis, we have developed new design methodologies and design optimization methods to address these problems. The contributions of the thesis are geared toward handling large, possibly unbounded, signal data sets, and providing novel trade-offs among measurement accuracy, memory constraints, and real-time performance. Motivated by performance bottlenecks that we observed in our experimentation with deep waveform analysis, we have also developed a new model of computation for representing signal processing applications in a way that improves the efficiency of data communication between computational modules.

The main contributions of this thesis are summarized in the following.

(1). Design methodology for deep waveform analysis systems. We have developed a new design methodology for deep waveform analysis under limited resources. The methodology builds on the formalisms of dataflow-based design and implementation of signal processing systems. Our proposed methodology is shown to help significantly advance the prior state of the art in jitter measurement system design, and it forms an important foundation for later contributions that are presented in the thesis. Our approach is demonstrated through extensive experiments using actual measured data. Through its incorporation of high-level dataflow principles, the approach is suitable for efficient mapping to a variety of platforms, including multicore processors and graphics processing unit (GPU) devices for high performance signal processing.

(2). Design optimization for gapless deep waveform analysis. We have developed novel models and design optimization methods for addressing the real-time processing challenges of gapless deep waveform applications. A gapless signal processing application is characterized by one or more continuous streams of input data, where the data must be processed reliably without dropping any of the input samples. The strict real-time processing requirements for gapless deep waveform applications can be very challenging when input data rates are high, processing requirements are intensive, or the target platform is significantly resource constrained. The methods developed in this part of the thesis focus on optimizing the throughput of deep waveform analysis subject to the on-board memory constraints of a given data acquisition system interface, processor memory constraints, and the constraint of gapless processing.

(3). Passive-active flowgraphs for dataflow-based implementation. We introduce a new model of computation called passive-active flowgraphs (PAFGs), which complement conventional dataflow-based application representations. We have developed PAFGs to address important bottlenecks in dataflow graph implementation associated with communication between computational modules (dataflow graph vertices). We demonstrate the use of PAFGs as an intermediate representation for refining dataflow graphs into efficient implementations. We develop formal underpinnings of the PAFG model of computation, and introduce systematic transformation techniques for deriving and optimizing PAFG representations.

MODEL-BASED DESIGN AND IMPLEMENTATION OF DEEP WAVEFORM ANALYSIS SYSTEMS

by

Yanzhou Liu

Dissertation 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
Doctor of Philosophy
2018

Advisory Committee:
Professor Shuvra S. Bhattacharyya, Chair/Advisor
Professor Manoj Franklin
Professor Sennur Ulukus
Doctor Lee Barford
Professor Mihai Pop

Dedication

To my parents and all of my friends, thank you for everything.

Acknowledgments

Firstly I would like to sincerely express my gratitude to my research advisor Professor Shuvra Bhattacharyya for his invaluable support, guidance, patience and encouragement throughout my Ph.D. study. His continuous support and instructions helped me a lot in my research and thesis preparation. His technical advice and motivation and enthusiasm to research are indispensable to my completion of research and dissertation. I could not have imagined having a better research advisor for my Ph.D. study.

I also want to thank my committee members, Professor Franklin, Professor Ulukus, Doctor Barford, Professor Pop and Professor Goldsman for providing useful and insightful feedbacks and suggestions to my research.

I am grateful to Professor Rong Chen, Doctor William Plishker, Doctor Laihuei Wang, Doctor Kishan Sudusinghe, Doctor Shuoxin Lin, Ms. Lin Li, Mr. Jiahao Wu, Mr. Honglei Li, Mr. Jing Geng, Mr. Kyunghun Lee and other colleagues and research project collaborators for their generous help and support. I would also like to thank Marshall Plan Foundation and Fachhochschule Salzburg for providing a great opportunity for research and study in Austria.

Finally I give my special thanks to my great parents for their love and support during my study.

The research underlying this thesis was supported in part by U.S. National Science Foundation, Agilent Technologies and Keysight Technologies.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

ABC	Adjacent Buffer Coordination
API	Application Programming Interface
BER	Bit Error Rate
BMR	Buffer Memory Requirement
CDF	Cumulative Distribution Function
CFDF	Core Functional Dataflow
CPU	Central Processing Unit
DAQ	Data acquisition
DSP	Digital Signal Processing
EVM	Error Vector Magnitude
FCFS	First Come First Serve
FFT	Fast Fourier Transform
FIFO	First-In, First-Out
FIR	Finite Impulse Response
GEMS	Graph Elements
GPU	Graphics Processing Unit
HEFT	Heterogeneous Earliest Finish Time
IIR	Infinite Impulse Response
LIDE	The DSPCAD Lightweight Dataflow Environment
OpenCL	Open Computing Language
PAFG	Passive-active Flowgraph
PDF	Probability Density Function
PRBS	Psuedo-random binary signal
RMS	Root mean square
SDF	Synchronous Dataflow Graph
SISO	Single-Input, Single-Output
SMSS	Single-mode steady state

SSE Streaming Single instruction multiple data Extensions
SPS Samples per second

TDD Targeted DAQ Device
TIE Time Interval Error

UI Unit Interval

Chapter 1: Introduction

In the information era, more and more data needs to be processed in daily life, which greatly challenges the ways in which people store data, as well as the methods for processing and managing data efficiently and accurately. Our increasing ability to acquire data in the real world has resulted in continually escalating requirements for higher throughput and optimized memory management in design and implementation of embedded signal processing systems.

Many embedded signal processing applications involve the continuous acquisition and sampling of data, and the management of signals that have long duration. The time spans of continuous signal acquisition for such applications range from minutes, hours to days or even longer. Often, there is no well defined bound on the input signal duration that is known in advance.

Signals with long, possibly indefinitely-long durations and high sample rates are referred to as *deep waveforms*. Deep waveforms present major challenges for resource-constrained embedded implementation due to the large volumes of samples that need to be processed, and the need for reliable, real-time performance to avoid "falling behind" in the processing of the continuously-arriving input samples. This thesis is concerned with developing new models and methods for addressing the chal-

lenges of deploying deep waveform applications on resource-constrained platforms.

1.1 Jitter Measurement

As a concrete example of an important deep waveform application, we focus on *jitter measurement*, which has important uses in instrumentation for electronic system design, such as in measurement equipment for communication systems. Although we focus on jitter measurement for a significant part of this thesis, the core approaches developed in the thesis are not specific to this application, and can be adapted to other relevant deep waveform applications.

The jitter of a signal is defined as the short-term deviation of the signal's transition time from its ideal position in time [1]. Continuous, accurate evaluation of jitter is useful, for example, in computing the Bit Error Rate (BER) of a communication system. The BER is a widely-used performance metric for quality evaluation in communication systems. It is defined as the ratio of the number of bits received or transmitted in error to the total number of bits received or transmitted in the system [2]. BER can be estimated from the statistics of the jitter of eye crossings in the input signals (e.g., see [1]).

1.2 Dataflow Modeling

We employ dataflow-based modeling and analysis extensively in this thesis, and make new contributions to the application of dataflow methods in deep waveform applications. *Dataflow models of computation* are widely-used in the design

and implementation of signal processing systems. While dataflow represents a broad spectrum of models and methods that are used in many types of computer hardware and software systems, we focus specifically on dataflow as it relates to model-based design of embedded signal processing systems (e.g., see [3]). In this form of dataflow, signal processing applications are represented as directed graphs, where vertices (*actors*) represent computational tasks, and edges represent first-in, first-out (FIFO) buffers that store data that is communicated between actors [4]. Dataflow techniques are used in a wide variety of commercial tools for design and implementation for signal and information processing systems. Prominent examples include LabVIEW (National Instruments), MATLAB (MathWorks), SystemVue (Keysight Technologies), and Tensor Flow (Google).

1.3 Contributions of This Thesis

In this thesis, we develop new methods for dataflow-based design and implementation of deep waveform applications on resource-constrained platforms. Our methods are demonstrated on multicore platforms and hybrid CPU-GPU platforms, which integrate central processing unit (CPU) and graphic processing unit (GPU) devices. As mentioned previously, our methods are demonstrated concretely in the context of an advanced system for jitter measurement.

The contributions of this thesis involve three main parts. The first involves a new design methodology for deep waveform analysis under limited resources using dataflow graph techniques. This methodology is shown to help significantly advance

the prior state of the art in jitter measurement system design, and it forms an important foundation for later contributions that are presented in the thesis. Details and demonstrations of the methodology are presented in Chapter 3. This chapter is based on a paper that we have published in the Proceedings of the 2015 IEEE International Instrumentation and Measurement Technology Conference [5].

The second main contribution in this thesis addresses the problems of *gapless* deep waveform analysis, and systematic integration of data acquisition devices and their associated real-time constraints into model-based design and implementation of deep waveform applications. A gapless signal processing application is characterized by one or more continuous streams of input data, where the data must be processed without gaps. In this context, by "without gaps", we mean continuous processing without dropping any of the input samples. The strict real-time processing requirements for gapless deep waveform applications can be very challenging when input data rates are high, processing requirements are intensive, or the target platform is significantly resource constrained.

In the second part of this thesis, we present novel models and design optimization methods for addressing the real-time processing challenges of gapless deep waveform applications. Details of these models and methods are presented in Chapter 4 and Chapter 5. We have published a preliminary version of this work in the Proceedings of the 2016 IEEE International Instrumentation and Measurement Technology Conference [6].

The third main contribution of this thesis addresses a fundamental limitation of dataflow semantics in design and implementation of signal processing systems.

This limitation involves inter-actor communication patterns that depart from the single-input, single-output (SISO) interface and FIFO behavior that are defined for dataflow edges (e.g., see [7]). When these types of communication pattern are mapped into hardware or software using pure dataflow semantics, the resulting implementations can be very inefficient. At the same time, these types of communication patterns are important in deep waveform applications.

To address this problem, we introduce a novel application modeling concept called *passive blocks*, which generalize the FIFO buffers of dataflow graphs. Like dataflow buffers, passive blocks are used to store data during the intervals between its generation by producing actors, and its use by consuming actors. However, passive blocks can have multiple inputs and multiple outputs, and can incorporate operations on and rearrangements of the stored data subject to certain constraints.

We introduce a new model of computation called *passive-active flowgraphs* (*PAFGs*), which complement conventional dataflow-based application representations. We demonstrate the use of PAFGs as an intermediate representation for refining dataflow graphs into efficient implementations. We develop formal underpinnings of the PAFG model of computation, and introduce transformation techniques for deriving and optimizing PAFG representations. We demonstrate the utility of PAFG-based modeling and optimization through application case studies involving jitter measurement and error vector magnitude monitoring, which are two important deep waveform applications used in instrumentation for communication system design.

Details on the PAFG model of computation and its application to signal pro-

cessing system design are presented in Chapter 6. This chapter is based on a paper that has been accepted for publication and is to appear in the Proceedings of the 2018 IEEE Workshop on Signal Processing Systems [8].

1.4 Outline of Thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces background on the different research topics discussed in the thesis. The chapter covers fundamentals of dataflow models for signal processing system design; background on specific software tools that we have used for prototyping and experimentation; and background on the jitter measurement application that we study in depth throughout the thesis. As described in Section 1.3, the three main contributions of the thesis are presented in Chapter 3, Chapter 4/Chapter 5, and Chapter 6, respectively. Chapter 7 concludes with a summary of the contributions of this thesis, and discusses directions for future work that are motivated by these contributions.

Chapter 2: Background

In this chapter, we introduce background on core concepts that are applied to the work in this thesis.

2.1 Synchronous Dataflow

Dataflow is a form of model-based design that is employed for design and implementation in many areas of signal processing [3,4]. By providing formal methods to represent and analyze the flowgraph structures within signal processing applications, dataflow methods help to enhance the efficiency and reliability of implementations, and assist in the retargeting of designs across different hardware platforms.

A dataflow graph is a directed graph in which vertices, called *actors*, represent computational tasks, and edges represent the communication of data between actors. More specifically, each edge $e = (u, w)$ in a dataflow graph represents a first-in, first-out (FIFO) buffer that stores data as it passed from the output of actor u to the input of actor v. Actor u is referred to as the *source actor* of edge e, and actor w is referred to as the *sink actor* of e.

Ports connect actors and edges. A port can be either an *input port* or an *output port*, depending on whether the actor consumes data from the incident edge or

produces data on it, respectively. Each unit of data that passes along a dataflow edge edge is referred to as a *token*. Tokens can have arbitrary data types. The execution of an actor in a dataflow graph is decomposed into discrete units of execution, which are referred to as *firings* of the actor.

Synchronous dataflow (*SDF*) is a restricted form of dataflow in which the number of tokens produced on each output edge is constant across all firings of a given actor, and similarly, the number of tokens consumed from each input edge is constant [9]. SDF graphs are widely used in the design and implementation of signal processing applications, and many kinds of useful analysis and optimization techniques have been developed for this class of graphs [3]. An important property of SDF graphs is that, if certain well-defined consistency properties hold, they can be implemented to operate on unbounded-length input streams with deadlock-free execution, and bounded memory requirements that can be determined at compile-time [9].

Figure 2.1 shows an example of a simple SDF graph. The annotations on the edges represent the *production and consumption rates* of the actors — that is, the constant numbers of tokens that are produced and consumed on each firing. For example, when actor Y is fired, c_Y tokens are consumed from edge e_1, and p_Y tokens are produced onto edge e_2. The production and consumption rates of an actor are collectively known as the *dataflow rates* of the actor.

An important task involved in implementing a dataflow graph is the task of scheduling the graph — that is, the process of determining the assignment of actor firings to processors and the order in which multiple firings assigned to the same pro-

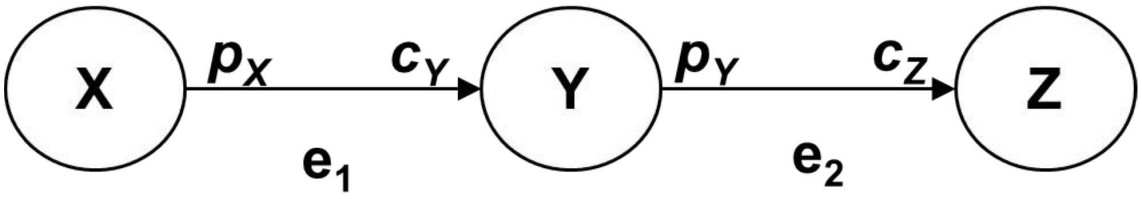

Figure 2.1: A simple synchronous dataflow graph.

cessor will execute [10]. Scheduling techniques can be distinguished based on when the assignment and ordering tasks described above are performed (e.g., at compile time versus at run time). This leads to important classes of scheduling strategies, including static, dynamic, quasi-static, and static assignment strategies [11]. Various useful abstractions have been developed for representing dataflow graph schedules, and providing a basis for analysis, transformation, and software synthesis of schedules [12, 13].

2.2 Core Functional Dataflow

In this thesis, we build on the methodology of dataflow-based design and implementation of signal processing systems, and we apply a specific form of dataflow-based design referred to as core functional dataflow (CFDF). In this section, we will provide background on CFDF graph.

Core functional dataflow (*CFDF*) is a form of dataflow that is useful for applying dataflow-based design methods to a wide variety of signal processing applications [14]. In CFDF, the behavior of an actor is decomposed into a set of *modes* such that any given firing corresponds to a single mode, and in each mode, the number of tokens consumed from each input edges is constant, and similarly, the

number of tokens produced on each output edge is constant. These numbers of tokens consumed and produced by the actor in each mode are *consumption rates* and *production rates*, respectively. More specifically, given a mode m of a CFDF actor A, the consumption rate associated with m and input edge e_i is the number of tokens consumed by A during m from e_i. Similarly, if e_o is an output edge of A, then the production rate associated with m and e_o is number of tokens produced by A during m on e_o.

As part of execution of a given mode m, a CFDF actor must determine the *next mode* in which the actor will operate. The next mode determines the mode that will be active during the next firing of the actor. Although production and consumption rates for any given mode are constant, the rates can vary across different modes, which allows for modeling and design of *dynamic dataflow* behavior. Additionally, the next mode of an actor can be data-dependent — i.e., it is not necessarily a function only of the current mode.

Implementation of a CFDF actor requires implementation of two specific functions, which are called the *enable* and *invoke* functions of the actor. Each of these functions take as an argument in the *actor context*, including the *mode state*, and the current values of all parameters of the actor. Here, by the mode state, we mean the current mode of the actor if the actor is currently firing or the next mode of the actor (as determined by the previous mode or an initialization process) if the actor is currently dormant (not firing). The enable function returns a Boolean value indicating whether or not there is sufficient data on the input edges and sufficient empty space on the output edges to allow the actor to fire based on the mode spec-

ified by the mode state. On the other hand, the invoke function executes a single firing of the associated actor in the mode specified by the mode state, and changes the value of the mode state based on the next mode that is determined as part of the firing.

The separation of enable and invoke functions helps to modularize the design of CFDF actors, and to implement more efficient and predictable scheduling techniques. It is also important to note that it is not always necessary to call the enable function at run-time prior to executing the invoke function. Compile-time analysis of dataflow properties may provide guarantees about data and output space availability for some proper subset of an actor's firings or for all firings. In such cases, the invoke function can safely be executed without first using the enable function to validate fireability. For example, when implementing static scheduling techniques (e.g., see [3, 9]) for CFDF graphs (when such schedules exist), there is no need to use the enable function at all.

In addition to applying CFDF semantics as a specific form of dataflow in this thesis, some of the techniques that we apply are related to parametric dataflow modeling techniques, such as parameterized dataflow [15]; parameterized sets of modes [16]; and parameterized and interfaced dataflow meta-model [17].

2.3 LIDE — The DSPCAD Lightweight Dataflow Environment

LIDE, which stands for the DSPCAD *LIghtweight Dataflow Environment*, provides a flexible and lightweight software environment for dataflow-based design and

implementation of DSP applications [18]. LIDE contains collections of pre-designed libraries for dataflow graph elements, including actor and edge implementations. These existing graph elements or *gems*, which stands for "Graph EleMentS", provide useful building blocks for constructing dataflow graphs, and also provide examples that designers can use as references or templates when extending LIDE with their own custom-designed actor and edge implementations.

A core part of LIDE is a compact set of abstract application programming interfaces (APIs) for implementing, integrating, and scheduling dataflow actors based on CFDF semantics. These APIs are abstract in the sense that they are defined in terms of mathematical dataflow principles, and are independent of any particular language for programming the actors (*actor design language*). The compact and abstract nature of this set of APIs makes it easy to map it into a wide variety of actor design languages. Presently, the set of actor design languages supported in LIDE includes C, CUDA, OpenCL and Verilog.

An actor in LIDE can be viewed as an abstract data type or a class (if an object oriented language is employed as the actor design language). Four interface functions are used in LIDE in the implementation of each actor — the construct, enable, invoke, and terminate functions. The construct function is used to create an instance of the actor, including all of the associated memory allocation and initialization. The enable and invoke functions implement their counterparts as defined by CFDF semantics. The terminate function is used to carry out any tasks, such as deallocation of memory, that are appropriate when an actor is no longer needed — e.g., if the enclosing application has terminated or if the dataflow graph

is being reconfigured in such a way that the actor will not be used any more.

2.4 Jitter Measurement

In this thesis, we investigate a jitter measurement system as a case study for dataflow-based design and implementation of real-time deep waveform analysis. Jitter is defined as the deviation of a timing event in a signal from its ideal appearance in time [1]. We specifically study jitter measurement in the context of wireless communications, where bit error rate (BER) is an important metric for assessing overall system performance.

Previous work on jitter measurement has been performed in conjunction with use of clock recovery algorithms, and measurement based on a reference clock signal (e.g. see [19]). Loken presents a fixed-frequency clock recovery algorithm for jitter measurement [20]. This work assumes a two-state digital signal and a duty cycle that is approximately equal to 50%. We maintain these assumptions in our research, as they are applicable in our primary application context of deep waveform analysis for wireless communication systems.

There are various limitations in state-of-the-art methods for jitter measurement that our proposed research seeks to overcome. For example, some of the previous research on jitter measurement relies on a stable reference clock period. This limits applicability of the approaches since a stable reference clock period is not always available for measuring jitter. We overcome this limitation in our proposed research by integrating clock period estimation with jitter measurement analysis

— that is, our proposed analysis system first estimates the clock period, and then estimates the jitter in the input signal based on the estimated clock period.

Loken has developed an algorithm for jitter measurement from fixed-frequency signals that does not require a reference clock [20]. However, this algorithm has the limitation of being a "swallow and wallow" technique for signal analysis. This swallow and wallow characteristic requires all of the signal data to be stored in memory before the computation for jitter measurement is initiated. Such swallow and wallow approaches require large amounts of memory to analyze long signals, and limit the length (number of samples) or "depth" of deep waveform signals that can be handled.

In this thesis, we have demonstrated new window- and dataflow-based design and implementation techniques for jitter measurement that overcome this swallow and wallow limitation, while also maintaining the feature that a reference clock is not needed for the underlying measurement algorithm. We apply a windowing method that partitions the signal to be analyzed into multiple subsets, and allows processing of windows to proceed in real-time without need for all of the signal data to be available at once. We apply dataflow methods to model the signal flow characteristics of the algorithm that analyzes the signal windows, and develop efficient design transformations using the dataflow model to optimize memory cost and real-time performance.

Chapter 3: Dataflow Design for Deep Waveform Analysis Systems

Deep waveform analysis applications are increasingly important in signal processing application areas, such as wired and wireless communication and biomedical instrumentation. In this chapter we begin to examine system design challenges for a specific deep waveform analysis application, jitter measurement, that we use throughout the thesis as a concrete case study to develop and demonstrate our research contributions.

The measurement of jitter is key when verifying the design or performing manufacturing test of ever more complex digital communications circuitry or equipment. As the requirements for bit error rates (BER) become more stringent and data volumes increase, it becomes increasingly important and interesting to measure timing jitter in long, or even temporally unbounded, waveforms.

Previous methods for doing constant rate clock recovery and jitter measurement required storing and computing all samples of the waveform at once. As the waveform grows, and especially if the waveform is unbounded, this storage of the waveform in its entirety becomes impractical. In this chapter, we demonstrate the transformation of the previous methods to a dataflow method where the entire waveform need never be stored.

The new method has been tested on actual measured data. Through its incorporation of dataflow principles, the new method is suitable for efficient mapping to a variety of platforms, including multicore and field programmable gate array platforms for high performance signal processing. Intermediate measurement results converge toward those obtained in the original method. The final measurement result, the jitter standard deviation, agrees with the original method to within well under one percent. Thus, a small amount of additional measurement error is added in order to remove the restriction that the entire waveform fit into memory.

Material described in this chapter has been published in [5].

3.1 Introduction

Complex systems often include or are connected to other complex systems by ever faster communications links with lower rates of error. Communications errors are often due to timing errors or jitter. Thus, the measurement of jitter is key when verifying the design or performing manufacturing test of digital communications circuitry or equipment. As the requirements for bit error rates (BER) become more stringent and data volumes increase, it becomes increasingly important and interesting to measure timing jitter in signals of longer duration (that is, of so-called "deep waveforms"). Measuring deep waveforms both (1) increases the chances that rare events leading to communications errors will be captured and identified [21], and (2) allows the statistical estimation of the tails of jitter probability distributions, which in turn permits better extrapolation of BER. [22]

Commercially available instrumentation is capable of measuring communications and/or timing signal waveforms containing billions of samples. Here, we are concerned with constant rate clock recovery, where the clock is assumed not to change its period during the course of the measurement. Any deviation of the timing of the communication signal being measured from that constant rate clock is to be considered jitter. This is the case for many communications physical layers. (Exceptions include example in designs utilizing spread spectrum clocks.) Clock recovery and computation of jitter statistics are straightforward and computationally inexpensive when considered on a per-sample basis.

As discussed in Chapter 2, performing these tasks on deep waveforms takes considerable computing time. One way to address this computation speed issue is through parallel computing. Previous work [20] described and demonstrated the measurement capability and computational performance of a parallel algorithm for constant rate clock recovery from a two-logic state digital waveform.

That method is suitable for implementation on multicore central processing units (CPUs) and graphics processing units (GPUs). The resulting computational speedups permits its use on waveforms with millions of samples. However, that algorithm must store not only the entire waveform in memory but also a number of working arrays with lengths comparable to that of the number of samples in the waveform. As a result, even a multi-gigabyte memory is inadequate to do constant rate clock recovery on more than a few hundred million samples.

Figure 3.1 demonstrates typical input signals that are analyzed in this chapter. The figure shows a small portion of a waveform with two logic states including high

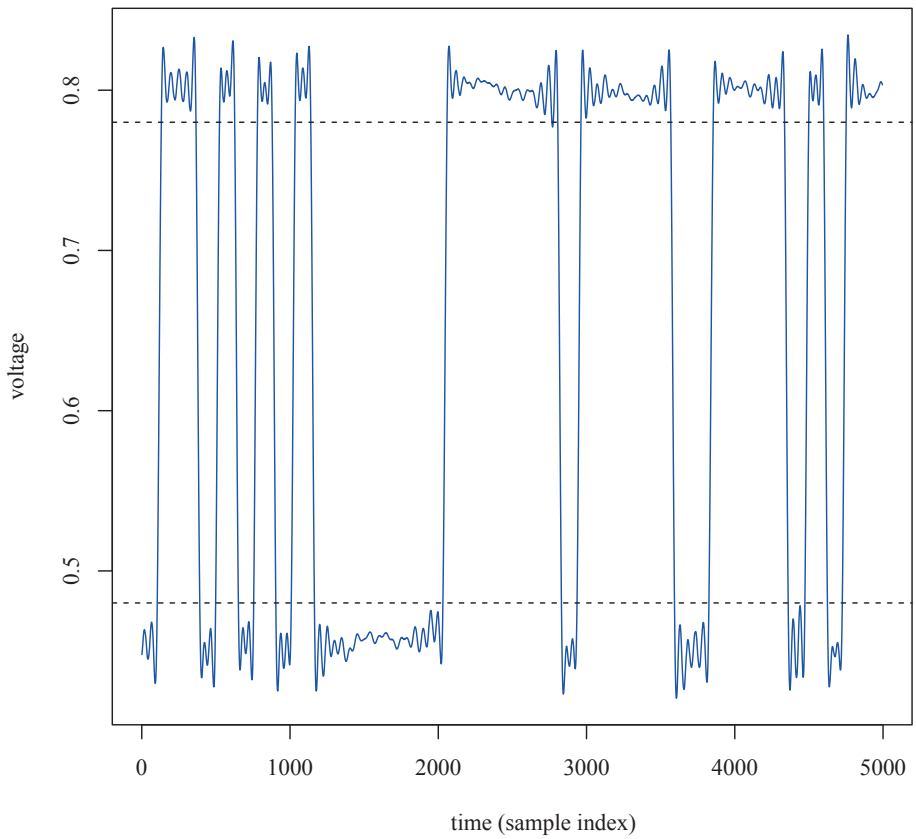

Figure 3.1: An illustration of typical input signals that are analyzed in this chapter.

and low states. The two dashed horizontal lines indicate the thresholds $l = 0.48$ and $h = 0.78$ of the low and high logic states, respectively.

An alternative to storing a large waveform in memory is to use a dataflow approach. That is, a programming model is used where samples and intermediate results move on the edges of a directed graph and computations on these data occur at the graph vertices, called actors. Languages commonly used in measurement that provide variants of a dataflow framework include LabVIEW [23], VEE Pro [24], and Simulink [25]. When using a dataflow programming tool, the signal is processed sequentially. Neither it not intermediate computations are not (and typically can not be) stored in their entirety. This feature is attractive when analyzing deep

waveforms, when all such data will not fit into the available memory. Another important advantage of the dataflow formalism is that the dataflows exactly specify the data dependencies between computations, and so the computations of the actors can in many cases automatically be scheduled on multicore processors [3], yielding a parallel implementation without the need for explicitly parallel programming. Vector-mode computation is sometimes still possible inside actors [26], yielding a second level of parallel speed-up.

Despite such advantages, there is a subtle incompatibility between the data flow approach and that of storing and computing on the entire waveform. When the whole waveform is available, measurements can be made of the waveform based on its entirety that are used in later measurement stages. Sometimes this sort of design is referred to as the "swallow and wallow" approach, because the entire waveform is stored (or "swallowed") and then computed on (or "wallowed over"). For example, in the present case, the voltage statistics of the whole waveform can be used as recommended by the relevant IEEE standard [27] to arrive at the voltage thresholds used to determine the low and high voltage levels of the signal and the timing of the signal for purposes of clock recovery and then jitter measurement. When using the dataflow paradigm, it is no longer possible to take measurements of the entire signal and use them in computing derived measurements.

Thus, a measurement algorithm designed for the swallow and wallow approach can not in general produce identical results when modified to be used with the dataflow paradigm. It would seem that the best that can be done is to design a dataflow algorithm that closely approximates the behavior of the swallow and

wallow algorithm. Note that this must be a dynamic approximation. Suppose that the dataflow algorithm has fully processed the first n samples of the waveform. The best it can do is to approximate the measurement results of the swallow and wallow algorithm if it were applied to the same n samples. Since differences between the "swallow and wallow" and dataflow measurement results are inevitable, the differences should be categorized and documented.

In this chapter we present a case study of creating such a dataflow algorithm from a swallow and wallow algorithm, where the measurement problem being solved is constant-rate clock recovery. First, we present the proposed dataflow method along with a discussion of how and where that method must deviate from the swallow and wallow algorithm of [20]. Then we present the results of a study comparing the measurement results of the two methods in the light of the approximation criterion of the previous paragraph.

3.2 Proposed method

The swallow and wallow algorithm takes considerable computing time and requires a large amount of memory. As we discussed in Section 3.1, application of the dataflow paradigm can help to reduce memory requirements significantly. In this section, we take the case of clock recovery [20] as an example to discuss how to develop a dataflow method, based on the swallow and wallow algorithm, that approximates the measurement results of that algorithm in the sense described above.

3.2.1 Dataflow Modeling

To create a dataflow model for the clock recovery application, we analyze the application to extract its high-level signal flowgraph structure. The basic steps for clock recovery [20] can be summarized as follows. The first step is to determine the voltage thresholds that correspond to high and low signal levels. The next step is to determine the complete set of transitions across these thresholds in the input signal. The third and last step is to estimate, from this set of transitions, the clock period; refine this estimate by rounding the differences of neighboring transitions in terms of the estimated clock period; and apply linear fitting to further refine the result. The input signal to this three-step clock recovery process is decomposed into a sequence of overlapping windows, where the three processing steps are applied iteratively to successive windows.

The resulting dataflow graph is shown in Figure 3.2. It is a dataflow model of the swallow and wallow algorithm for clock recovery. Table 3.1 lists the actors employed in the dataflow graph of Figure 3.2, and briefly summarizes the functionality of each one. The dataflow graph in Figure 3.2 is annotated with the *production and consumption rates* of the edges (flowgraph connections) in the graph. Such dataflow properties associated with edges are important when analyzing dataflow graphs to construct schedules and derive other parts of implementations. Given a dataflow edge e that is directed from an actor x to an actor y, the production rate of e is the number of tokens (data values) that is produced onto e in each firing of x, and similarly, the consumption rate of e is the number of tokens consumed from e during

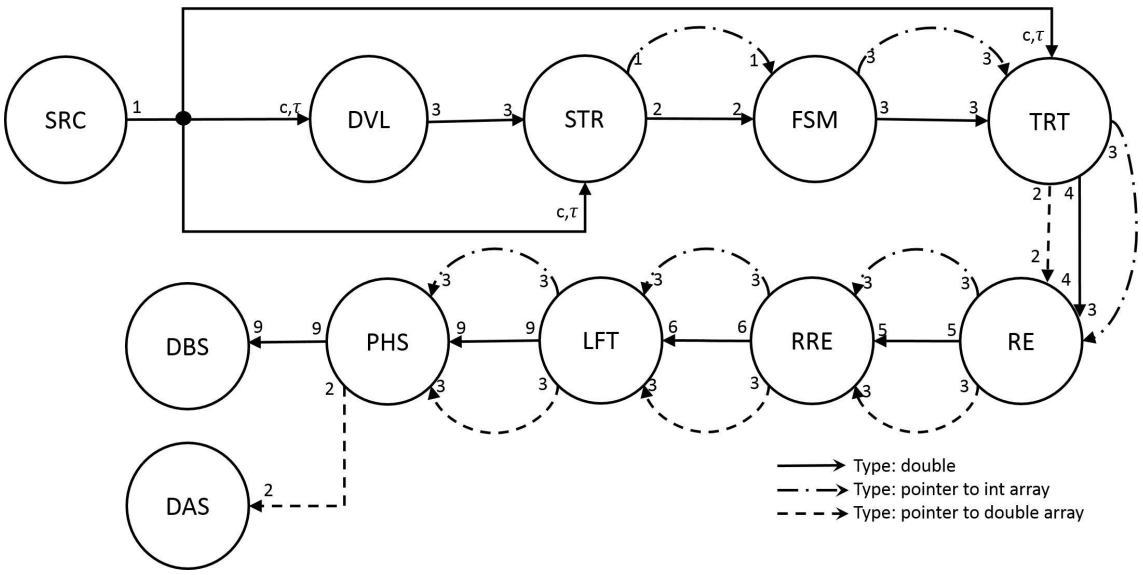

Figure 3.2: Dataflow model for signal frequency recovery and jitter computation

each firing of y.

The input port p of the DVL actor is annotated with two values $[c, \tau]$, where c is the consumption rate of the DVL actor from the input edge associated with p, and τ is another dataflow-edge-related attribute called the *threshold* of the edge [28]. The threshold specifies the number of tokens that must be present on the corresponding input FIFO before the actor can fire. In general, the consumption rate of a port is less than or equal to the threshold. Consumption rate / threshold pairs are also indicated at certain input ports of the STR and TRT actors in Figure 3.2. In Figure 3.2, the consumption rates and thresholds for all edges are equal except for the input edges of the DVL, STR and TRT actors that are annotated with pairs of values on the associated input ports. More details about the key actors in Figure 3.2, including the DVL actor, are discussed in Section 3.2.2.

Table 3.1: Summary of actors in the dataflow graph of signal frequency recovery

Actor	Description
SRC	Source. Load data from input file.
DVL	Determine Voltage Level. Sort the input data in the current window and determine the high and low voltage thresholds.
STR	State Representation. Perform analog-to-digital conversion; assign state to data in the current window of the input signal.
FSM	Finite State Machine. Determine transitions from high to low voltage states or low to high states.
TRT	Compute Transition Time. Compute the transition time for each transition in the current window.
RE	Rough Estimation. Derive an preliminary estimate of the clock period.
RRE	Refine Rough Estimation. Refine the rough estimation of the clock period to improve its accuracy.
LFT	Linear Fitting. Further refine the estimated clock period by linear fitting.
PHS	Phase. Compute the phase and time interval errors at the current transition using the refined clock period estimate.
DBS	Double Sink. Store double precision numeric data to an output file; each input token encapsulates a scalar, double precision value.
DAS	Double Array Sink. Store double precision numeric data to an output file; each input token encapsulates an array of double precision values.

3.2.2 Implementation in LIDE-C

In this section, we discuss our implementation of the dataflow graph (Figure 3.2) for clock recovery. To develop this implementation, we have used *LIDE* (*Lightweight Dataflow Environment*) [18, 29]. In our implementation of clock recovery, we employ *LIDE-C*, which provides APIs for implementing signal processing dataflow graphs using the C language. In general, a LIDE-C implementation includes C implementations of actors, edges, and a schedule to execute the overall dataflow graph.

Next, in Section 3.2.2.1 and Section 3.2.2.2, we discuss details of two of the most critical actors in our implementation.

3.2.2.1 DVL

Each firing of the DVL actor examines a window of samples from the input signal, sorts the values of these samples, and determines high and low voltage thresholds based on the results of this sorting operation. The parameters of the DVL actor include the window size W_s, and the amount of overlap O_p between successive windows. The consumption rate and threshold for this actor are, respectively, $W_s \times (1 - O_p)$, and W_s. Intuitively, this means that before the actor can fire, a full window of data must be available at the input, but only part of this input is consumed during the firing — the rest remains in the input FIFO to be processed as part of the next (overlapping) window during the subsequent firing.

Since the applied voltage thresholds will influence the value of the estimated

clock period, it is essential to determine an appropriate voltage threshold. In our implementation, we consider two alternative methods for determining the voltage threshold. Selection between these two alternative methods is controlled by a third parameter Y of the DVL actor. The first method ($Y = 1$) is to use the sorted result from the current window to dynamically determine the voltage threshold associated with the current firing of the DVL actor. The second method ($Y = 2$) is to fix the voltage thresholds across all iterations based on the sorted result from the first window. We experiment with both of these methods, and results of this experimentation are discussed in Section 3.3. From these experiments, we find that the accuracy for $Y = 2$ is slightly better than that for $Y = 1$; however, the difference is so small for the examined application scenarios that it is not worth the added complexity to implement and apply the $Y = 2$ case.

3.2.2.2 LFT

This actor optimizes (further refines) the result of clock period estimation by linear least square fitting. The sequences of transition times and phases at the transition times are the two data streams used in this linear fitting operation. The phase at a given transition time t depends on t and the estimated clock period. Since the total number of transitions is proportional to the number of windows processed, performing linear fitting across all transitions computed is computationally very expensive. To make this process more efficient, we select a subset of data for linear fitting. We allocate a buffer in the LFT actor to store the selected transition times,

and the corresponding phases to be used for linear fitting. There is a variety of methods to select data for linear fitting (e.g., see [30]). In our design of the LFT actor, we apply a rule for selecting data in which the selected data is composed both of data from previous transitions and from the current input signal window.

3.2.3 Scheduling the Dataflow Graph

As discussed in Section 3.2.1, scheduling is an important step in simulating or implementing a dataflow graph application model. There are many possible schedules for our dataflow model of the clock recovery application. For the experiments in this chapter, we use a relatively simple, sequential (single-processor) schedule since the main objective in this chapter is to validate and study functional properties of the proposed clock recovery system. Applying the developed dataflow model to derive fast implementations (with correspondingly fast schedules) is a useful direction for future work that builds naturally on the developments of this chapter.

The specific schedule that we used in our experiments is

$$SRC\ (W_s\ DVL)\ STR\ FSM\ TRT\ RE\ RRE\ LFT\ PHS$$

$$DBS\ DAS$$

where the parenthesized term $(W_s\ DVL)$ represents a *schedule loop* that executes actor DVL a number of times in succession that is determined by the window size W_s. In addition to providing a simple execution pattern that is suitable for rapid prototyping, this schedule is efficient in terms of buffer memory requirements —

i.e., the amount of memory required to implement the dataflow graph edges. This buffering efficiency is useful for our computations because our experiments involve input signals that contain large numbers of samples (in the range of 10^7 to 10^9 samples in each experiment).

3.3 Results

The above dataflow implementation was tested using actual measured data acquired and processed using the apparatus in Figure 3.3. Two waveforms were used for testing. One is the signal used in [20] to test a GPU implementation of the swallow and wallow algorithm for the present measurement. That waveform comprises approximately 1.6×10^7 samples of a pseudo-random binary (PRBS) signal. The first few thousand samples of that signal are shown in Figure. 3.1. That waveform was chosen for testing because the outputs of the dataflow method, especially the corrected clock period and phase, could be directly compared with those obtained by the previous swallow and wallow implementation in [20].

A second PRBS waveform comprising approximately 2×10^9 samples was used to test that dataflow implementation's ability to operate on waveforms too large to process in memory all at once. The proposed method produced the expected results.

As was discussed in the introduction, a difficulty of modifying a swallow and wallow measurement algorithm into a dataflow on is the presence in the original algorithm of intermediate measurement results that depend on the entire acquired waveform. In the dataflow algorithm such intermediate measurement results can

only depend on the prefix of the waveform that has previously been processed. Thus, these intermediate measurement results in the dataflow method can only approximate those of the swallow and wallow method. The question therefore arises as to whether this approximation is sufficiently good or whether it negatively influences the accuracy of the final measurements.

Here, in the swallow and wallow algorithm percentile levels of all the sample voltages are used to compute the low, medium, and high voltage thresholds used to determine the locations of the state transitions. In the dataflow method this is done in the actor DVL. In that actor, two different methods were tried to obtain the thresholds. One was to use the voltages of the first window to compute the thresholds and then use those thresholds for all succeeding windows. The second approach was to re-compute the thresholds for each window. In order to investigate the sensitivity of the measurement results to the choice between these methods, processing of the measured signals was done twice, one using each of these methods.

The result of using the proposed method with 10 windows on the signal with 1.6×10^7 samples is shown in Figure 3.4 and Figure 3.5. Figure 3.4 shows evolution of the measured value of the clock period from the waveform with 1.6×10^7 samples. Its beginning is shown in Figure 3.1. The dashed, horizontal line in Figure 3.4 indicates the clock period measured in [20] as a reference for accuracy comparison.

Figure 3.5 demonstrates evolution of the measured value of clock phase offset from the waveform with 1.6×10^7 samples using the same input data as in Figure 3.4. The dashed, horizontal line in Figure 3.5 shows the phase offset found in [20]. Both Figure 3.4 and Figure 3.5 compare results with the results reported in [20].

Clock recovery was not sensitive to whether or not the voltage thresholds were fixed or varied according to each window's voltage statistics. Both the clock period and phase offset converge toward the values obtained from the swallow and wallow algorithm, as was desired. The final measurements from the dataflow algorithm are not identical to those obtained by the swallow and wallow algorithm. Some difference is expected because the two methods use slightly different voltage thresholds and perform the final correction by linear fitting using different time interval errors. However, the final clock periods and phase offsets produced by the two methods have absolute relative differences of under 1×10^{-8} samples per clock cycle and 1×10^{-2} samples, respectively.

Figure 3.6 is a scatter plot matrix [31] that shows the correlations of the time interval errors (TIEs) measured using each of the two tested variants of the proposed method and the method of [20]. Here, we index the three different methods under investigation as 1, 2, 3. Index 1 corresponds to TIEs obtained using the method of [20] (labeled "SW"). Index 2 represents TIEs obtained using the proposed method with voltage thresholds fixed ("DF (fixed thresh)"). Index 3 represents TIEs obtained using the proposed method with flexible thresholds ("DF (flexible thresh)"). For each $i \neq j$, the subfigure of Figure 3.6 in row i and column j shows the correlation of TIEs measured using the methods with indices i and j. All axes in the subfigures show time, as a multiple of the sample time of the measured signal.

Both of the proposed methods increase the measurement uncertainty of the TIE by 0.39 sample times over the prior method. This increase is less than the inherent timing accuracy of the measurements, which is one sample time. However,

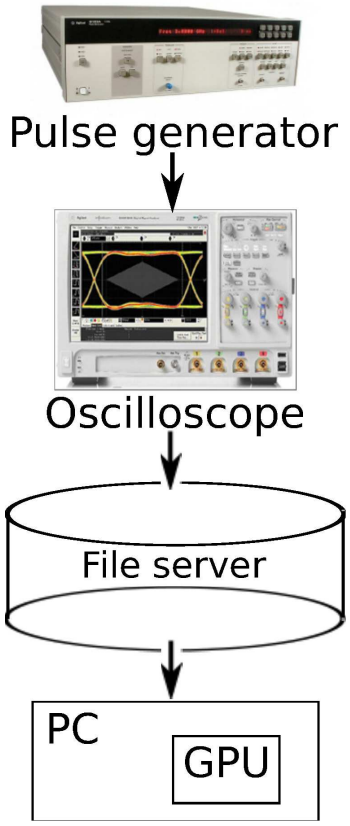

Figure 3.3: Measurement apparatus used to verify the proposed method

there is no discernible difference in the added uncertainty between the two variants of the proposed method.

To quantify further this added uncertainty, the key metric of jitter, the jitter standard deviation (that is, the standard deviation of the TIE), was computed for the three methods (Table 3.2). The absolute relative error between each of the two variants of the proposed method and the prior method are also tabulated there. Both variants of the proposed method increase the measured jitter standard deviation by roughly 0.1 sample or well under one percent. Thus, a small amount of additional measurement error is added in order to remove the restriction that the entire waveform fit into memory.

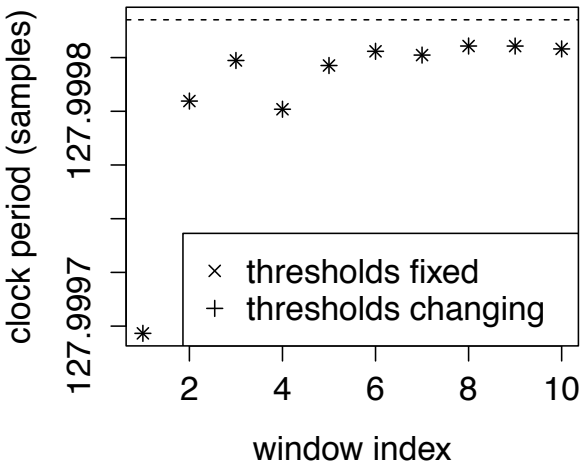

Figure 3.4: Recovered clock period in different windows.

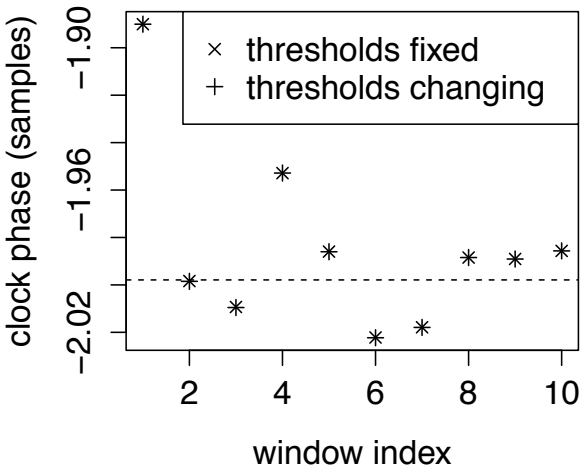

Figure 3.5: Clock phase in different windows.

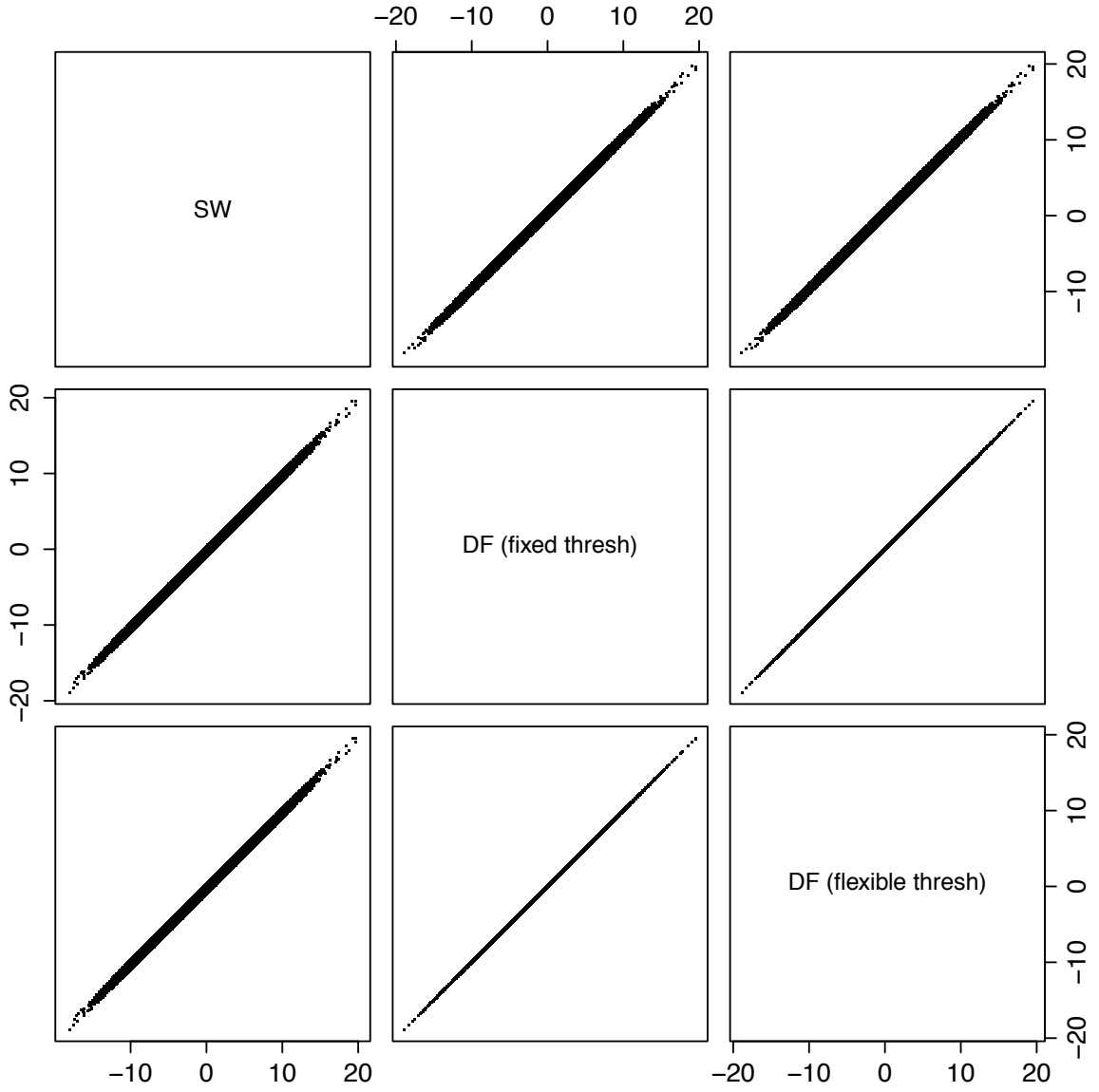

Figure 3.6: Scatter plot matrix for correlation of TIEs using different methods.

Table 3.2: Obtained jitter standard deviation, by method

Method	TIE standard deviation	Absolute relative error
Method of [20]	4.630	—
Proposed, fixed threshold	4.647	0.00378
Proposed, threshold updated	4.646	0.00351

3.4 Summary

In this chapter, we considered the problem of transforming an algorithm for extracting the parameters (period and phase) of a fixed frequency clock from one based on computing from all available measured samples to a dataflow algorithm that can only base its results at any time on a prefix of the samples that have already been processed. One novel contribution of this chapter is the proposed dataflow method for modeling and design of the application. This method provides a formal connection to a wide variety of dataflow-based techniques for deriving efficient implementations on high performance signal processing platforms.

Another novel contribution arises from consideration of the question: how does one tell if such a transformation has adequately been performed? That is, what results of the original and dataflow methods should be compared in order to establish that the dataflow method is approximating well the results of the original method. We propose and demonstrate that two kinds of results should be compared. Firstly, intermediate values computed in the original method on all samples and used in later computational phases should in the dataflow method converge toward the values obtained in the original algorithm. For the proposed dataflow method, these values

are the recovered clock period and phase. In the present example, these do converge. Secondly, the final measured quantities of the original and dataflow methods should differ by a small amount relative to the uncertainty of the measurement results of the original method. Here, the final measured quantity is the jitter standard deviation, which is found by either variant of the dataflow method to with about one third of a percent.

Chapter 4: Deep Waveform Analysis with Parallelization and Constant Memory

In Chapter 3, we presented a novel jitter measurement algorithm that significantly improved measurement response time compared to previous work. The algorithm achieves its efficiency by partitioning the overall data set into windows and allowing jitter measurement results to be reported for earlier windows before later windows are received. This re-formulation of jitter measurement eliminates the swallow and wallow characteristic, and provides improved speed.

However, a memory requirement limitation still remains: the memory required (like the method of [20]) is unbounded. In other words, the memory requirement grows without bound as the size of the data set is increased. This characteristic again limits the amount of signal data that can be measured, which is problematic, for example, in measuring relatively long signals or signals with high sample rates with limited memory resources.

In this chapter, we improve the algorithm of Chapter 3 to overcome its limitation of having unbounded memory requirements. In the jitter measurement approach proposed in this chapter, the memory requirements are fixed for a given system design configuration — in particular, the memory requirements are inde-

pendent of the amount of data that is processed when the system operates. This allows processing of unbounded signal streams: the measurement system can process as much data as it receives during a given execution of the system. At the same time, the method proposed in this chapter provides significantly faster response time compared to previous work, and is capable of delivering measurement results in real time.

For design and implementation of the jitter measurement system presented in this chapter, we integrate the application of Graphics Processing Units (GPUs) [32], the Open Computing Language (OpenCL) [33], and dataflow-based modeling of signal processing systems [3]. GPUs are massively parallel processors that execute large numbers of specialized computational modules, called *kernels*, concurrently to achieve improved performance in terms of throughput and latency. OpenCL is an open standard for programming applications, and executing programs on heterogeneous computing platforms, including platforms that integrate CPU and GPU devices. Dataflow-based modeling provides representations for signal processing application design that help to formally capture high level algorithmic and computational structure in a systematic way. The structure exposed by well-designed dataflow models can help to significantly enhance the reliability and efficiency of derived implementations.

In summary, the novel contributions of this chapter are three-fold. First, we present the design and implementation of a jitter measurement system that jointly provides (a) constant-memory requirements (independent of the amount of data processed) and (b) potential for real-time response. Second, we investigate

fundamental trade-offs among accuracy, processing speed, and memory requirements in the implementation of jitter measurement systems. Third, we demonstrate the integrated application of GPU, OpenCL and dataflow technologies to address design challenges of high speed signal measurement applications.

Material described in this chapter has been published in [6].

4.1 Jitter Measurement System Design

In Chapter 3, we developed dataflow modeling methods and window-based signal analysis methods to improve the efficiency of jitter measurement. In Section 4.1.1 and Section 4.1.2, we provide a brief review of these methods in the context of the objectives in this chapter. We then present the main contributions of this chapter, which enable real-time jitter measurement by (1) significantly improving response time, and (2) providing bounded memory requirements that are dependent on the window length rather than on the overall duration across which the jitter measurement is performed. We discuss novel GPU implementation techniques that we have applied to improve the efficiency of jitter measurement in these dimensions of response time and memory requirements.

4.1.1 Dataflow Modeling

Our jitter measurement system design takes the form of a *computation graph* [28]. Computation graphs are similar to SDF, except that the consumption rate of a port can be different from the number of tokens from the associated input edge that

is accessed during a firing. Tokens that are accessed but not consumed during a firing remain in the associated FIFO so that they can be accessed or consumed in subsequent firings. The number of tokens that is accessed from an edge is referred to as the *threshold* for the associated actor port.

An important step in the implementation of a dataflow graph is the assignment of actors to processing resources, and the ordering of actors that share the same processor. This step is referred to as dataflow graph *scheduling*. The result of the scheduling step is a design component called a *schedule*, which is used to execute the actors in the graph. A wide variety of scheduling techniques have been developed based on specific constraints and objectives in different signal processing application areas (e.g., see [3]).

4.1.2 Window-based Signal Analysis

To help reduce memory requirements for jitter measurement computations on large input data sets, we have developed a windowing method that decomposes the jitter analysis process into fixed-size blocks of successive samples, where the block ("window") size W_s is relatively small compared to the size of the overall data set (see Chapter 3). The dataflow graph can then be executed repeatedly on successive windows of the input data stream. The measurement system designer can set the window size W_s to influence an underlying trade-off between jitter measurement accuracy and memory requirements. Larger window sizes generally provide increased accuracy at the expense of increased memory cost.

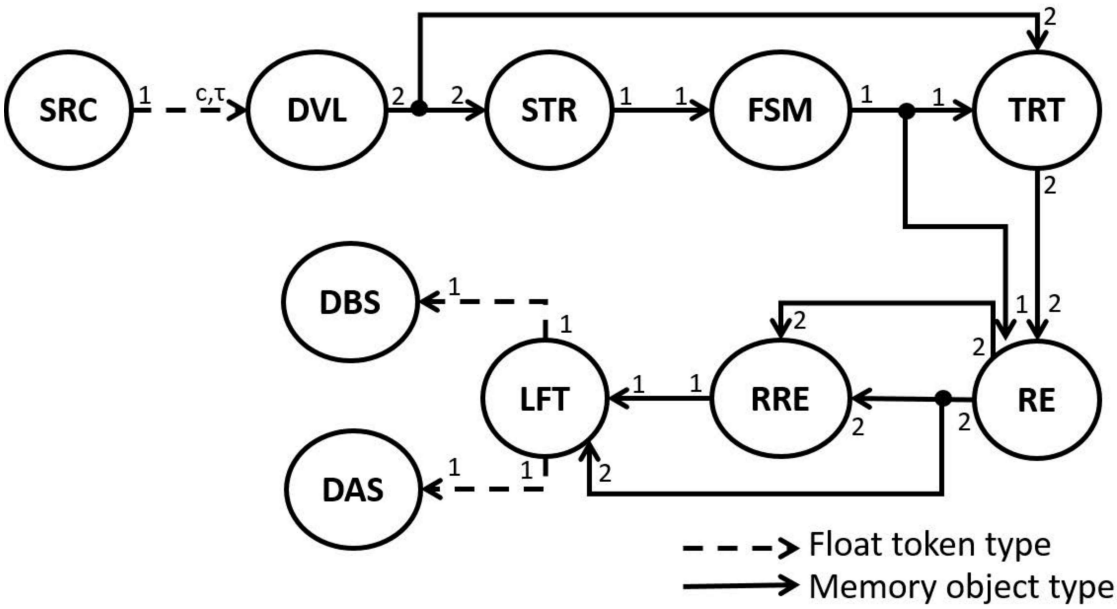

Figure 4.1: Dataflow model for real-time jitter measurement system.

4.1.3 System-Level Model

The dataflow (computation graph) model of our jitter measurement system is illustrated in Figure 4.1. We implement the individual dataflow modeling components (actors and edges) in OpenCL. Details on these implementations are discussed below. The integers next to the actor ports represent the production and consumption rates associated with the ports. For all input ports except one (the input port of the DVL actor), the consumption rate and threshold are equal, so they are not shown separately. The dataflow behavior of the input port of the DVL actor is represented by the parameter pair $[c, \tau]$, where c is the consumption rate and τ is the threshold of the port. The parameter τ is the window size for the actor, and satisfies $\tau \geq c$.

4.1.3.1 Actor Descriptions

Here, we briefly summarize selected actors that are employed in Figure 4.1. The actors summarized here are those whose implementations have changed (compared to the system presented in Chapter 3) due to our use of a GPU for the new measurement system.

Actor DVL (Determine Voltage Level) sorts the input data of the current window and determines the high and low voltage thresholds. Actor STR (State Representation) performs analog-to-digital conversion based on the voltage thresholds to assign low or high voltage states. Actor FSM (Finite State Machine) identifies voltage transitions from high to low or low to high voltage states. Actor TRT (Compute Transition Time) computes the transition time for every voltage transition in the current window. Actor RE (Rough Estimation) derives a preliminary estimate of the clock period. Actor RRE (Refine Rough Estimation) refines the rough estimate of the clock period to improve the accuracy. Actor LFT (Linear Fitting) further refines the clock period estimate using a linear fitting method, and computes time interval errors using the refined clock period estimate. For descriptions of the other actors in Figure 4.1, we refer the reader to Chapter 3.

In addition to changing the implementation platform to a GPU, we have made important improvements in the dataflow graph structure compared to the design in Chapter 3. In particular, phase computation is now implemented as part of the LFT actor instead of as a separate actor. This graph transformation is motivated from observations that (1) GPU kernels for both actors have similar levels of parallelism,

and (2) combination of the actors provides significant reduction in GPU memory requirements. In the transformed graph of Figure 4.1, the outputs of the LFT actor encapsulate the derived clock period and time interval error (TIE) for jitter estimation.

4.1.4 Actor Implementation

In this section, we discuss the implementation of the dataflow graph actors summarized in Section 4.1.3. We emphasize our optimized application of GPU features to jitter measurement tasks, and improvements incorporated in our GPU-based actor implementations compared to the system presented in Chapter 3.

The GPU-targeted actors in our implementation are developed using LIDE-OCL. LIDE-OCL provides an integrated software tool for implementing signal processing dataflow graphs using OpenCL. LIDE-OCL is centered on OpenCL implementations of the abstract (platform- and language-independent) dataflow programming APIs (application programming interfaces) in the lightweight dataflow environment (LIDE) [18].

4.1.4.1 Jitter Measurement Optimization using LIDE-OCL

In LIDE-OCL implementation, computations in an actor can be decomposed into one or more kernels with different amounts of concurrency (GPU "work group" sizes). Initialization of device and kernel configurations is performed before graph execution. Before actors and FIFOs are constructed, relevant host device and GPU

device properties, including device and command queue initialization, are set up. The *command queue* can be viewed as a dynamically-managed list of commands that have been submitted ("issued") to the GPU for execution, and are waiting to be fetched and executed by the GPU. When a GPU-targeted actor is constructed, GPU memory used by the kernels in the actor is allocated, and the kernels are dynamically loaded and compiled so that they are available to the dataflow graph schedule when the graph is executed. Once these initialization and configuration steps are completed, the dataflow graph is ready to execute.

In our implementation of jitter measurement, the deep waveform analysis computations are performed on the GPU. After a window of data is processed, the derived clock period results, and Time Interval Error (TIE) results are sent from GPU memory to the host device (CPU). Since all of the waveform analysis is performed within the GPU, parallelism be exploited extensively throughout the associated computations, and all inter-actor communication is performed within the GPU (rather than between the GPU and the host). These features help significantly to improve jitter measurement response time. The token type used by the GPU-targeted actors is a generic type associated with OpenCL objects [33]. This organization provides flexibility and efficiency in processing different kinds of data within actors on the GPU.

In the remainder of this section on actor implementation, we provide details on how efficient parallel execution of jitter measurement computations is achieved on the targeted GPU platform.

4.1.4.2 DVL and RE

Efficient sorting of numeric values is important in our jitter measurement system. For example, in the DVL actor shown in Figure 4.1, a sorting algorithm is applied to determine thresholds for high and low voltage values. Similarly, the RE actor operates by sorting the differences between neighboring transition times.

We apply a sorting algorithm called *bitonic sort* to accelerate the sorting operations required for jitter measurement calculation. Bitonic sorting was applied originally in the construction of sorting networks [34], which can be viewed as interconnections of comparators and wires that are used to sort collections of values. It is well known that bitonic sort is useful for its utility as a parallel sorting algorithm. We apply this feature to derive fast implementations of the DVL and RE actors on the targeted GPU.

In the design of the RE actor, we compute the intervals of pairs of neighboring transitions, and then sort these intervals in ascending order using *bitonic sort*. Then the 25^{th} percentile of the sorted intervals is computed as the rough estimation of the clock period.

4.1.4.3 RRE and LFT

The RRE and LFT actors involve computing sums over large numbers of data values. The associativity of the addition operator allows for use of efficient GPU implementation techniques that are based on parallel computation methods for *reduction operations* [35]. However, due to the large volumes of data involved,

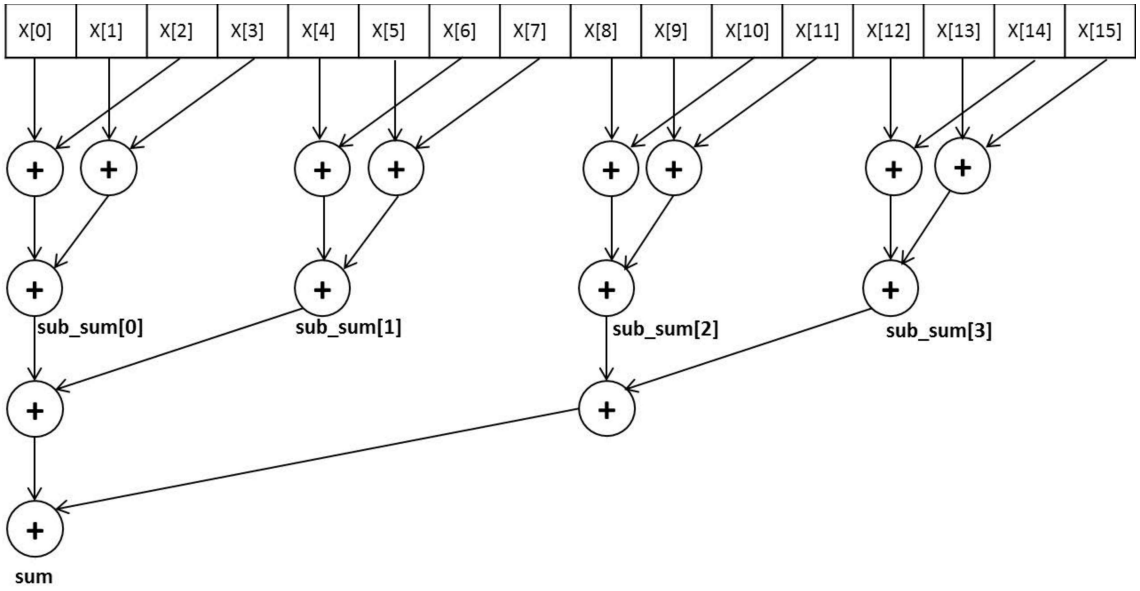

Figure 4.2: Illustration of reduction methods applied to summation

reduction techniques must be applied carefully in our implementation to optimize performance. For example, non-local synchronization of data between every component operation within a reduction operation is costly. In OpenCL-based GPU implementation, relatively costly non-local synchronization arises when the data involved in an operation crosses the boundary of a set of operations called a *local work group*. Thus, we structure the summations in the RRE and LFT actors such that they are reduced first at the level of local work groups. This makes maximal use of local memory operations (fast synchronization) during the reduction process.

Figure 4.2 illustrates our application of reduction methods to summation operations. In the example of Figure 4.2, the total data length (number of values to be added) is 16, and the local work group size is 4.

4.1.4.4 TRT

Another computationally-intensive process in our jitter measurement system is the *stream compaction* [36] of the transition time array in the TRT actor. Stream compaction refers to the process of compressing the storage requirements of a sparse array by removing zero-valued elements from the array. In each jitter measurement window, transitions are detected at switching points between high and low voltage levels, the corresponding transition times are computed, and these transition times are stored by setting array elements to non-zero values at array indices that correspond to the transition time intervals. This approach is efficient for initial computation and storage of the transition times, but it results in a sparse array that is costly in terms of memory.

Thus, within our implementation of the TRT actor, we compress the sparse transition time array using a prefix sum [37] technique. This stream compaction process is illustrated in Figure 4.3 with a simple example involving 10 data items.

In OpenCL, a prefix sum computation on a large data set can be implemented in a manner similar to a reduction operation. Using such an approach, we compute the prefix sum for the local work groups in parallel. Then we compute the prefix sum of those partial prefix sums. The result of this intermediate prefix sum operation is called the "summation offset". The final prefix sum result is then computed by adding the summation offset to the results computed on the local work groups.

Figure 4.3: An illustration of stream compaction.

4.1.5 Schedule for Dataflow Graph Execution

As is discussed in Section 4.1.1, a schedule is needed to execute the dataflow graph on a targeted hardware platform. Many possible schedules can be constructed for the dataflow graph of Figure 4.1. The alternative schedules in general have different implementation costs in terms of relevant metrics. For the experiments in Section 4.2, we derive a *static schedule* that employs a heuristic to decrease the CPU and GPU memory requirements. A static schedule is one in which the assignment and ordering of all actors are fixed before the graph is executed. Static schedules are useful for reducing the run-time overhead of schedule execution, and for improving the predictability of the implementation [3]. Constant-valued dataflow rates and thresholds allow for the construction of static schedules, and we exploit this property of our dataflow graph model when constructing the schedule.

The static schedule that we employ in our implementation can be expressed

as

$$(W_s\ SRC)\ DVL\ STR\ FSM\ TRT\ RE\ RRE$$
$$LFT\ DBS\ DAS,$$

where the parenthesized term $(W_s\ SRC)$ represents the successive execution W_s times of the SRC actor. Recall from Section 4.1.3 that W_s represents the window size for jitter measurement.

4.2 Experimental Verification

In this section, we present an experimental validation of the proposed method. Actual measured data were used: a two-state digital waveform representing bits from a PRBS sequence measured using a deep-memory digital oscilloscope. A fuller description of the waveform and the measurement apparatus used to acquire it was provided in Chapter 3. Also in that reference, there were proposed two criteria for judging the correctness of a measurement algorithm that is transformed in order to meet software engineering goals such as increased throughput, decreased latency, or decreased memory consumption:

- Intermediate values obtained using both the old and new algorithms should be comparable, and
- The difference between the final measurement results of the old and new algorithms should be small compared to the total measurement uncertainty in

either.

Here, we compare the measurement results of [20] and Chapter 3 with those of two variants of the proposed method, which we will refer to as the accumulated and real-time variants. The same actual measured data is used for the comparison. Our experiments are carried out on a hybrid CPU-GPU platform that includes an Intel Core i7-2600K Quad-core CPU with an NVIDIA GeForce GTX680 GPU running Ubuntu Linux 12.04 LTS. OpenCL 1.0 and GCC 4.6.3 are used for code compilation.

In the accumulated variant, the transition times from the very first window are accumulated in a buffer and the clock period estimation is based on all transitions found from the first window to the current window. Voltage thresholds are fixed based on the voltage statistics found in the first window. Thus, the first window needs to contain at least one logic state transition. The accumulated variant needs only to store one array that grows with waveform depth: the transition times. Thus, it has lower memory consumption than the methods of [20] and Chapter 3, but nevertheless can run only for some finite time before computer memory is exhausted.

In the real-time variant, clock recovery in every window is regarded as a complete and independent process. The voltage thresholds, transition locations, clock recovery, and time interval errors within each window are computed independently of all other windows. This variant has memory requirements that are fixed, independently of the amount of time it runs: the amount of time it can run is not memory-limited. By "real-time" we refer to the ability of this method to operate successfully on a temporally unbounded waveform. The term is not to be under-

stood as a claim to be able to measure at the full, many-gigasample per second, sampling rate of contemporary instruments.

Both variants were implemented in LIDE-C and in LIDE-OCL. The two implementations produced identical results to 8 decimal places, and so are identical for all practical purposes. This validates the correctness of the LIDE-OCL implementation. The LIDE-OCL results are used in the following graphs.

The accumulated variant reports the desired statistic about jitter, the standard deviation of time interval error (TIE), only after processing the entire waveform. Table 4.1 shows the value of that statistic obtained using the accumulated variant for various window sizes and reported in [20] and Chapter 3 for the same waveform. All of the results are within 0.02 of a sample time, under a half a percent of relative error.

Validation of the real-time variant is more complicated because it produces a value of standard deviation of TIE for each window. And, as shall be seen below, the statistical behavior of the measurement results varies considerably with the window size. The same measured waveform was provided to the real-time variant, once for each of a number of window sizes.

Figure 4.4 summarizes the statistical distribution of the key intermediate computed value, the waveform unit interval (UI). In this figure, box plots show the distribution of corrected UIs as a function of window size, as determined by the LIDE-OCL implementation of the real-time variant, and given the actual measured data described in Chapter 3. The known *a priori* correct value is 128.00. Boxes extend from the 25th to the 75th percentile. The median corrected UI is shown as

a thick black horizontal line. Outliers are shown as circles. One extreme outlier was deleted for window size 8192.

Figure 4.5 shows the standard deviation of TIE found for each combination (w_i, w_s), where w_i is a window index and w_s is a window size.

More specifically, Figure 4.5 demonstrates the evolution of the standard deviation of the TIE as windows are processed in the real-time variant. Each point in the figure corresponds to the standard deviation of the TIE in a single window. Different points therefore correspond to different combinations of window indices and window sizes. Different colors indicate results that are based on different window sizes. The x-axis indicates the last sample index in each window. The horizontal black line is at the standard deviation value of 4.63 samples, which is obtained from experiments in [20].

Figure 4.6 summarizes with a box plot the statistical distribution of the standard deviations of the previous figure. The horizontal line passing through the figure shows the standard deviation of 4.63 sample times obtained by experiments [20]. Figure 4.6 shows that the UI estimates and standard deviation of TIEs of the real-time variant converge toward those of the accumulated variants and results of [20] and Chapter 3. However, there is a significant frequency of high measurement errors (shown with the outlier circles in Figures 4.4 and 4.6) at lower window sizes. On the other hand, the measurement results are visually as accurate with windows of 131,072 samples as they are when processing the entire waveform of over 14 million samples — yet the windowed version requires approximately 1% of the memory.

Table 4.1: Standard deviation of TIE: Accumulated variant

window size (samples)	[20]	Chap. 3	8192	16384	32768	65536	131072
std dev of TIE (sample times)	4.63	4.65	4.65	4.65	4.65	4.65	4.65

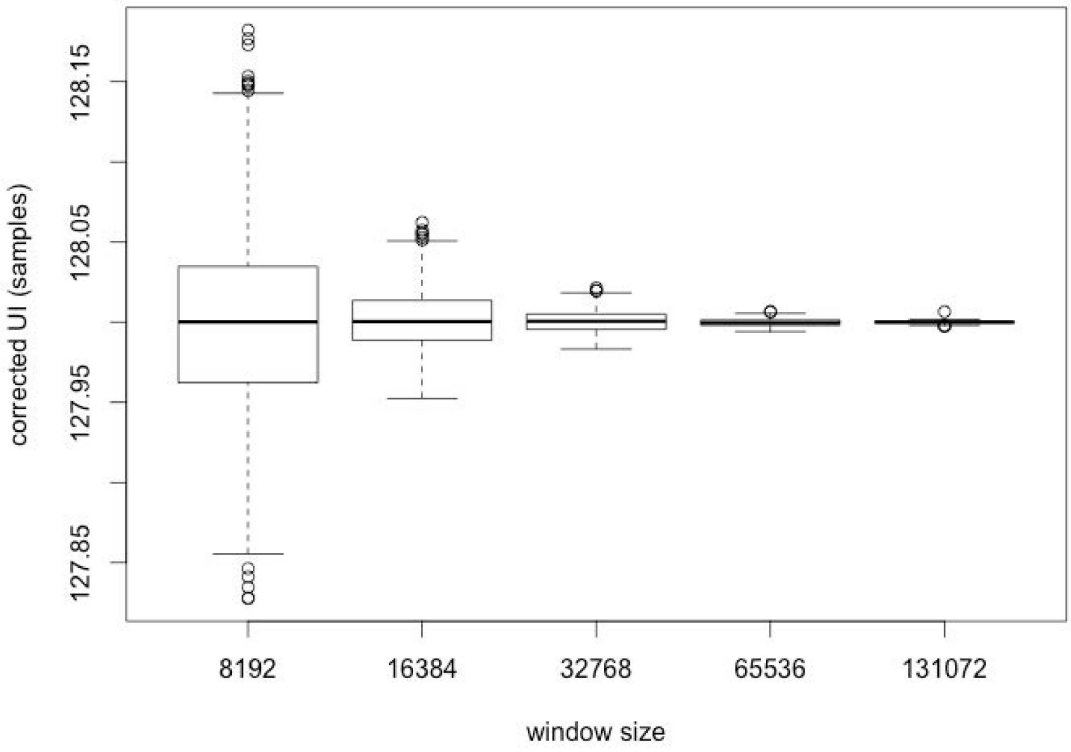

Figure 4.4: Box plots of corrected waveform unit intervals (UIs).

4.3 Summary

In this chapter, we have presented a system for deep jitter measurement that achieves real-time operation, and memory requirements that are constant (independent of the amount of data that is processed). Such constant memory requirements

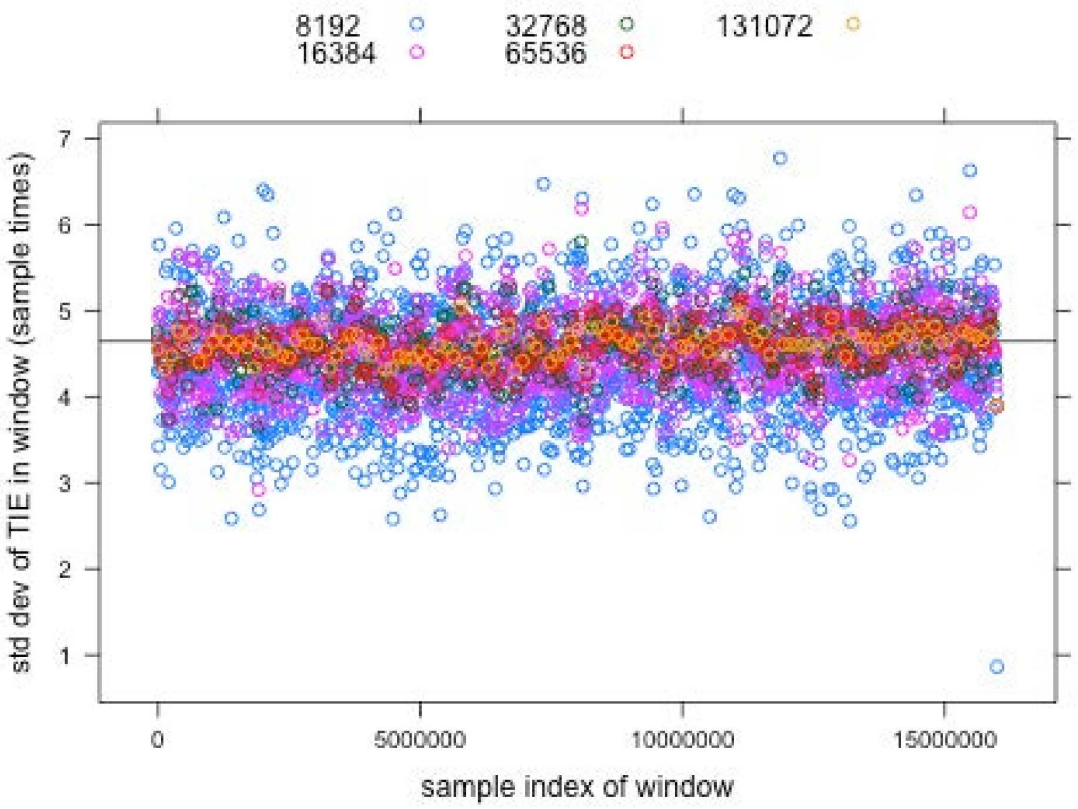

Figure 4.5: Standard deviation of TIE for different window sizes.

enable the novel capability of processing unbounded-length signal streams in a jitter measurement system, which in turn provides more thorough and accurate jitter assessment. We have also exposed and investigated system-level design trade-offs among computation accuracy, memory requirements and latency in jitter measurement. Finally, we have validated our new jitter measurement system's consistency with related previous systems by verifying that it produces both intermediate computed values and final measurement results that converge to within sub-percent measurement errors compared to two previous systems. Useful directions for future work include optimization of the dataflow graph actors and the schedule to further improve the execution speed of the implementation.

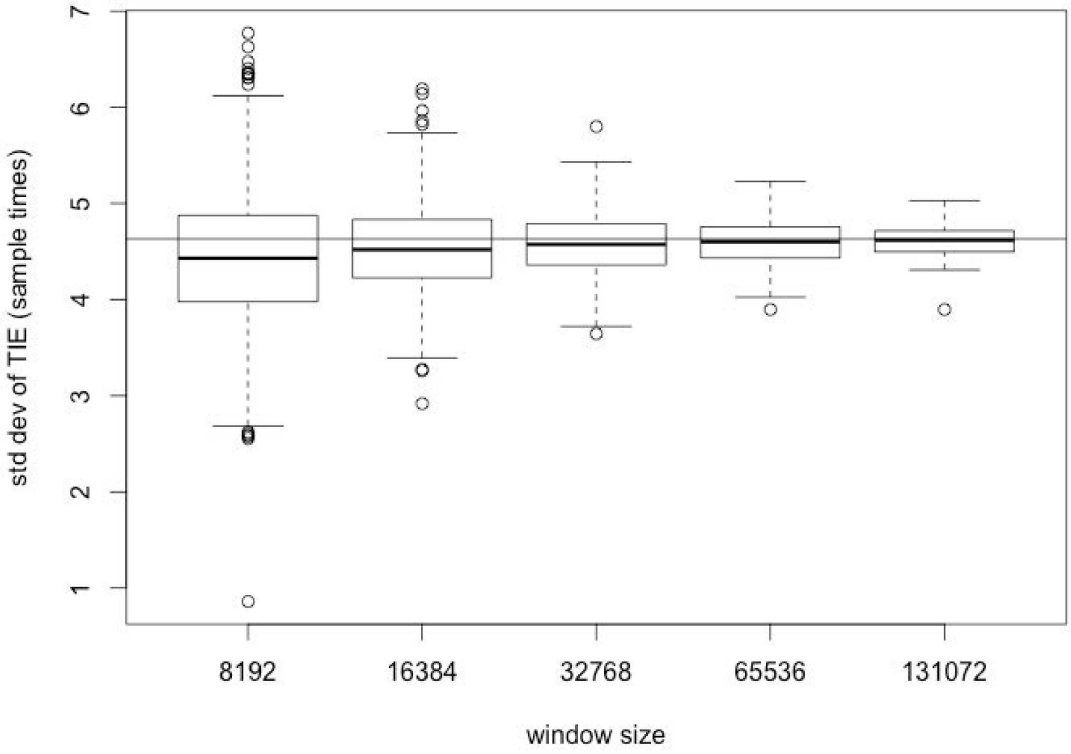

Figure 4.6: Box plots of TIE standard deviation for different window sizes.

Chapter 5: Design Methods for Gapless DSP Applications

In Chapter 4, we presented a novel deep jitter measurement system that loads and processes constant-frequency signal data from an input file. The contribution of Chapter 4 was focused on streamlining memory requirements and efficiently trading off accuracy and performance. The contribution improved the algorithm of Chapter 3 to overcome its limitation of having unbounded memory requirements. This led to a novel deep jitter measurement system whose memory requirements are fixed for a given system design configuration — in particular, the memory requirements are independent of the amount of data that is processed when the system operates. This allows processing of unbounded signal streams: the measurement system can process as much data as it receives during a given execution of the system.

In this chapter, we go beyond the developments of Chapter 4 in the following ways. First, we incorporate methods to process input from a data acquisition (DAQ) device under the constraint that samples received from the device must be reliably stored and processed. We refer to this form of reliable operation as *gapless* operation. Second, we present design optimization techniques that significantly improve memory management efficiency and system throughput. Additionally, we incorporate methods to dynamically monitor the frequency of the input signal, and

adapt relevant system parameters when changes in the input frequency are detected.

5.1 Introduction

This chapter is concerned with the design and implementation of an important class of digital signal processing (DSP) applications that we refer to as *gapless* DSP applications. A gapless DSP application is characterized by one or more continuous streams of input data, where the data must be processed without gaps — that is, without dropping any of the input samples. The strict real-time processing requirements for gapless DSP applications can be very challenging when input data rates are high, processing requirements are intensive, or the target platform is significantly resource constrained. The major objective of this chapter is to provide structured models and systematic methods for addressing this challenge. For concreteness, the models and methods are developed in the context of a specific gapless DSP application, which is an application involving jitter measurement of deep waveforms. However, the core approaches developed in our chapter are not specific to this application, and can be adapted to other relevant applications.

Real-time jitter measurement of deep waveforms is an example of a gapless DSP application that has important applications in instrumentation for digital communication systems. *Deep waveforms* are signals with long durations and high sample rates that result in large numbers of samples that need to be processed. For conciseness, we refer to jitter measurement in this context as *deep jitter measurement*.

In this chapter, we develop techniques for optimized mapping of deep jitter measurement onto a high-performance, heterogeneous computing platform. The techniques are designed to address the challenges associated with gapless operation, real-time processing, and deep waveform analysis in a systematic, model-based manner. Here, by *model-based*, we mean that the methods are developed in terms of formal models of computation, and at a level of abstraction that is higher than that of conventional platform-oriented design languages, such as C, C++, CUDA, and OpenCL. For detailed background on model-based design for signal processing systems, we refer the reader to [38].

Specifically, we develop model-based techniques based on *dataflow* models of computation, which are widely-used in signal processing design and implementation. In this form of dataflow, signal processing applications are represented as directed graphs in which vertices (*actors*) represent DSP hardware/software components and edges represent first-in, first-out (FIFO) buffers that store data as it passed from the output of one actor to the input of another. Formal underpinnings of this form of model-based design are presented in [4].

An important aspect of the techniques that we develop in this chapter is the model-based integration of data acquisition (DAQ) devices into dataflow-based design processes. DAQ boards are widely applied in gapless DSP applications to enable continuous data collection from input sources. DAQ boards are widely used in numerous signal processing application areas, such as astronomy, environmental monitoring, biomedical instrumentation, and satellite communication (e.g., see [39, 40]).

In the deep jitter measurement system that we develop in this chapter, we employ as the target platform a hybrid CPU-GPU computing platform that is connected to a DAQ board. This provides a state-of-the-art platform for high-speed, heterogeneous signal processing of continuously arriving digital communications waveforms. The methods developed in this chapter focus on optimizing the throughput of jitter measurement subject to the on-board memory constraints of a given DAQ interface, GPU memory constraints, and the constraint of gapless processing.

More broadly, the techniques developed in this chapter provide a novel framework for addressing in an integrated manner the following important challenges of gapless DSP system design: (1) the requirement for processing unbounded data streams without DAQ buffer overflow; (2) the need for efficient methods to trade-off signal processing accuracy and throughput subject to the constraint of gapless processing; and (3) iterative platform-based optimization of dataflow actor implementations to maximize system throughput.

The remainder of this chapter is organized as follows. Section 5.2 presents some background beyond the concepts discussed in Chapter 2 that is relevant to this chapter. Section 5.3 presents dataflow graph design approaches for efficient implementation of gapless DSP applications, using deep jitter measurement as a concrete case study. In Section 5.4, we present design optimization methods to improve the real-time performance of the deep jitter measurement system. Experiments and analysis of the optimized design are presented in Section 5.5.

5.2 Background

The developments of this chapter depend on some background on dataflow-based design beyond the background that has been reviewed in Chapter 2. In this section, we review this additional background.

An important task in the implementation of a dataflow graph is the task of constructing a *schedule* for the graph. A schedule specifies the assignment of actors to processing resources, and the execution order of actors that are assigned to the same resource. If all of these assignment and ordering decisions are made at compile time, the schedule is said to be *static*, whereas if some of the decisions are deferred to execution time, it is said to be a *dynamic* schedule [11]. If the decisions are made after compile time but prior to graph execution, the schedule is said to be a *just-in-time* schedule [41]. Static and just-in-time scheduling techniques offer increased predictability and reduced run-time scheduling overhead at the expense of generality — they cannot be applied to all types of dataflow models.

In this chapter, we focus primarily on static scheduling techniques. In the dataflow graph execution model that we apply, a statically constructed schedule is executed iteratively, where each iteration is triggered by the availability of a new block of input samples from a DAQ device. The dataflow graphs that we apply in this chapter are sufficiently predictable to enable this form of static scheduling. Extension of the static scheduling techniques proposed in this chapter to just-in-time deployment contexts is an interesting direction for future work.

5.3 System Design

In this section, we discuss our methods for dataflow graph design of gapless deep waveform analysis applications. As described in Section 5.1, we present these methods in the context of a concrete application — deep jitter measurement. The deep jitter measurement system that we develop is a gapless DSP system where a DAQ subsystem supplies continuously arriving input samples, and these samples are processed to analyze the jitter of input waveform.

The primary challenges when integrating jitter measurement algorithms with DAQ devices for real-time analysis include adhering to memory capacity constraints, ensuring that system throughput does not fall below the sampling rate of the DAQ device, and avoiding excessive latency in the jitter measurement computation. The methods developed in this section provide our system design foundations for addressing these challenges. The core dataflow-based system architecture presented in this section is built upon in Section 5.4 with various optimization techniques. These optimizations further improve the trade-offs among memory cost, throughput, and latency that are achieved by our deep jitter measurement system design.

5.3.1 Window-based Analysis

The dataflow graph for our deep jitter measurement system is designed to measure jitter continuously so that intermediate results of jitter analysis and the recovered clock period are accessible, and so that computational latency is streamlined while meeting throughput constraints.

A windowing method is applied to reduce the memory requirements of the jitter measurement system. The windowing method decomposes the input stream into a set of fixed-size subsequences. The fixed size is referred to as the *window size* W_s. In our implementation, the dataflow graph memory requirements are dependent only on W_s and not on the number of windows that is processed. Thus, the jitter measurement dataflow graph can be executed on an unbounded number of windows with predictable, bounded memory requirements. The window size is a system parameter that can be configured by the designer to control an associated trade-off between measurement accuracy and memory requirements for deep jitter measurement. Larger values of W_s in general lead to improved accuracy at the expense of higher memory requirements. We discuss this trade-off further in Section 5.4.1.

5.3.2 DAQ Interfacing

In design and implementation of gapless DSP systems, we are concerned with processing data that arrives continuously from one or more DAQ subsystems. The data processed by the system dataflow graph is accessed from one or more internal buffers on the DAQ devices rather than from files that are stored on disk.

In our deep jitter measurement system, we employ a single DAQ device. To integrate use of the device into the system-level dataflow graph, we develop a source actor that encapsulates the functionality associated with acquiring data from the DAQ device. Here, by a *source actor*, we mean a dataflow actor that has no inputs; such actors are commonly used to model interfaces between dataflow graphs and

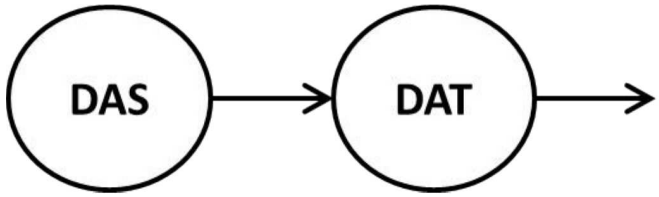

Figure 5.1: Subgraph for acquiring data.

sources of input data. Similarly, *sink actors*, which have no outputs, are used to model output interfaces of dataflow graphs.

We use the dataflow subgraph shown in Figure 5.1 to model the process of acquiring data from the DAQ subsystem and converting the data to a stream of digital input samples that is to be processed by the rest of the enclosing dataflow graph. The subgraph consists of two actors: the *DAS* (*Data Acquisition Source*) actor handles configuration of the DAQ subsystem as well as acquisition of raw data, while the *DAT* (*Data Acquisition Transformation*) actor performs any preprocessing required on the raw data (to extract individual samples), as well as the sending of the preprocessed data to a GPU device for the core signal processing tasks in the given gapless DSP application.

In the remainder of this section (Section 5.3.2), we demonstrate concrete implementations for the DAS and DAT actors that target the specific type of DAQ device that we have used in our experiments. The targeted DAQ device is the Keysight U5303A PCIe High-Speed Digitizer. For conciseness, we refer to this specific DAQ device in the remainder of this chapter as the *Targeted DAQ Device* (*TDD*). The implementations of the DAS and DAT actors are developed using LIDE-OCL (see Chapter 2).

5.3.2.1 DAS Actor Implementation

The design of the DAS actor is decomposed into three CFDF modes, called the *initialization*, *inject*, and *error* modes. Before acquiring data from the TDD, a DAQ configuration, including selection of the sample rate, needs to be set up. The triggering process for the device also needs to be set up. The initialization mode handles these setup tasks, and then transitions the actor to the inject mode, which can be viewed as representing the steady state functionality of the actor.

Upon each firing in the inject mode, a new frame of data is fetched from the internal buffer of the TDD and made accessible to the rest of the dataflow graph for processing. A new frame corresponds to a new window based on the window-based analysis described in Section 5.3.1. The actor is enabled (allowed to fire) only when there is a new frame of data available within the TDD internal buffer, and there is sufficient empty space on the actor's output edge e_{out} for transfer of the new frame to the DAT actor. If we model the internal buffer as a self-loop edge connected to the DAS actor, then the enable method involves checking for sufficient data on this self-loop edge. In a dataflow graph, a *self-loop edge* is an edge whose source and sink vertices are identical. Self-loop edges are an established method for modeling actor state in signal processing dataflow graphs (e.g., see [42]).

Instead of copying raw data from the internal buffer to e_{out}, only a pointer value p_{out} is written to e_{out}. This value contains the starting address of the block of memory in the internal buffer where the next frame of acquired data is stored. The DAT actor can then use this pointer value to access the acquired data directly from

the TDD internal buffer so that the data does not need to be copied.

Once the actor is in the inject mode, it remains in this mode indefinitely until the system is stopped or reset through external control, or until an error, such as overflow of the TDD internal buffer, is detected. Upon detection of an error, the actor transitions to the error mode, and remains in that mode until the system is reset. As one might expect, further data acquisition from the TDD is disabled while the DAS actor is in the error mode.

5.3.2.2 DAT Actor Design

The TDD packages pairs of adjacent input samples as two 16-bit data items within a single 32-bit *packed pair* of samples. In our hybrid CPU-GPU implementation, the TDD actor sends packed pairs to a GPU to be unpacked and then injected into the dataflow subgraph that carries out the core signal processing functionality for deep jitter measurement. The overall dataflow graph for deep jitter measurement, including the subgraph of Figure 5.1 and the subgraph for core signal processing, is presented in Section 5.3.3. Within the GPU, the accesses of the packed pairs and the operation of all of the core signal processing actors are parallelized to optimize real-time performance.

5.3.3 Dataflow Graph for Deep Jitter Measurement

Figure 5.2 illustrates the overall dataflow graph for our deep jitter measurement system. Here, as described in Section 5.3.2, the DAS and DAT actors provide

the input interface for the deep jitter measurement system. The output interface is provided by the SKC and SKT actors, which store measurement results in output files. Descriptions of these actors along with all of the other actors in Figure 5.2 are summarized in Table 5.1. For further background on computations involved in jitter measurement, we refer the reader to [5, 6, 20].

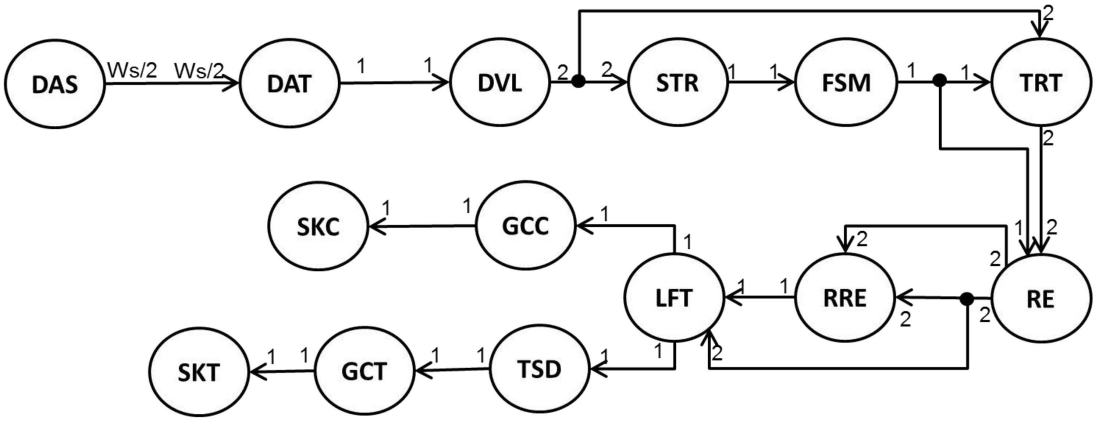

Figure 5.2: Dataflow graph for deep jitter measurement system.

In the context of a gapless DSP application, we say that a CFDF actor is a *single-mode steady state* (*SMSS*) actor if it contains a unique mode, called the *signal processing mode*, that is intended to be executed during the continuous data processing ("steady state") phase of the enclosing application. If an SMSS actor has one or more modes in addition to its signal processing mode, then those modes must be executed during system initialization or during error handling (e.g., as illustrated in Section 5.3.2.1 for the DAS actor). Since CFDF modes must have constant production and consumption rates on all actor ports (see Chapter 2), the steady state behavior of an SMSS actor can be represented by an SDF actor that corresponds to only the signal processing mode.

In the dataflow graph of Figure 5.2, all of the actors are SMSS actors. The edges in the figure are annotated with the production and consumption rates associated with the signal processing modes of the actors. For example, the signal processing mode of the RRE actor consumes two tokens on each of its input edges and produces one token on its output edge on each firing. Recall from Section 5.3.1 that W_s, which appears in the annotations associated with edge $(\mathrm{DAS}, \mathrm{DAT})$, represents the window size.

The token types associated with the edges in Figure 5.2 are summarized as follows. The edge $(\mathrm{DAS}, \mathrm{DAT})$ has `long` type. The edges $(\mathrm{GCC}, \mathrm{SKC})$ and $(\mathrm{GCT}, \mathrm{SKT})$ have `double` type. All of the other edges in Figure 5.2 have *OpenCL memory object* token type. A memory object in OpenCL is a pointer that points to a linear arrangement of bytes that resides on the GPU and can be accessed by the host.

After each complete firing of the DAS actor, the following static subschedule of the remaining 13 actors in the graph is executed to process the next frame of data acquired from the TDD.

$$\begin{array}{l} DAT\ DVL\ STR\ FSM\ TRT \\ RE\ RRE\ LFT\ TSD\ GCC\ GCT \\ SKC\ SKT \end{array} \tag{5.1}$$

Acquisition of a new frame of data by the TDD can then proceed concurrently

Table 5.1: Actors in the dataflow graph of Figure 5.2.

Actor	Description
DAS	Data acquisition source. Interface for acquiring data from the TDD.
DAT	Data acquisition transformation. Sends packed pairs of samples to the GPU, and unpacks the samples on the GPU.
DVL	Determine voltage level. Sorts the input data in the current window and determines high and low voltage thresholds.
STR	State representation. Converts samples that encapsulate voltage values into digital form (high/low voltage states).
FSM	Finite state machine. Determines voltage transitions from high to low voltage states or low to high voltage states.
TRT	Compute transition time. Computes the transition time for each voltage transition in the current window.
RE	Rough estimation. Derives a preliminary estimation of the clock period.
RRE	Refine rough estimation. Refines the rough estimation of the clock period to improve its accuracy.
LFT	Linear fitting. Further refines the estimated clock period with linear fitting. Computes time interval errors (TIEs) using the refined clock period estimate.
TSD	Compute TIE standard deviation. Computes the standard deviation of the TIEs for the current window.
GCC	GPU to CPU data transfer. Transfers clock period result from GPU memory to CPU memory.
GCT	GPU to CPU data transfer. Transfers TIE standard deviation from GPU memory to CPU memory.
SKC	Corrected refined estimation sink actor. Produces the result of the corrected refined estimation for the recovered clock period.
SKT	Standard deviation of TIE sink actor. Produces the standard deviation of the TIEs.

with execution of the subschedule in Equation 5.1. The subschedule of Equation 5.1 involves no run-time scheduling overhead since the ordering is constructed as a topological sort, which respects all of the data dependencies among the actors. The run-time testing of data availability in the system is limited to just the DAS actor, which is polled for availability of a new data frame whenever an iteration of the static subschedule (Equation 5.1) completes and there is no input data on the (DAS, DAT) edge that is available to trigger the next subschedule iteration.

5.4 Performance Optimization

Gapless DSP applications generally require high throughput to process input streams without missing data points and while reliably avoiding memory overflow. In this section, we demonstrate algorithm- and implementation-based optimization methods to help address these multi-faceted implementation constraints. Taking the dataflow graph presented in Section 5.3 as a starting point, we improve the design by applying a sequence of optimizations. These optimization techniques are described in Section 5.4.1 through Section 5.5.5. Experimental results from applying these optimization are then presented in Section 5.5.

5.4.1 Window Size Optimization

In this section, we discuss optimized, dynamic configuration of the window size parameter W_s, which was introduced in Section 5.3.1. In our deep jitter measurement system, the window size, along with sorting-related parameters (discussed

in Section 5.4.2) that are directly influenced by W_s, have significant impact on trade-offs among measurement accuracy, execution time performance, and memory requirements.

In jitter measurement systems, the frequencies of the input signals are typically not known at design time, and vary dynamically at run-time. A larger window size in general improves the accuracy of signal frequency and TIE estimation. For lower frequencies (larger clock periods), a larger window size is preferred to encapsulate a sufficient number of signal periods per signal frame. Larger window sizes also provide improved accuracy, as demonstrated in Chapter 4. Larger window sizes also improve throughput.

However, memory requirements increase linearly with the window size. Thus, we initialize execution of our jitter measurement system to support an initial minimal frequency of f_{init}, and we increase the window size dynamically if we encounter signals that have lower estimated frequency levels than the currently supported minimum frequency.

More specifically, In our deep jitter measurement system, the window size is dynamically optimized by monitoring the number of high/low signal transitions found in each window. If the number of transitions falls below a threshold C_{trt_num}, then the window size for subsequent signal frames is doubled.

In our experiments, we use $f_{\text{init}} = 130$ kHz, and we use the empirically-determined value of $C_{trt\_num} = 32$ transitions per frame. The value of C_{trt_num} can be varied to tune system-level trade-offs — lower threshold values lead to lower memory requirements and faster execution time at the expense of decreased accuracy

of gapless signal analysis.

5.4.2 Sorting Optimization

Sorting operations are involved in two actors of our jitter measurement system, the DVL and RE actors. These operations account for significant portions of the overall computation in a given dataflow graph iteration. We employ bitonic sort [34] in an effort to enhance the efficiency of the sorting process.

To further improve the efficiency of sorting, we sort only part of the relevant data associated with each signal frame, and perform the required analysis on the partially-sorted data. This again represents a way to trade-off reduced accuracy for improved real-time performance. We configure the optimized sorting process carefully to ensure that the reduction in accuracy stays within a reasonable level.

In the DVL actor, the input data in a given signal frame is sorted to select high and low voltage thresholds. These thresholds are then used to find the high-to-low and low-to-high signal transitions in the given frame. We randomly select a subset of the data samples in each data frame to sort. The size S_{DVL} of this subset is determined as

$$S_{\text{DVL}} = \text{power}(k_{\text{DVL}} \times \text{ceil}(W_s/N_{\text{trans}})), \tag{5.2}$$

where k_{DVL} is a positive integer parameter, ceil(x) gives the smallest integer that is greater than or equal to the real-valued argument x, power(y) gives the smallest power of two that is greater than or equal to the integer argument y, and N_{trans} is

the number of signal transitions that were detected in the previous frame. In other words, (S_{DVL}/W_s) gives the fraction of available samples that are used in the sorting process.

For example, suppose that $k_{\text{DVL}} = 4$, and $W_s = 65,536$, and $N_{\text{trans}} = 135$, then:

$$S_{\text{DVL}} = \text{power}(4 \times \text{ceil}(65536/135)) = \text{power}(4 \times 486) = 2^{11} = 2048. \quad (5.3)$$

In each firing of the RE actor, a sorting operation is performed as part of the process for deriving a rough clock period estimate. In each signal frame, the differences in pairs of neighboring transition times are sorted, and the 25^{th} percentile of the sorted transition time differences is taken as the rough estimate.

Here, we use a threshold C_{RE} to determine the size S_{RE} of the subset (of all transition time differences) that is sorted. If $N_{\text{trans}} > C_{\text{RE}}$, then S_{RE} is set to C_{RE} for the current frame; otherwise, S_{RE} is set to N_{trans}.

In our experiments, we use $k_{\text{DVL}} = 4$, and $C_{\text{RE}} = 1,024$. Through experimentation, we have determined these values to provide improvements in sorting efficiency without significantly degrading jitter measurement accuracy.

5.4.3 Throughput Optimization

In this section, we focus on further methods that we have applied to optimize the throughput of computationally-intensive actors in the proposed deep jitter

measurement system. As discussed previously, we targeted our implementation to a hybrid CPU-GPU platform with C and OpenCL as the actor implementation languages for CPU- and GPU-based mapping, respectively.

All of the computationally-intensive actors in our jitter measurement system employ GPU acceleration. Specifically, the following actors employ GPU kernels: DAT, DVL, STR, FSM, TRT, RE, RRE, LFT, and TSD. However, some GPU-mapped operations, are not fully parallelized (see Chapter 4). In particular, sorting, prefix sum, and reduction operations significantly limit the performance of several actors. Both the DVL and RE actors involve sorting; the TRT actor includes prefix sum computation; and the RRE and LFT actor include reduction operations.

For the RE and DVL actors, we described in Section 5.4.2 how we employed approximate computing techniques that trade-off acceptable decrease in accuracy for improvement in execution time. In addition to these techniques, we employ dynamic configuration of the vectorization degree to further improve processing efficiency.

By the vectorization degree of a kernel, we mean the number of data parallel instances of a kernel that are launched simultaneously. In OpenCL terminology, the vectorization degree is commonly referred to as the number of global work items. Careful optimization of vectorization degrees can have major performance benefit for GPU acceleration of dataflow graphs [43].

For the sorting operation within the RE actor, an efficient value for the vectorization degree is S_{RE}. However, as discussed in Section 5.4.2, the value of S_{RE} is determined dynamically. Thus, in our implementation, the vectorization degree of the sorting kernel K is adapted at run-time. After computation of the number

of transitions N_{trans} on the GPU, the value of N_{trans} is communicated to the CPU, and then used by the CPU to configure the vectorization degree of K before executing the kernel. The performance benefit here of dynamically optimizing the vectorization degree significantly overshadows the overhead of communicating the N_{trans} value from the GPU to the CPU.

The prefix sum operation in the TRT actor, and the reduction operations in the RRE, LFT, and TSD actors also represent performance bottlenecks. For these actors, we optimize the prefix sum and reduction implementations in a number of ways. First, we perform interleaved addressing so that active kernels have consecutive indices (IDs). We also implement sequential addressing for memory read and write operations in the GPU to avoid shared memory bank conflicts. Furthermore, we apply loop unrolling (e.g., see [44–46]) for further performance improvement.

5.5 Experiments and Analysis

In this section, we present experimental results of our novel system for gapless deep jitter measurement. The TDD that we apply is the Keysight U5303A PCIe High-Speed Digitizer [39]. This is a fast 12-bit PCIe digitizer with programmable on-board processing. The U5303A device stores acquired data on its on-board memory, and the data can then be transferred from the on-board memory to the host computer through a PCIe bus. The host computer that we use in our experiments contains a hybrid CPU-GPU platform. The platform includes an Intel Core i7-3820 quad-core CPU with an NVIDIA GeForce GTX680 GPU running Windows 7.

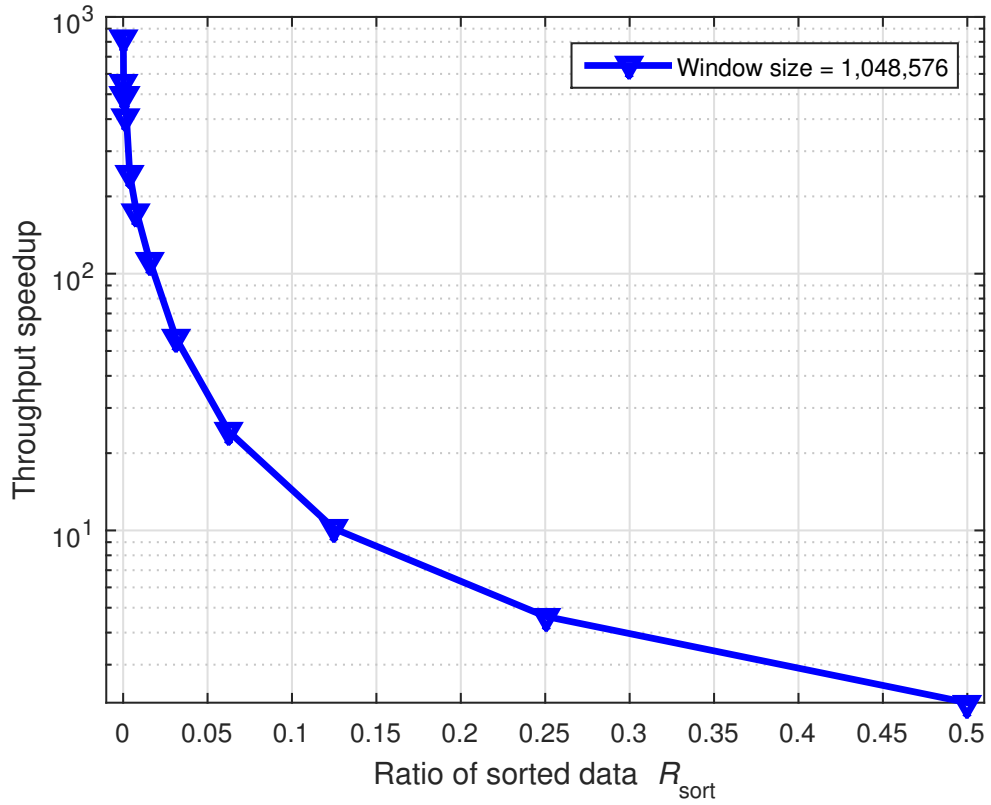

Figure 5.3: Throughput speedup for the DVL actor for varying values of R_{sort}.

OpenCL 1.2 and Visual Studio 2010 are used for code compilation.

5.5.1 Sorting in the Optimized DVL and RE Actors

In this section, we examine results related to the optimization techniques for sorting that were discussed in Section 5.4.2. Figure 5.3 shows the throughput speedup measured for the DVL actor as the ratio R_{sort} of data used for sorting is varied. For example, when $R_{\text{sort}} = 0.25$ (3 out of 4 samples are ignored), a speedup of 4.63 is obtained. The range of speedup values represented in Figure 5.3 is from 2.13 (when $R = 0.5$) to 822.57 (when $R = 0.00012$).

Figure 5.4, 5.5, 5.6 and 5.7 show the relative error of results produced by

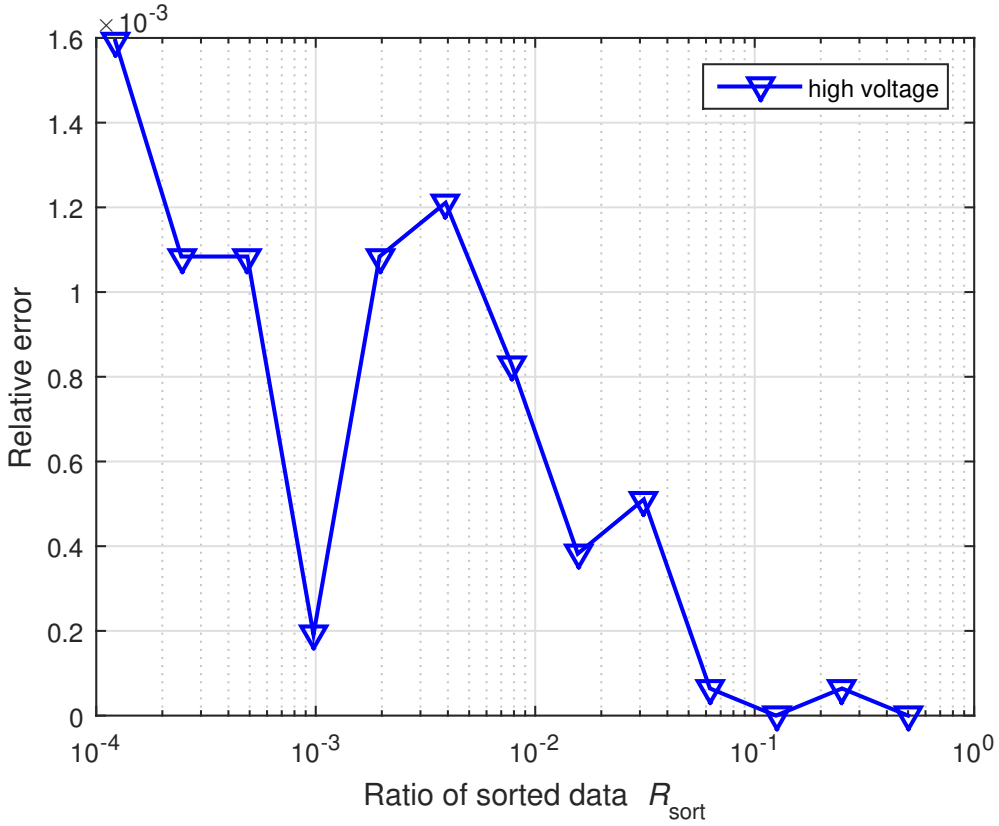

Figure 5.4: Relative error of high voltage threshold for various R_{sort} in DVL actor.

the DVL actor for varying values of R_{sort}. Figure 5.4 shows the relative error of the high voltage threshold, and Figure 5.5 gives corresponding results for the low voltage threshold. Figure 5.6 and Figure 5.7 show results on the recovered clock period and TIE standard deviation, respectively.

The input data set for this experiment is as described in Chapter 4. The results in Figure 5.4, Figure 5.5, Figure 5.6 and Figure 5.7 show that low levels of relative error (high levels of analysis accuracy) are observed across the entire range of R_{sort} values evaluated.

Figure 5.8 and 5.9 summarize experimental results on the throughput speedup and relative error for different values of R_{sort} in the RE actor. Figure 5.8 demon-

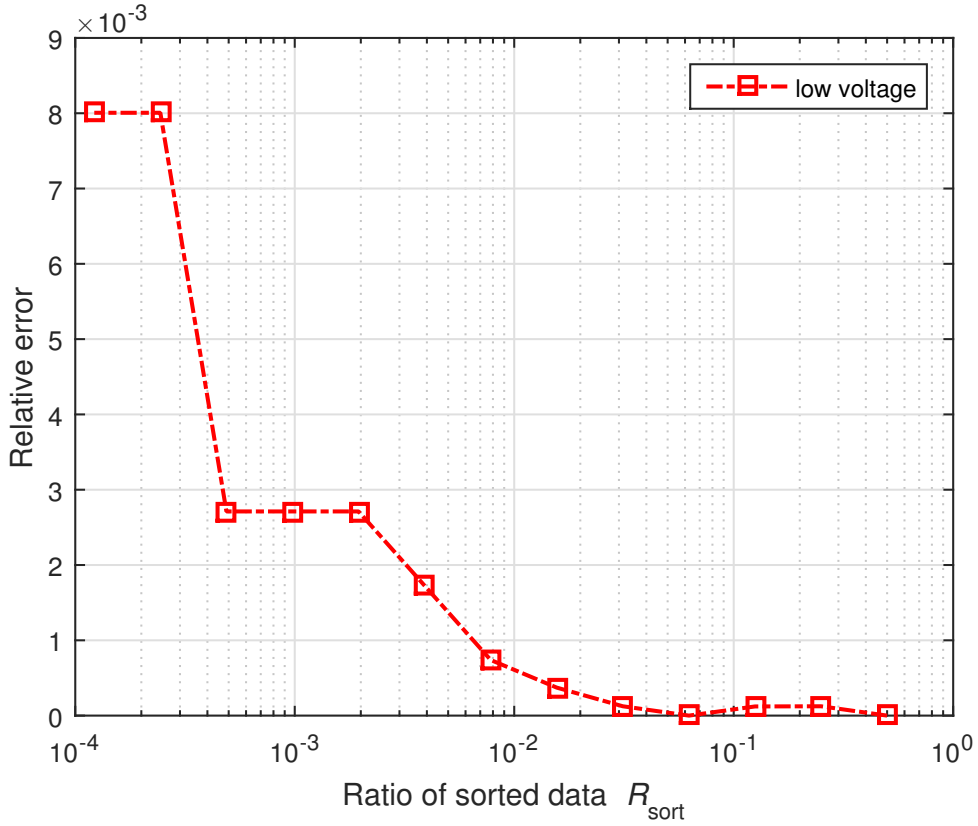

Figure 5.5: Relative error of low voltage threshold for various R_{sort} in DVL actor.

strates the relative error of results for different values of R_{sort}, and Figure 5.9 shows the throughput speedup for different values of R_{sort}.

The data in Figure 5.8 and 5.9 exhibits the same general trends observed in Figure 5.3, Figure 5.4, 5.5, 5.6 and 5.7 — significant speedups achieved with relatively low reduction in accuracy — although the magnitudes of throughput speedups are somewhat lower. Also, the throughput speedup is not linear. We expect that this is due to nonlinear effects related to memory cache operations and work group size organization in OpenCL.

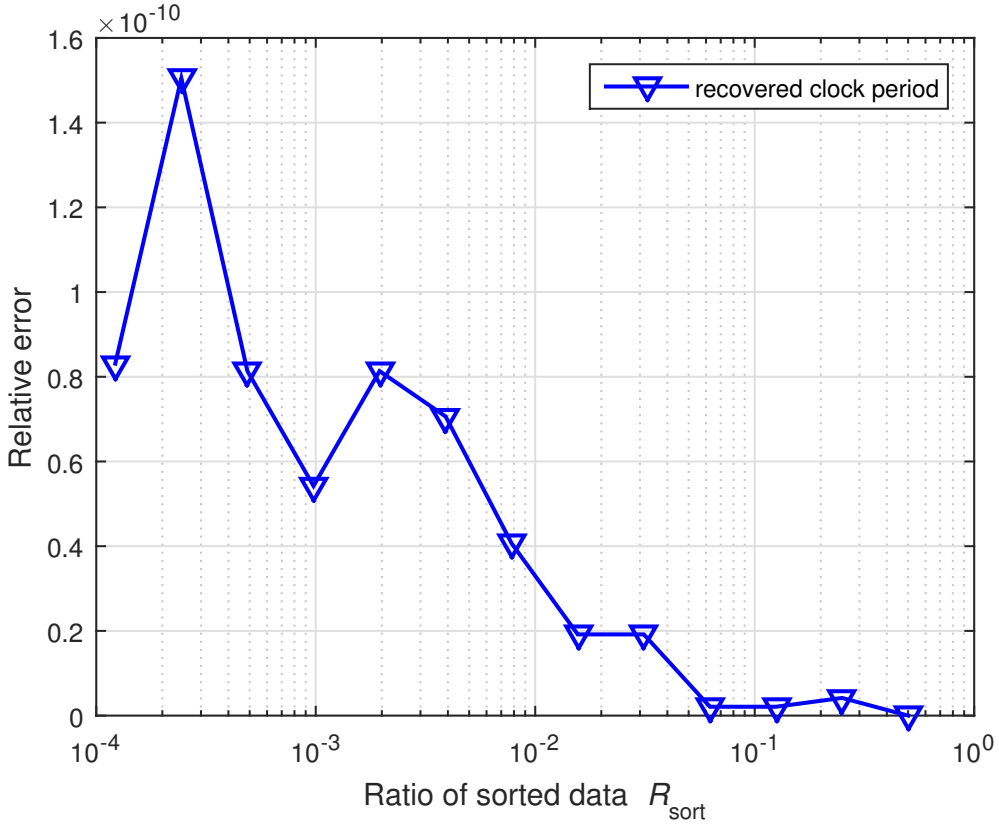

Figure 5.6: Relative error of recoved clock period for various R_{sort} in DVL actor.

5.5.2 Optimization of Reduction and Prefix Sum Operations

Table 5.2 shows the throughput speedup achieved for the three actors — TRT, RRE and LFT — that contain reduction and prefix sum operations. The design optimizations that produced these speedups were discussed in Section 5.5.5. The throughput values listed in the third and fourth columns of the table are in units of samples per second (SPS). A representative window size (given in the second column) is used in this experiment. Compared to the system design presented in Chapter 4, all three of these actors exhibit over 10X speedup. Unlike the optimizations related to manipulating R_{sort}, the optimizations examined in Table 5.2 do not affect signal

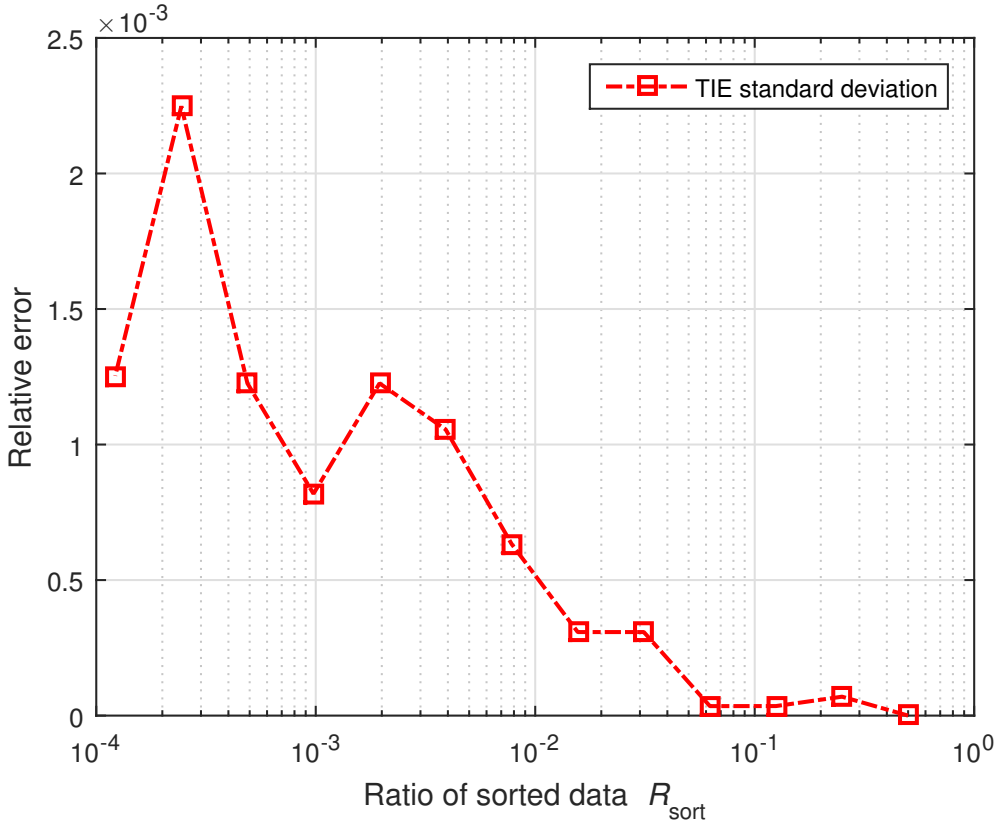

Figure 5.7: Relative error of TIE standard deviation for various R_{sort} in DVL actor.

processing accuracy.

Table 5.2: Throughput speedup for TRT, RRE and LFT actors.

Actor	Window Size (samples)	Baseline Throughput	Optimized Throughput	Speedup
TRT	1,048,576	1.38×10^7	1.63×10^8	11.79
RRE	1,048,576	2.08×10^8	6.88×10^9	33.12
LFT	1,048,576	5.02×10^7	2.17×10^9	43.26

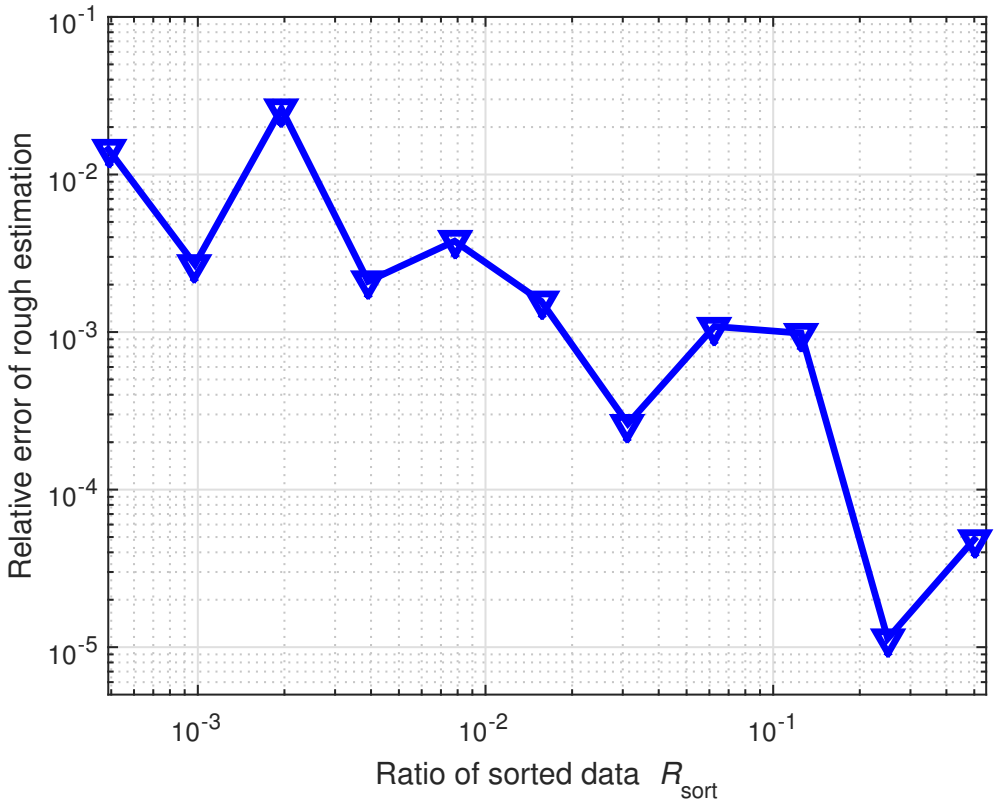

Figure 5.8: Relative error of rough estimation for various R_{sort} in RE actor.

5.5.3 Window Size Configuration

In our system design, we consider only powers of two for the window size. That is, the window size is always of the form $W_s = 2^k$ for some positive integer k. This power-of-two constraint is motivated by our use of bitonic sort, and parallel computations for prefix sum and reduction operations, as described in Section 5.4. In our design, all of these critical operations are performed more efficiently (e.g., by avoiding the need for zero padding) when the window size is a power of two.

In general, hardware characteristics may impose constraints on W_s for a given implementation. For example, the TDD that we apply has a minimum sampling

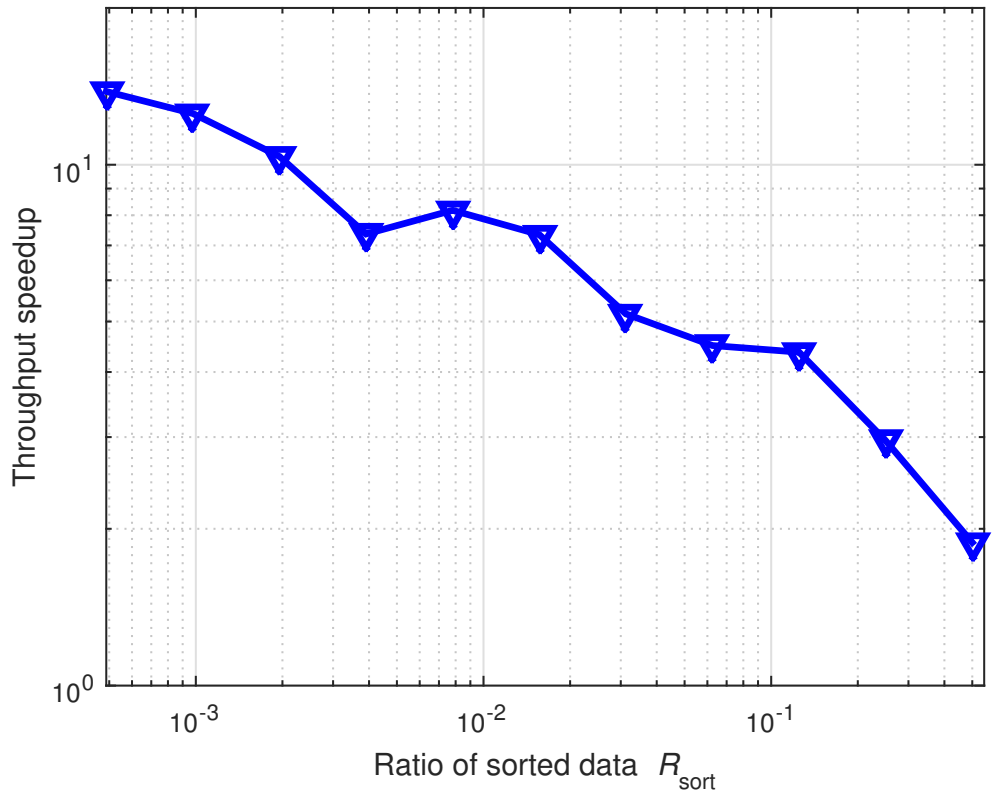

Figure 5.9: Throughput speedup for the RE actor for varying values of R_{sort}.

rate of 125M SPS. From our experiments involving system throughput (presented in Section 5.5.5), we have determined empirically that this minimum sample rate constraint leads to a minimum window size of $W_{\min} = 2^{21}$. This minimum window size is required to provide sufficient processing throughput to use the TDD. On the other hand, the memory constraints on the GPU of our target platform impose a maximum limit of $W_{\max} = 2^{22}$ on the window size. Thus, most of our experiments in the remainder of this section apply window sizes within the set $\{W_{\min}, W_{\max}\}$.

5.5.4 Overhead Analysis for Dynamic Adaptation

As described in Section 5.4.1 and Section 5.4.2, the window size W_s and the ratios R_{sort} of data samples to sort — for both the DVL and RE actors — are adapted dynamically based on continuously-monitored characteristics of the input signal.

Table 5.3 shows the execution time overhead measured for these dynamic adaptation operations. The overhead includes both the cost of computations to perform the relevant signal monitoring, and the cost of changing the relevant parameter settings in memory. The columns of the table correspond to the overhead of adapting W_s, R_{sort} for the DVL actor, and R_{sort} for the RE actor. The overhead is reported as a percentage of the total execution time for the optimized jitter measurement system as the window size is dynamically changed from W_{min} to W_{max}.

Table 5.3: Adaptation overhead in gapless jitter measurement system.

Window Size Configuration	Sorting Configuration for DVL Actor	Sorting Configuration for RE Actor
0.74%	0.0034%	0.0018%

5.5.5 System Throughput

Figure 5.10 shows the measured throughput of the jitter measurement system for different values of the window size W_s. These results are shown for a representative input signal frequency of 800kHz. The implementation is tested with 10 different window sizes from 2^{13} to 2^{22} (W_{max}), with each value of W_s corresponding

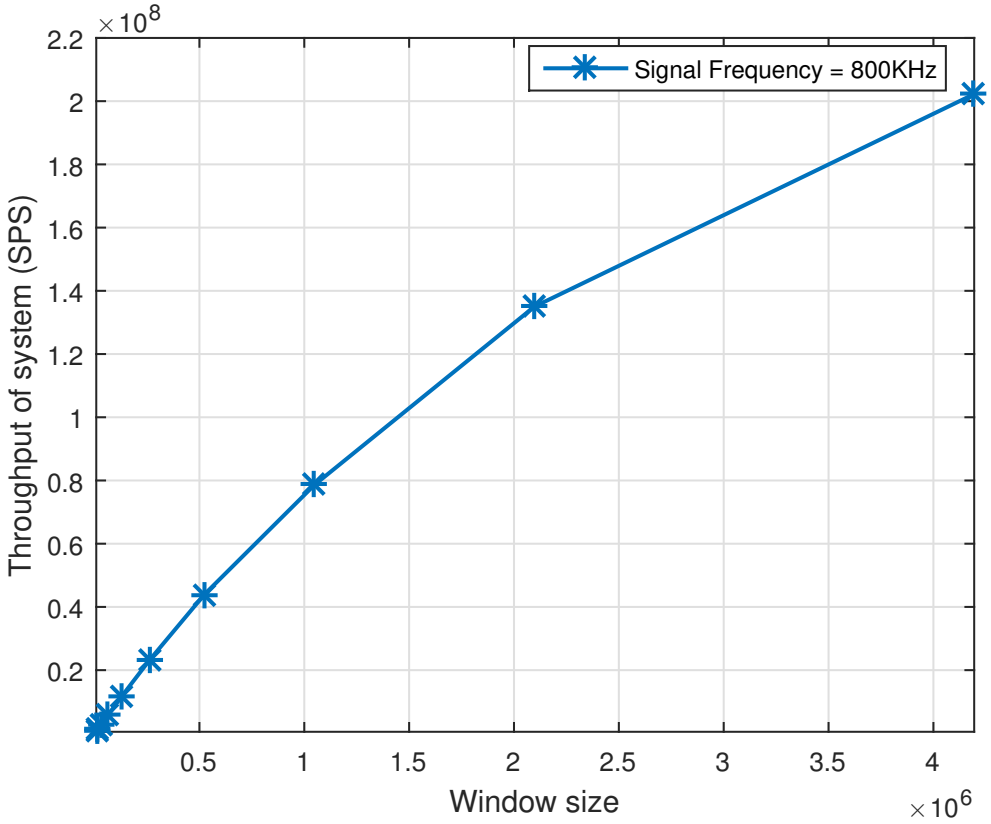

Figure 5.10: System throughput versus window size.

to a different power of 2. The results demonstrate that system throughput increases consistently with increases in W_s. We expect that this trend is due to the enhanced performance of parallel operations with increased window sizes. However, increases in W_s also result in CPU-GPU communication and memory operations accounting for larger percentages of the overall execution time. Thus, with increases in W_s, we see a decrease in the rate of throughput increase.

Figure 5.11 shows the system throughput for different signal frequencies when the window size is fixed at $W_s = W_{\min}$. The results show relatively small variation in throughput for different frequencies. More specifically, the relative difference between different levels of throughput is less than 1.5%.

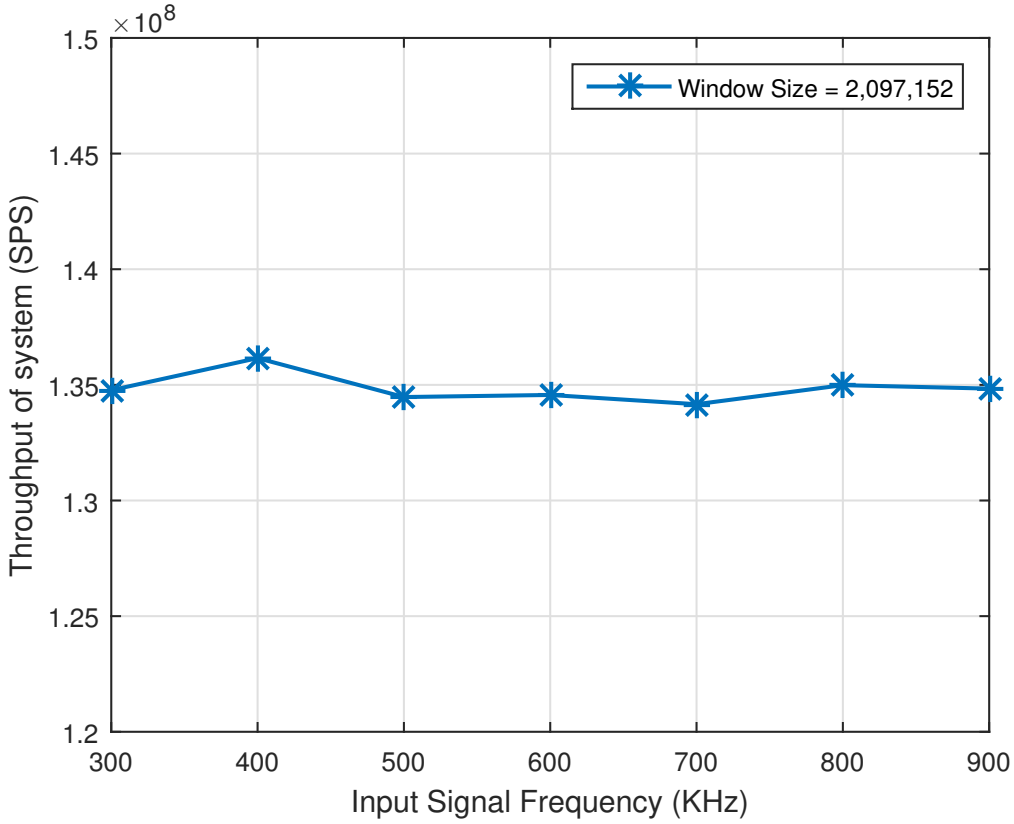

Figure 5.11: System throughput versus frequency.

In summary, the experimental results presented in this section demonstrate significant improvements achieved by the design optimization techniques applied in our novel system for deep jitter measurement. Additionally, the results demonstrate low levels of accuracy loss in the approximate computing approaches that we applied to improve the performance of sorting operations. Furthermore, our results provide quantitative insight into other relevant trends in dynamic adaptation overhead and overall system performance.

Chapter 6: Generalized Graph Connections for Dataflow Modeling

As discussed in Chapter 2.1, dataflow representations are directed graphs in which vertices represent computations and edges correspond to buffers that store data as it passes between computations. The buffers are single-input, single-output components that manage data in a first-in, first-out (FIFO) fashion. In this chapter, we generalize the concept of dataflow buffers with a concept called *passive blocks*. Like dataflow buffers, passive blocks are used to store data during the intervals between its generation by producing actors, and its use by consuming actors. However, passive blocks can have multiple inputs and multiple outputs, and can incorporate operations on and rearrangements of the stored data subject to certain constraints. We define a form of flowgraph representation that is based on replacing dataflow edges with the proposed concept of passive blocks. We present a structured design methodology for utilizing this new form of signal processing flowgraph, and demonstrate its utility in improving memory management efficiency, and execution time performance for deep waveform applications.

Material described in this chapter will be published in the Proceedings of the 2018 IEEE Workshop on Signal Processing Systems.

6.1 Introduction

Dataflow modeling is widely used in design processes and tools for signal processing systems. In this form of modeling, applications are represented as directed graphs, called *dataflow graphs*, in which vertices (*actors*) represent discrete computations that are executed iteratively (*fire*) to process semi-infinite streams of input data. Each edge $e = (x, y)$ in a dataflow graph represents a logical communication channel between actors x and y. More specifically, each $e = (x, y)$ represents a first-in, first-out (FIFO) buffer that stores data during the period between its production by actor x and its consumption by actor y. Actors can be fired when certain conditions, referred to as *firing rules*, are satisfied [4].

Dataflow modeling has proven to be of great utility in the design and implementation of signal processing systems for various reasons, including its provisions for ensuring determinacy, support for exploiting parallelism, and capability for exposing high-level application structure that is useful for many kinds of design optimization beyond those associated with exploiting parallelism [38].

A limitation of signal processing dataflow representations, however, is that they are inefficient in describing inter-actor communication patterns that depart from the simple single-input, single-output (SISO) interface and FIFO behavior that are defined for dataflow edges. As a canonical example of this kind of inefficiency, consider the *fork* actor illustrated in Figure 6.1. This is a synchronous dataflow (SDF) [9] actor that consumes a single token t and produces two tokens — one on each output edge — on each firing. The values of the two tokens that are produced

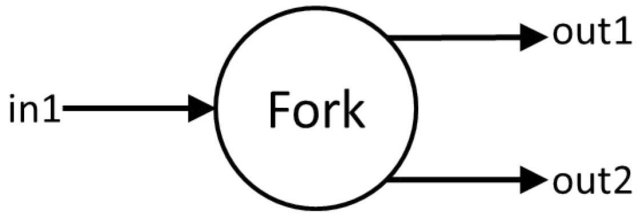

Figure 6.1: Fork actor illustration.

```
Firing of fork actor:
1. Read data from input FIFO in1 and store it
   in a temporary variable d1
2. Send data value in d1 to output FIFO out1
3. Send data value in d1 to output FIFO out2
```

Figure 6.2: Pseudo code of fork actor.

are identical to the value of the input token t. Thus, this actor can be viewed as providing a kind of broadcast functionality.

Figure 6.2 shows a pseudocode fragment for the fork actor. From this pseudocode, we can see that there is overhead of copying the value of the input token to each of the outputs. This overhead in general includes a run-time cost as well as a cost in terms of increased memory requirements. The overhead is required under a pure dataflow interpretation since the input token must be replicated on each of the two output edges (FIFOs).

The functionality of the fork actor can be realized more efficiently if we abandon this pure dataflow interpretation, and implement the actor instead using the one-input, two-output component illustrated in Figure 6.3. This component, which we refer to here as a *passive fork*, is not *fired* as a dataflow actor is. Instead, the component operates in a manner similar to a typical FIFO implementation, where

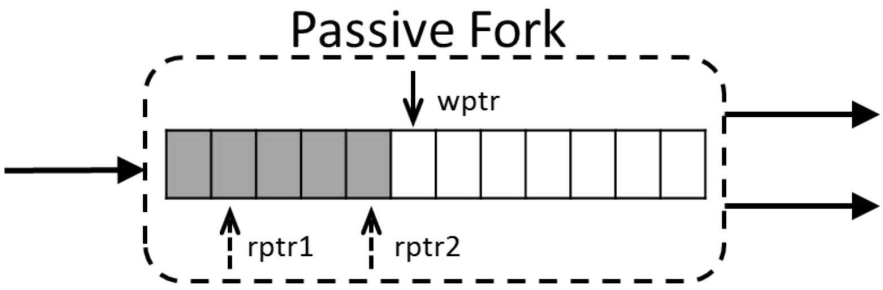

Figure 6.3: Passive form of fork actor.

a buffer is associated with the component, and tokens are written to and read from the buffer using write and read pointers, respectively. However, the passive fork has *two* read pointers — one corresponding to each output edge of the fork actor — instead of the single read pointer that would be used in a FIFO. In effect, we have transformed the fork actor, which operates in an "active" manner (by firing) into an passive component, which is used by writing to and reading from the component's ports. In this passive version of fork actor, wptr, rptr1, and rptr2 in the figure show possible positions of the write pointer and the two read pointers.

A more powerful form of this "active-to-passive" conversion is illustrated on Figure 6.4, which shows a gain actor that is connected at the input of the fork actor. This gain actor corresponds to a constant multiplication, where the constant factor k is a parameter of the actor. The gain together with the fork can be replaced by a single passive component. This component is similar to the passive fork actor, except that when a value is written into the buffer, it is multiplied by k before being stored. In this chapter, we generalize this process of converting certain kinds of actors into passive components, which achieve equivalent functionality through read/write interfaces rather than through the mechanism of being fired. This gen-

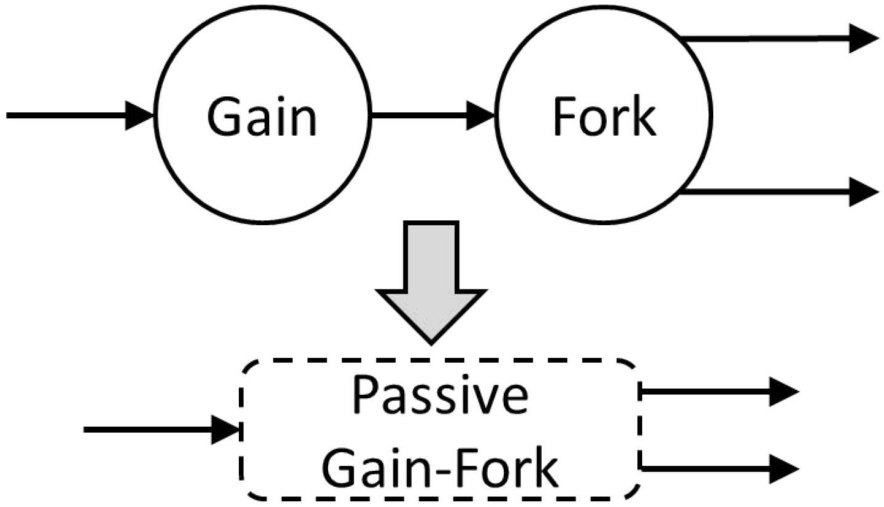

Figure 6.4: Gain actor illustration.

eralization leads to a powerful new design methodology in which passive components of arbitrary complexity can be designed to provide streamlined functionality for actors or subgraphs that are more efficiently realized with internal buffers and read/write interfaces. When dataflow graphs are transformed to incorporate such passive components, we refer to the resulting graphs as *Passive-Active FlowGraphs* (*PAFGs*). A central objective of this chapter is to introduce PAFGs as a useful new representation for model-based design of signal processing systems.

6.2 Related Work

Many researchers have investigated efficient buffer memory management in dataflow graphs (e.g., see [38,47–49]). Bhattacharyya and Lee discussed the concept that certain actors, such as the fork actor described above, can be implemented more efficiently by deviating from pure dataflow semantics [50]. However, this earlier work did not propose any approach for integrating such deviations systematically into the

modeling framework. In this new work, we develop such a systematic approach based on the novel abstraction of PAFGs.

Perhaps the most closely related form of dataflow memory management optimization to what we develop in this chapter is *buffer merging*, which involves mapping subsets of input and output buffers of a given actor to a common memory space (e.g., see [51, 52]). Like the method of [52], the PAFG approach allows for memory sharing across arbitrary numbers of input and output buffers for a given actor. Similarly, like the method of [51], the PAFG approach does not involve expansion to a single rate graph, which can be costly in terms of compiler memory requirements and time complexity for highly multirate applications (e.g., see [53]). In this sense, the PAFG approach provides a novel combination of useful features in the two previously developed buffer merging approaches described above. Additionally, while the methods of [51, 52] are limited to SDF graphs, the PAFG approach is not restricted to any specific form of dataflow. For example, Boolean dataflow switch and select actors [54] can be formulated as optimized PAFG components using the same methodology that is presented in this chapter. Applicability beyond SDF is also a distinguishing point compared to the abstraction of deterministic SDF with shared FIFOs (DSSF) [55].

While there are significant differences between buffer merging, DSSF, and PAFG-based memory management, investigating and exploiting complementary relationships among the different approaches is an interesting direction for future work.

6.3 PAFG Representations

In this section, we develop in detail the PAFG model of computation. In this work, PAFGs are derived from dataflow graphs, and are intended as intermediate representations or implementation architectures for dataflow *application graphs* (dataflow models of signal processing applications). For concreteness, we develop the concepts of PAFGs here in the context of core functional dataflow (CFDF) as the application graph model; however, the concepts are not specific to CFDF and can be adapted to other forms of dataflow. CFDF is a highly expressive model that can be used to represent other well-known dataflow models, including synchronous, cyclo-static, and Boolean dataflow [56]. CFDF is the model that underlies the lightweight dataflow environment (LIDE) tool [57], which we use for our experiments in Section 6.5.

We first define some notation that will be useful throughout the remainder of this chapter. Given an edge e in a directed graph, we denote the source and sink vertices of e by $src(e)$ and $snk(e)$, respectively. A *self-loop* is an edge whose source and sink vertices are identical. In the remainder of this chapter, we consider only directed graphs that do not contain self-loops. Self-loops can be incorporated easily into the methods developed in this chapter; we omit the details for brevity.

Given an edge e, we say that $src(e)$ is a *predecessor* of $snk(e)$, $snk(e)$ is a *successor* of $src(e)$, and $src(e)$ and $snk(e)$ are *adjacent* vertices. The sets of all predecessors and successors of a vertex v in a given graph are denoted by $pred(v)$ and $succ(v)$, respectively. The sets of all input edges and output edges of v are

denoted by $in(v)$ and $out(v)$, respectively.

We refer to PAFG vertices as *blocks*. In a dataflow graph, vertices correspond to computational modules, and edges correspond to SISO buffers between the modules. In contrast, in a PAFG, both computational modules and buffers are represented as vertices, and edges represent connections between computational modules and buffers. Additionally, PAFG buffers are not restricted to SISO interfaces — they can have multiple inputs, multiple outputs, or both. A third distinguishing characteristic of the PAFGs that we are interested in this chapter is that they are bipartite graphs. We define this bipartite characteristic precisely in Section 6.3.3.

For conciseness and clarity, we assume that dataflow graphs and PAFGs are directed graphs rather than multigraphs (which can contain multiple edges directed in the same direction and between the same pair of vertices). The adaptation of the PAFG model to multigraphs can be readily achieved when implementing the model.

Additionally, we develop the concepts of dataflow graphs and PAFGs in terms of vertices and edges and avoid details associated with *ports*, which provide interfaces between vertices and incident edges. In practical dataflow design processes and tools, multigraph and port representations are both important. The methods developed in this chapter can be extended naturally to incorporate such representations. However, because these extensions are not essential to conveying the main ideas of this chapter, we avoid introducing the additional notation required to accommodate them.

We refer to an ordered pair of actors (x_d, y_d) as a *dataflow pair* and an ordered pair of PAFG blocks (x_b, y_b) as a *PAFG pair*.

The PAFGs that we are concerned with in this chapter are derived from corresponding dataflow graphs (application graphs). We elaborate on the process of deriving a PAFG from a dataflow graph in Section 6.3.4. This derivation process places blocks in a PAFG F in correspondence with actors or edges in the dataflow graph from which F was derived. A *simple passive buffer* is a PAFG block that corresponds in this way to an edge in some dataflow graph. A PAFG block that is not a simple passive buffer is referred to as a *non-simple block*. We often refer to simple passive buffers as *simple blocks*.

6.3.1 PAFG Blocks

A PAFG block is either a *passive* block or an *active block*. The distinction between these two types was motivated intuitively in Section 6.1. More precisely, an active block corresponds to an application graph actor that is used in the usual way — that is, through interfaces that are associated with firing the actor and (if available) for testing fireability. In CFDF, these are referred to as *invoke* and *enable* interfaces, respectively [56]. In contrast, a passive block is used through read/write interfaces, as illustrated by the passive fork example in Section 6.1.

Given an application graph G, we assume that implementations of the actors in G are available in an *actor library*. We assume that each actor in G has one active implementation (with enable/invoke interfaces) in the library, and that it may or may not have a passive implementation (with read/write interfaces). We refer to an actor A as a *buffer actor* if it has a passive implementation; otherwise, we refer to A

as a computational actor. Thus, only buffer actors can be placed in correspondence with passive blocks.

Like active blocks, non-simple passive blocks correspond to actors. However, they are used (executed) in a different way — again, as illustrated by the difference between the active ("standard") and passive versions of the fork actor in Section 6.1. A non-simple passive block should implement the same input/output behavior as its corresponding actor — that is, it should perform the same mapping from input streams into output streams. For background on the interpretation of actors as mappings from input streams to output streams, we refer the reader to [4]. In this work, we assume that unit testing processes are used to validate such "mapping equivalence" between passive blocks and their corresponding actors. For background on synergies between unit testing and dataflow-based design processes, we refer the reader to [57]. We envision as an interesting area for future work the automation of the equivalence checking process between active and passive implementations of the same buffer actor.

A block in a PAFG is either a *computational block* or a *buffer block*. The computational/buffer dichotomy is another relevant way to distinguish between blocks in addition to the active/passive and simple/non-simple dichotomies. All computational blocks are active blocks. However, buffer blocks can in general be either passive or active.

6.3.2 Coordination Functions and Alternating PAFGs

When deriving a PAFG, each buffer block needs to be designated as being an active or passive buffer. An active buffer is executed like any other actor (using enable/invoke interfaces), while passive buffers are read from and written to directly by computational blocks and active buffers (using read/write interfaces). Coordination functions are used to specify whether a given block is executed in a passive or active fashion. Thus, coordination functions specify how schedulers should manipulate the blocks when executing the associated application graph.

Given a PAFG F, we represent the set of blocks (vertices) in F by $blks(F)$, and we define a *coordination function* of F as one that specifies for each $b \in blks(F)$ whether or not b is to be executed in an active or passive fashion. More precisely, a coordination function is a mapping $C : blks(F) \rightarrow \{pssv, actv\}$, where $C(b_c) = actv$ for every computational block $b_c \in blks(F)$, and $C(b_s) = pssv$ for every simple block $b_s \in blks(F)$. We refer to $C(b)$ as the *coordination type* of block b with respect to C. Computational blocks and simple blocks must be coordinated in an active and passive fashion, respectively, and a coordination function just "reminds us" of this. On the other hand, a coordination function C specifies for each non-simple buffer block whether or not the block is to be executed in a passive or active fashion (if we execute the PAFG based on C).

A coordinated PAFG is an ordered pair $Z = (F, C)$, where F is a PAFG and C is a coordination function for F.

6.3.3 Alternating PAFGs

In this work, we are interested in a specific form of coordinated PAFG, which we refer to as an *alternating PAFG*. An alternating PAFG is defined to be a coordinated PAFG that is bipartite in terms of the active blocks and passive blocks. More precisely, an alternating PAFG $Z = (F, C)$ with $F = (V_f, E_f)$ is one that satisfies $C(src(e)) \neq C(snk(e))$for all $e \in E_f$.

A block in a PAFG is an *interface block* if it has no output edges or it has no input edges. The concept of coordinated PAFGs allows for the possibility of interface blocks that are passive. However, we have not yet experimented with the design of passive interface blocks. Exploration into the utility of passive interface blocks appears to be an interesting direction for future work.

In our context, direct communication between pairs of active blocks or pairs of passive blocks is ambiguous. Intuitively, some form of buffer is needed to manage the flow of data between active blocks (just as dataflow edges connect pairs of communicating actors in dataflow graphs). Generalization of the developments of this chapter beyond alternating PAFGs is potentially another interesting direction for future work.

6.3.4 Direct PAFGs

We propose a design methodology in which dataflow graphs are converted into a kind of equivalent PAFG representation, and then transformed so that some subset of the active buffers is converted into passive coordination form. In this section, we

define the equivalent PAFG representation, which we refer to as *direct PAFG* form, and in Section 6.4, we define the process of transforming active buffers into passive form.

Suppose that we are given a dataflow graph $G = (V, E)$. For each edge $e \in E$, we define a corresponding passive buffer $\rho(e)$. We denote the set of passive buffers defined in this way as $P(G)$. Thus, $P(G) = \{\rho(e) \mid e \in E\}$. Each $\rho(e) \in P(G)$ is a simple block (see Section 6.3.1) since it is defined in correspondence with a distinct dataflow graph edge e.

Similarly, for each $v \in V$, we define a corresponding block $\alpha(v)$. Each $\alpha(v)$ is referred to as an *actor block* with corresponding actor v. If v is a computational actor, then $\alpha(v)$ is defined as a computational block. Otherwise, $\alpha(v)$ is defined as a non-simple buffer block. For a given dataflow graph $G = (V, E)$, we define the set of all actor blocks by $\mathcal{A}(G) = \{\alpha(v) \mid v \in V\}$

For each $z = \rho(e) \in P(G)$, we define the PAFG pairs $\kappa_i(z) = (\alpha(src(e)), z)$ and $\kappa_o(z) = (z, \alpha(snk(e))$. Recall that PAFG pairs are ordered pairs of blocks, and actor blocks and passive buffers both represent different types of blocks. Thus, $\kappa_i(z)$ and $\kappa_o(z)$ can correctly be referred to as PAFG pairs. The sets of all pairs defined in this way are represented by $\mathcal{K}_i = \{\kappa_i(z) \mid z \in P(G)\}$, and $\mathcal{K}_o = \{\kappa_o(z) \mid z \in P(G)\}$.

The *direct PAFG* representation of G is a coordinated PAFG $Z_d = (F_d, C_d)$. The PAFG $F_d = (V_d, E_d)$ is defined by $V_d = (\mathcal{A}(G) \cup P(G))$, and $E_d = (\mathcal{K}_i \cup \mathcal{K}_o)$, and the coordination function is defined by $C_d(b) = actv$ for every non-simple block b.

By construction, each edge in Z_d connects a simple block to a computational

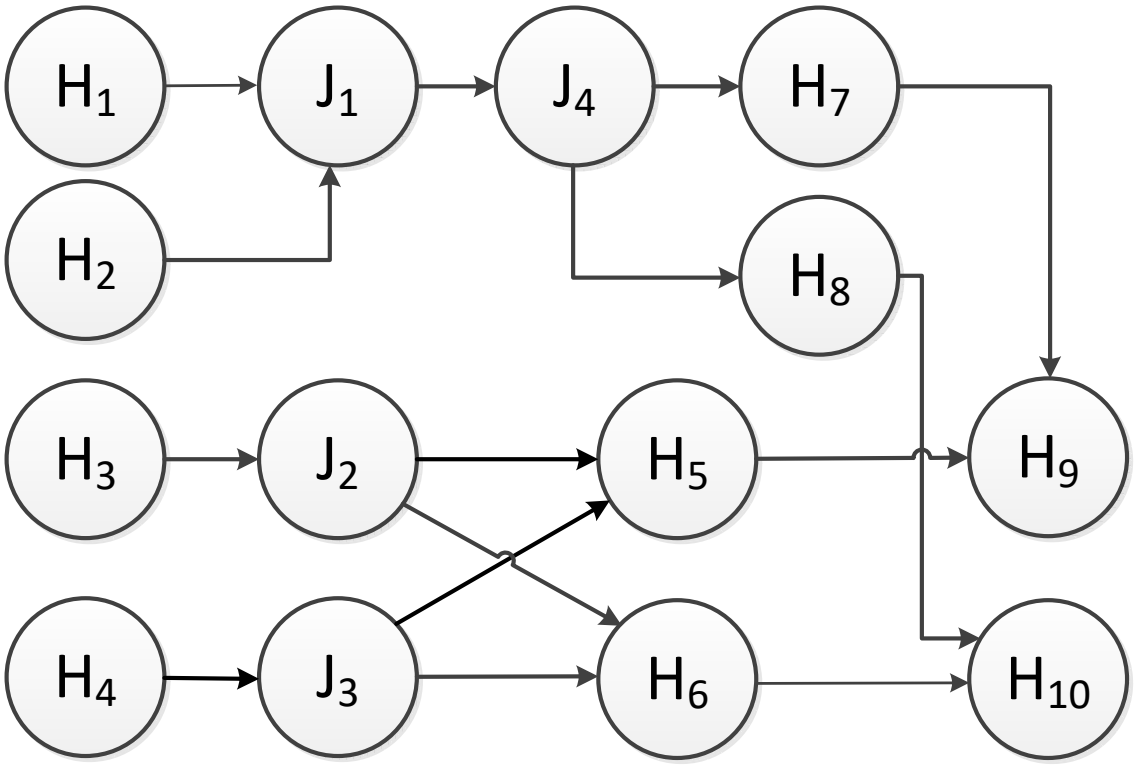

Figure 6.5: A dataflow graph illustration (application graph).

Table 6.1: Coordination function for the direct PAFG of Figure 6.6.

Block (B)	Coordination type $C(B)$
$Y_i, i = 1, 2, \ldots, 10$	*actv*
$Z_j, j = 1, 2, \ldots, 4$	*actv*
$L_k, k = 1, 2, \ldots, 15$	*pssv*

block or an active buffer block. Thus, a direct PAFG is always an alternating PAFG.

To illustrate key concepts introduced in this section, Figure 6.5 shows an example of a dataflow graph (application graph), Figure 6.6 shows the direct PAFG that results from this application graph, and Table 6.1 shows the coordination function for the direct PAFG. In Figure 6.5 and Figure 6.6, each H_i is a computational actor, each J_i is a buffer actor, each Y_i corresponds to H_i, each Z_i corresponds to J_i, and each L_i is a simple passive buffer.

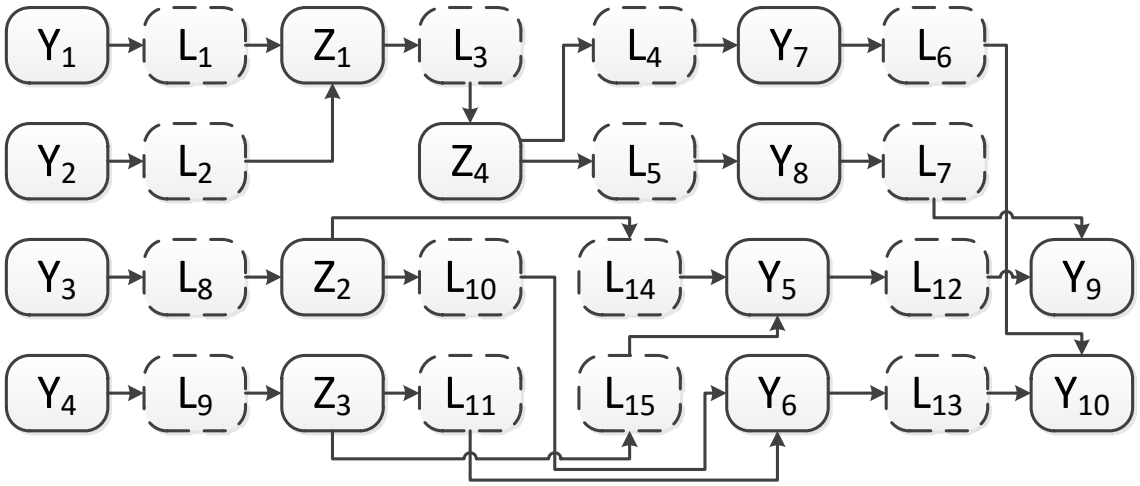

Figure 6.6: The direct PAFG that is derived from Figure 6.5.

As illustrated in Figure 6.5 and Figure 6.6, we use the convention that dataflow graph actors are drawn with circles, PAFG blocks are drawn with rectangles or squares, and the borders of PAFG blocks are solid or dashed based on whether the blocks are active or passive, respectively.

6.3.5 Association between Dataflow Graphs and PAFGs

Given a dataflow graph $G = (V, E)$ and a PAFG $F = (V_f, E_f)$, we say that G and F are *associated* (each is associated with the other) if each simple block p in F corresponds to an edge e in G ($p = \rho(e)$), and each non-simple block q in F corresponds to an actor a in G ($q = \alpha(a)$). By construction, the direct PAFG representation of a dataflow graph G is always associated with G.

6.4 Passivization Transformation

In the direct PAFG representation of a dataflow graph, all non-simple buffer blocks are coordinated as active buffers. In this section, we define the process of converting an active buffer to passive form. This conversion process is defined as a transformation process for alternating PAFGs — that is, a process that takes as input an alternating PAFG and produces as output another alternating PAFG.

If b and c are adjacent blocks in a PAFG, then we disallow coordination functions that assign a passive form to both b *and* c. We refer to this as the *adjacent buffer coordination* (*ABC*) restriction. We impose the ABC restriction because we do not have any mechanism defined for direct communication between two passive blocks. Intuitively, communication between passive buffer blocks "stalls" because each is "waiting" for a read or write operation to be initiated by the other. It may be interesting as future work to investigate communication mechanisms that allow one to relax the ABC restriction.

Given an alternating PAFG (F, C) and a block b in F, we say that b is *simply surrounded* if all of its predecessors and successors are simple passive buffers. Formally, this means that x is a simple passive buffer for all $x \in (pred(b) \cup succ(b))$. For example, in Figure 6.6, blocks Z_1 and Z_2 are simply surrounded, while blocks L_1 and L_2 are not.

Suppose that we have an alternating PAFG $Z_a = (F_a, C_a)$, where $F_a = (V_a, E_a)$, and suppose we have an active buffer $\beta \in V_a$ that is simply surrounded. Then we can perform the *passivization transformation* of Z_a with respect to β. This

transformation, which is the primary contribution of this section, produces a new PAFG $Z_b = (F_b, C_b)$, $F_b = (V_b, E_b)$. The vertex set of F_b is defined by the set difference $V_b = V_a - V_z$, where $V_z = pred(\beta) \cup succ(\beta)$.

To define the edge set E_b, we first define the sets $Y_p = \{y \in pred(x) \mid x \in pred(\beta)\}$, and $Y_s = \{y \in succ(x) \mid x \in succ(\beta)\}$. Since β is simply surrounded, we have from the ABC restriction that all elements of Y_p and Y_s are active blocks. Next, we construct the set E_β of PAFG pairs that are directed from members of Y_p to β, or from β to members of Y_s: $E_\beta = (\{(x, \beta) \mid x \in Y_p\} \cup \{(\beta, y) \mid y \in Y_s\})$. We also define the set of all input and output edges of blocks that are adjacent to β: $E_r = \{e \in out(x) \mid x \in V_z\} \cup \{e \in in(x) \mid x \in V_z\}$. We can then define E_b by $E_b = ((E_a - E_r) \cup E_\beta)$.

The coordination function $C_b : V_b \to \{pssv, actv\}$ is derived by changing the form of β, while "copying" the values from C_a for all other blocks in V_b: $C_b(\beta) = pssv$, and $C_b(x) = C_a(x)$ for all $x \in (V_b - \{\beta\})$.

To summarize, the passivization transformation with respect to a simply surrounded active buffer β involves the following steps:

1. Change the form of β from *actv* to *pssv*;
2. Remove all of the predecessor and successor blocks of β along with their input and output edges;
3. Add edges that are directed to β from each member of Y_p;
4. Add edges that are directed from β to each member of Y_s.

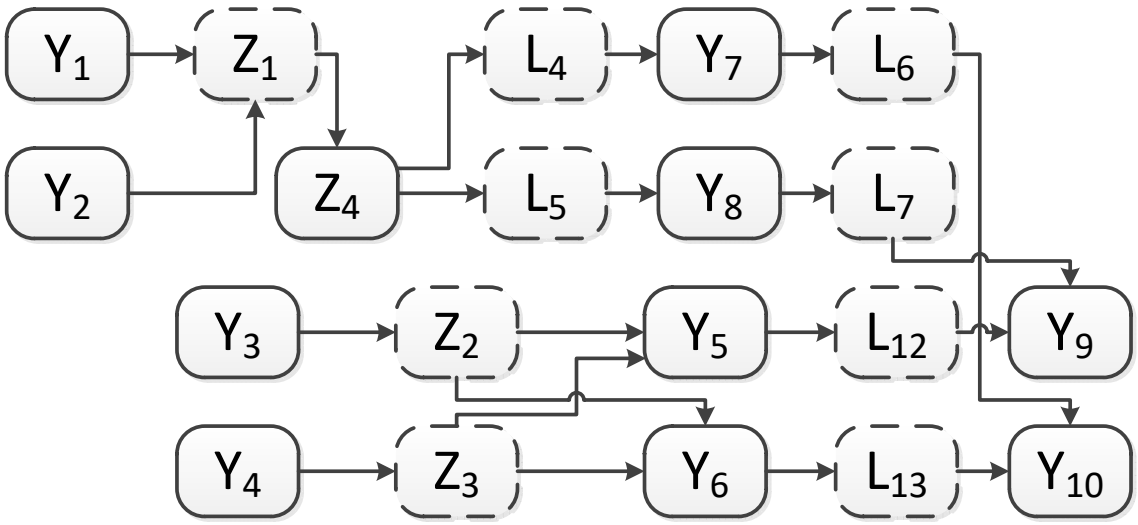

Figure 6.7: Resulting PAFG after applying the passivization transformation.

Table 6.2: Coordination function for the PAFG in Figure 6.7.

Block (B)	Coordination type $C(B)$
$Y_i, i = 1, 2, \ldots, 10$	*actv*
$Z_j, j = 1, 2, 3$	*pssv*
Y_4	*actv*
$L_k, k = 4, 5, 6, 7, 12, 13$	*pssv*

The passivization transformation can be applied multiple times, where in each application (transformation step) after the first, the transformation is applied on the graph that results from the previous step.

For example, Figure 6.7 illustrates the PAFG that results after applying the passivization transformation three times on the direct PAFG of Figure 6.6. The transformation is applied with respect to Z_1, Z_2, and then Z_3. The coordination function associated with Figure 6.7 is illustrated in Table 6.2

6.5 Application Examples and Experiments

In this section, we present experiments on two relevant applications. These experiments demonstrate the utility of design optimization using PAFGs. In both of these experiments, we carried out a sequence of passivization transformations by hand, and implemented the original dataflow graph and the optimized PAFG (derived through the transformations) using the lightweight dataflow environment (LIDE) [57]. In this work, we have developed extensions in LIDE to provide complete support for design and implementation using PAFGs, including features that allow implementation and interfacing of non-simple passive blocks. The experiments for both applications are conducted on an Intel Core i7-2600K Quad-core CPU running Ubuntu Linux 16.04 LTS, and using GCC 5.4.0 for code compilation.

6.5.1 Error Vector Magnitude Computation

The error vector magnitude (EVM) is a figure of merit for signal quality in communication systems. EVM computation is an important application in measurement and test equipment for communications. For background on EVM computation, we refer the reader to [58].

A dataflow graph for measuring the EVM for a given reference signal and received signal is shown in Figure 6.8. This is a dynamic dataflow graph modeled using CFDF semantics, as supported in LIDE. Here, SRC_1 provides on each ith firing the input data length for the ith EVM computation. The actors SRC_2 and SRC_3 provide the real and imaginary parts, respectively, of the reference signal;

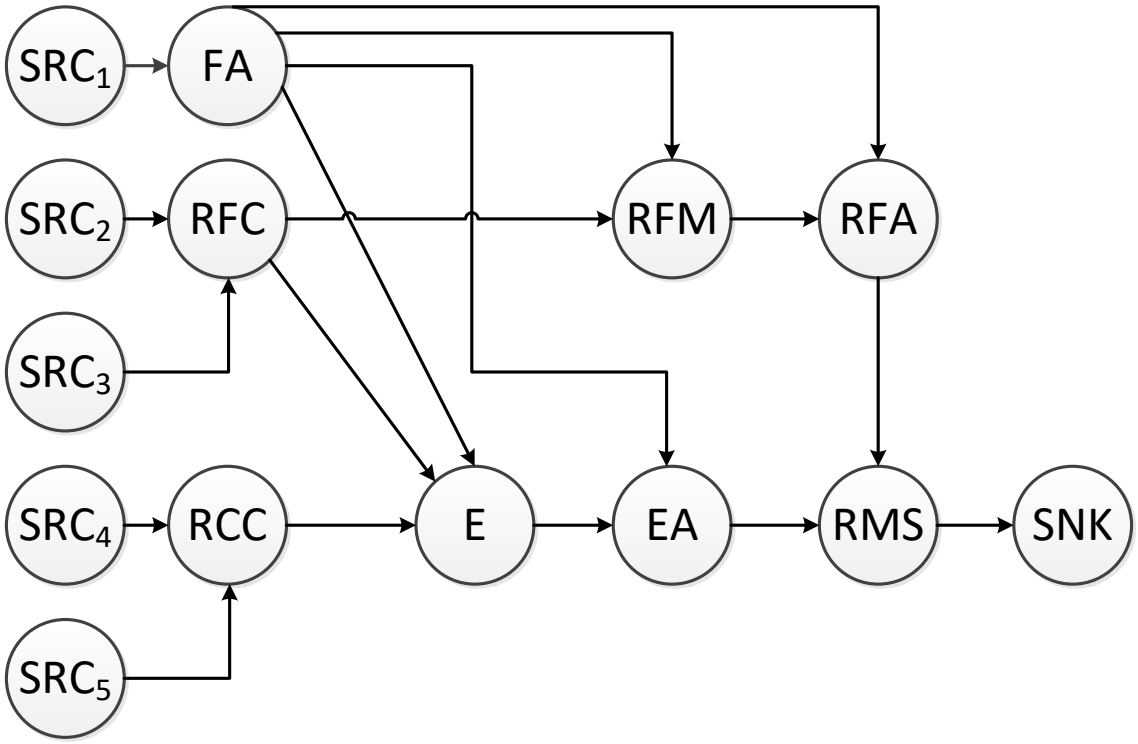

Figure 6.8: Dataflow graph for EVM measurement.

and similarly, SRC_4 and SRC_5 provide the real and imaginary parts of the received signal. The actor FA is a fork actor (see Section 6.1), which broadcasts data to multiple output ports. The actors RFC and RCC are interleavers that interleave corresponding pairs of input tokens so that the real and imaginary parts of each signal sample are arranged in successive elements of the actors' output streams. The actors E and RFM compute the error vector and reference signal magnitude, respectively. The actor RMS computes the root mean square (RMS) k_e of the error signal and the RMS k_r of the reference signal, and derives the EVM result as the ratio k_e/k_r. The actors RFA and EA compute the average magnitudes of the reference and error signals, respectively. The SNK actor represents the output interface of the graph; in our experiments, we use a file writing interface.

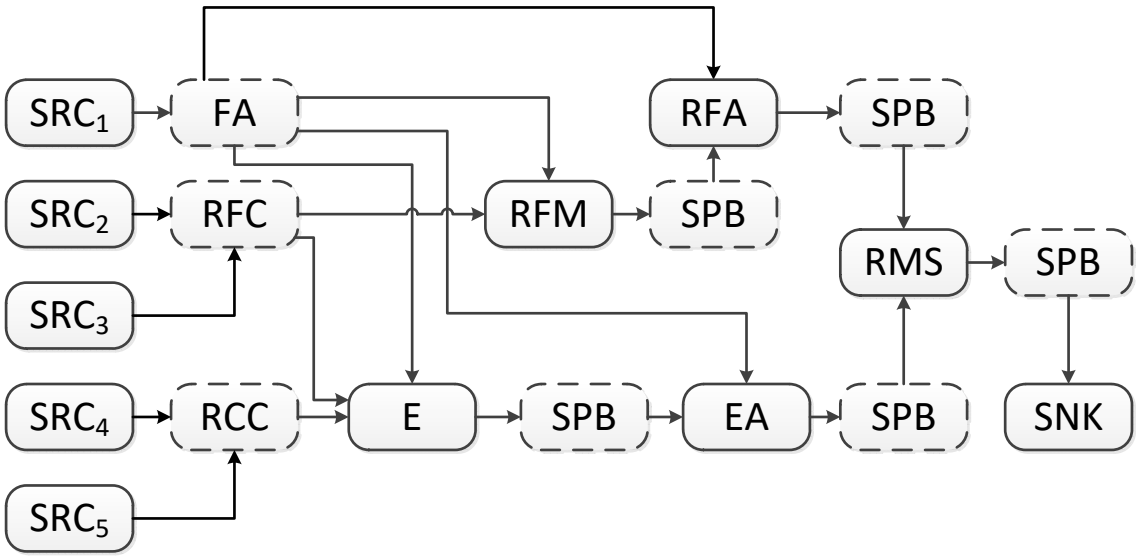

Figure 6.9: Optimized PAFG for EVM measurement.

We first derive a direct PAFG, which represents the implementation of the application graph (Figure 6.8) using pure dataflow semantics. To the direct PAFG, we apply the passivization transformation three times with respect to the actors FA, RFC and RCC. All three of these actors are simply-surrounded, and can be implemented efficiently in passive form.

The resulting optimized PAFG is illustrated in Figure 6.9. We use a minor abuse of notation where non-simple blocks in the PAFG are labeled with the same names as their corresponding actors in the application graph. Blocks labeled as SPB represent simple passive buffers.

Table 6.3 compares the performance of the direct and transformed PAFGs. Through passivization, the throughput is improved by about 31.88%, and the buffer memory requirement (BMR) is reduced by about 25%. We define the BMR of a PAFG G as the total memory requirement for all passive blocks in G.

Table 6.3: Results for the EVM application.

	Throughput (samples/sec)	BMR (MB)
Direct PAFG	7.93×10^5	29.30
Optimized PAFG	1.05×10^6	21.97

6.5.2 Jitter Measurement Application

In this section, we apply PAFG-based modeling and optimization for the jitter measurement system design that was presented in Chapter 3. For details on this system design, including the dataflow model and the constituent actors, we refer the reader to Chapter 3.

An important parameter in the jitter measurement system is the *window size*, which determines the number of samples that are processed in a given dataflow graph iteration. Larger window sizes in general improve the throughput at the expense of a larger BMR (see Chapter 3).

The dataflow graph for this application is illustrated in Figure 6.10. The actors labeled FA2, FA3, FA4 are 2-output, 3-output, and 4-output fork actors, respectively. The graph in Figure 6.10 is based on the dataflow model presented in Chapter 3, except that the graph in Figure 6.10 incorporates fork actors for all inter-actor broadcast operations. In contrast, several of the broadcast operations in the dataflow model of Chapter 3 are achieved by replicating data across multiple output ports of the producing actors ("broadcasting actors"). The use of fork actors, as represented in Figure 6.10, provides a more modular approach since a broadcasting

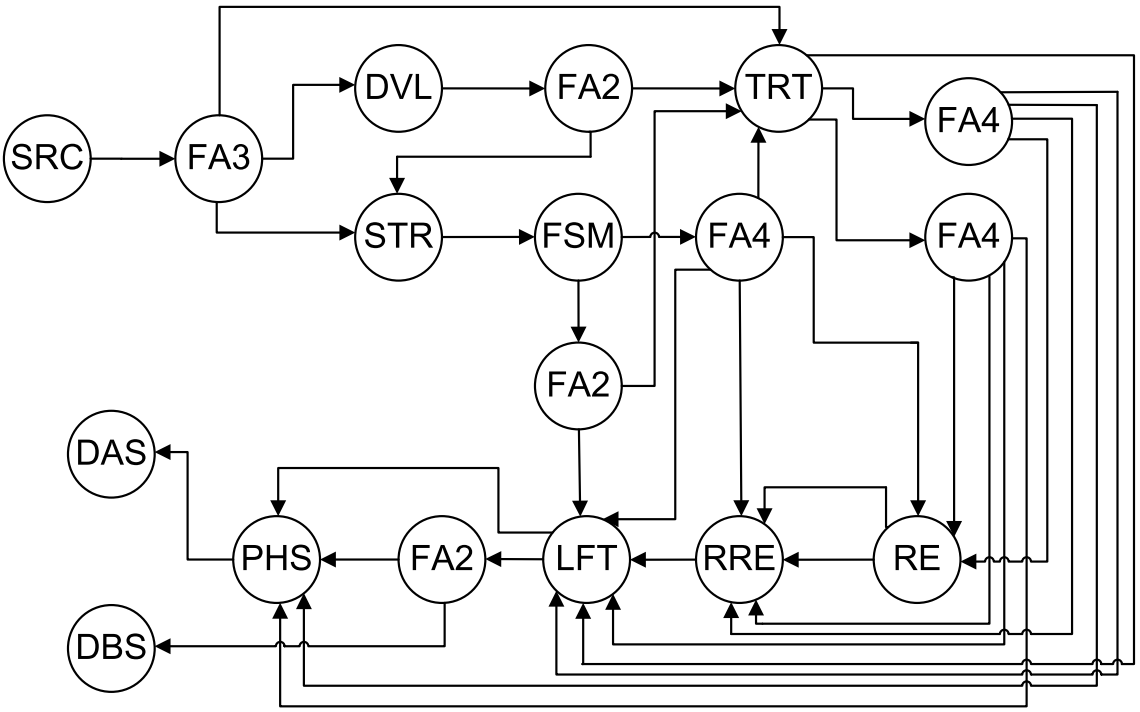

Figure 6.10: Dataflow graph for jitter measurement application.

actor need not have multiple implementations or multiple configurations (with different numbers of output ports) depending on whether its output data is broadcast or how many actors the output data is broadcast to.

Again, we first derive the direct PAFG and then transform this into an optimized PAFG through a sequence of passivization transformations. In this transformation process, we convert the six fork actors in the design, from active to passive buffer form.

The resulting optimized PAFG is illustrated in Figure 6.11. In this figure, the non-simple passive blocks corresponding to the fork actors are denoted $F_1, F_2, \ldots, F_6$.

Table 6.4 shows the improvement measured from the optimized PAFG compared to the direct PAFG for different window sizes. From these results, we see significant improvements delivered by the optimized PAFG in terms of the trade-off

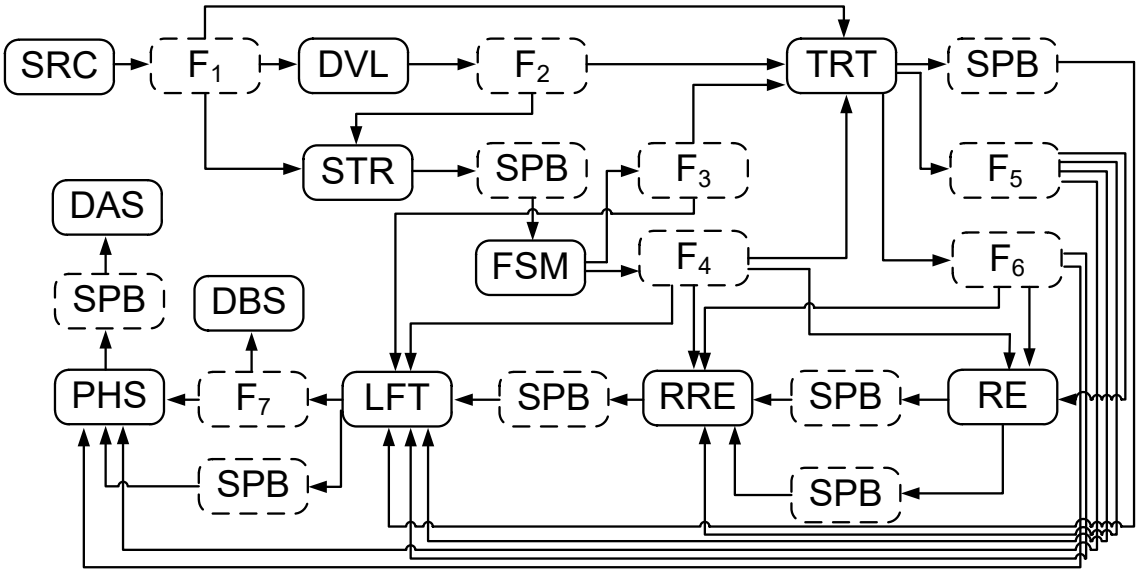

Figure 6.11: Optimized PAFG for jitter measurement application.

between throughput and BMR. For the optimized PAFG, the BMR ranges from 0.38MB to 6.0MB for increasing window sizes, and the throughput ranges from 1.9×10^6 samples/sec to 3.1×10^6 samples/sec.

Table 6.4: Results for the jitter measurement application.

Window size	16,384	32,768	65,536	131,072	262,144
Throughput	11%	7.0%	7.7%	6.5%	7.0%
BMR	60%	60%	60%	60%	60%

6.6 Summary

In this chapter, we have introduced passive-active flowgraphs (PAFGs) as a model of computation that complements dataflow models for design and implementation of signal processing systems. PAFGs generalize the concept of dataflow edges into multi-input, multi-output components that are called "passive blocks".

PAFGs provide a new approach to integrating designer-specified memory management optimization systematically into the framework of dataflow-based design and implementation. In addition to presenting details of the PAFG model of computation, we have introduced the passivization transformation, which can be used iteratively to derive progressively more efficient PAFGs. We have demonstrated the utility of PAFGs and the passivization transformation on two important deep waveform analysis applications. The optimized implementation of these applications using PAFGs leads to significant throughput improvement and reduction in buffer memory requirements.

Chapter 7: Conclusions and Future Work

In this chapter, we first summarize our work and contributions, as presented in the previous chapters of this thesis. Then we discuss interesting directions for future research.

7.1 Conclusions

In this thesis, we have presented new methods for dataflow-based design and implementation of deep waveform applications on resource-constrained platforms. Throughout the thesis, we have demonstrated our methods concretely in the context of deep waveform analysis for jitter measurement, which finds important applications in many areas, including communication system design. The contributions of the thesis consist of three main parts.

First, we have developed new dataflow-based design methods and windowing techniques for efficient, model-based implementation of deep waveform measurement systems. We have demonstrated that our approach to dataflow modeling together with window-based signal analysis helps to significantly reduce memory requirements and reduce latency compared to conventional "swallow and wallow" analysis. We have also developed novel methods to assess the accuracy of transformations from

swallow and wallow analysis to dataflow analysis that bases its results at any time on a prefix of the samples that have already been processed. These methods help to ensure that transformations maintain sufficient accuracy, which in turn allows system designers to provide more strategic trade-offs between real-time performance and analysis quality.

Second, we have developed novel models and design optimization methods for gapless deep waveform applications, where continuous streams of data must be processed reliably without any gaps. The approaches developed in this part of the thesis involve unified dataflow-based modeling of the interfaces and signal processing functionality of gapless deep waveform analysis. Bottleneck actors in the resulting dataflow model are then identified and tackled with approximate computing techniques. These techniques are developed and configured carefully so that large performance gains are achieved while keeping reductions in signal processing accuracy to a manageable level. Efficient actor- and graph-level code optimization techniques are also applied to further improve real-time performance. In addition to providing accurate, real-time processing on the experimental platform used in our experiments, the algorithm- and model-based formulation of the contributions in this part promote their general utility in deep waveform analysis, and their retargetability to other platforms.

Third, we have developed a modeling approach that enables new ways of optimizing memory management efficiency in signal processing dataflow graphs. In particular, we have developed passive-active flowgraphs (PAFGs) as a model of computation that provides a useful new intermediate representation for dataflow-

based design flows. The PAFG model generalizes dataflow edges into multi-input, multi-output elements that are referred to as passive blocks. We develop systematic methods for transforming a signal processing dataflow graph into an equivalent PAFG representation. Furthermore, we develop transformation techniques for deriving progressively more efficient PAFG representations for an application. When applying these transformations, the final PAFG that results can be converted into optimized embedded software that realizes the original application in a manner that provides significantly improved efficiency of inter-actor communication. We have demonstrated the utility our new PAFG model and its associated transformation techniques on complex applications for deep waveform analysis.

7.2 Future Work

Various useful directions for future work have been motivated from the developments of this thesis. These can be divided into two major areas — future work on efficient parallelization of deep waveform analysis systems, and future work on the PAFG model of computation.

7.2.1 Parallelization of Deep Waveform Analysis Systems

The design optimization methods developed in this thesis for efficient parallel computation have focused on accelerating the parts of the system that are mapped to the GPU in a hybrid CPU-GPU implementation. This has been an effective approach in our jitter measurement case study since all of the computationally

intensive actors in the system are amenable to GPU acceleration.

However, a more general design methodology would support optimizations that strategically map some computationally-intensive actors to the CPU. This way, CPU and GPU resources could cooperate in parallel on time-consuming parts of the required deep waveform analysis.

Additionally, more thorough investigation into vectorization of selected actors would enable more comprehensive design space exploration of trade-offs involving dataflow buffer memory requirements, latency, and throughput. Recent work by Lin et al. has introduced new algorithms for integrated vectorization and CPU-GPU parallelization of synchronous dataflow graphs [43]. We anticipate that this would be a promising starting point for investigation into more comprehensive design space exploration of deep waveform analysis systems.

Extension of the static scheduling techniques proposed in this thesis to just-in-time deployment contexts is another interesting direction for future work.

7.2.2 Future Work on PAFG-Based Design and Implementation

Our work on the new PAFG model of computation has introduced several directions for future work. These include the following.

- Automation of the equivalence checking process between active and passive implementations of the same buffer actor.
- Exploration into the utility of passive interface blocks, corresponding to source and sink actors of a dataflow graph.

- Generalization of the developed PAFG-based optimizations beyond alternating PAFGs.
- Investigation of communication mechanisms in PAFGs that allow relaxing of the adjacent buffer coordination (ABC) restriction.
- Exploration of complementary relationships among buffer merging, deterministic SDF with shared FIFOs (DSSF), and PAFG-based memory management.

Bibliography

[1] A. Kuo, T. Farahmand, N. Ou, S. Tabatabaei, and A. Ivanov. Jitter models and measurement methods for high-speed serial interconnects. In *2004 International Conference on Test*, pages 1295–1302, 2004.

[2] J. G. Proakis. *Digital Communications*. McGraw-Hill, fourth edition, 2001.

[3] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, second edition, 2013. ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).

[4] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–799, May 1995.

[5] Y. Liu, L. Barford, and S. S. Bhattacharyya. Constant-rate clock recovery and jitter measurement on deep memory waveforms using dataflow. In *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*, pages 1590–1595, Pisa, Italy, May 2015.

[6] Y. Liu, L. Barford, and S. S. Bhattacharyya. Jitter measurement on deep waveforms with constant memory. In *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*, pages 1161–1166, Taipei, Taiwan, May 2016.

[7] S. S. Bhattacharyya and E. A. Lee. Memory management for synchronous dataflow programs. Technical Report UCB/ERL M92/128, Electronics Research Laboratory, University of California at Berkeley, 1992.

[8] Y. Liu, L. Barford, and S. S. Bhattacharyya. Generalized graph connections for dataflow modeling of DSP applications. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, Cape Town, South Africa, October 2018. To appear.

[9] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[10] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009. ISBN:1420048015.

[11] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real time DSP. In *Proceedings of the Global Telecommunications Conference*, volume 2, pages 1279–1283, 1989.

[12] N. Bambha, V. Kianzad, M. Khandelia, and S. S. Bhattacharyya. Intermediate representations for design automation of multiprocessor DSP systems. *Journal of Design Automation for Embedded Systems*, 7(4):307–323, November 2002.

[13] S. S. Bhattacharyya and J. Lilius. Model-based representations for dataflow schedules. In *Principles of Modeling*, pages 88–105. Springer, 2018.

[14] 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.

[15] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.

[16] S. Lin, L.-H. Wang, A. Vosoughi, J. R. Cavallaro, M. Juntti, J. Boutellier, O. Silvén, M. Valkama, and S. S. Bhattacharyya. Parameterized sets of dataflow modes and their application to implementation of cognitive radio systems. *Journal of Signal Processing Systems*, 80(1):3–18, July 2015.

[17] K. Desnos, M. Pelcat, J.-F. Nezan, S. S. Bhattacharyya, and S. Aridhi. PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 41–48, Samos, Greece, July 2013.

[18] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, Washington DC, USA, November 2010.

[19] Wajih Dalal and Daniel Rosenthal. Measuring jitter of high speed data channels using undersampling techniques. In *Test Conference, 1998. Proceedings., International*, pages 814–818. IEEE, 1998.

[20] T. Loken, L. Barford, and F. C. Harris. Massively parallel jitter measurement from deep memory digital waveforms. In *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*, pages 1744–1749, 2013.

[21] D. Murray. Using RF recording techniques to resolve interference problems. In *Proceedings of AUTOTESTCON*, pages 1–6, 2013.

[22] Wolfgang Maichen. *Digital Timing Measurement: From Scopes and Probes to Timing and Jitter*, chapter 9.3. Springer, 2006.

[23] John Essick. *Hands-On Introduction to LabVIEW for Scientists and Engineers*. Oxford University Press, 2nd edition, 2012.

[24] Robert B. Angus and Thomas E. Hulbert. *VEE Pro: Practical Graphical Programming*. Springer, 2004.

[25] Luca Zamboni. *Getting Started with Simulink*. Packt Publishing, 2013.

[26] Lee Barford, S. S. Bhattacharyya, and Yanzhou Liu. Data flow algorithms for processors with vector extensions: handling actors with internal state. In *Proceedings of the IEEE Global Conference on Signal and Information Processing*, pages 170–174, Atlanta, Georgia, December 2014.

[27] IEEE standard for transitions, pulses, and related waveforms. *IEEE Std 181-2011 (Revision of IEEE Std 181-2003)*, pages 1 –71, 6 2011.

[28] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Math*, 14(6), November 1966.

[29] C. Shen, L. Wang, I. Cho, S. Kim, S. Won, W. Plishker, and S. S. Bhattacharyya. The DSPCAD lightweight dataflow environment: Introduction to LIDE version 0.1. Technical Report UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://hdl.handle.net/1903/12147.

[30] Alan Miller. *Subset selection in regression*. CRC Press, 2012.

[31] NIST/SEMATECH e-Handbook of Statistical Methods, sec. 1.3.3.26.11.

[32] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[33] Khronos OpenCL Working Group et al. The opencl specification. *version*, 1(29):8, 2008.

[34] K. E. Batcher. Sorting networks and their applications. In *Sorting networks and their applications*, 1968.

[35] Yan Shengen, Zhang Yunquan, Long Guoping, et al. Reduction algorithm optimization based on the opencl. *Journal of Software*, 22(S2):163–171, 2011.

[36] Markus Billeter, Ola Olsson, and Ulf Assarsson. Efficient stream compaction on wide simd many-core architectures. In *Proceedings of the conference on high performance graphics 2009*, pages 159–166. ACM, 2009.

[37] Mark Harris, Shubhabrata Sengupta, and John D Owens. Parallel prefix sum (scan) with cuda. *GPU gems*, 3(39):851–876, 2007.

[38] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, second edition, 2013.

[39] Keysight Technologies. *Keysight U5303A PCIe High-Speed Digitizer with On-Board Processing: Data Sheet*, 2015.

[40] L. Giannone et al. Data acquisition and real-time signal processing of plasma diagnostics on asdex upgrade using LabVIEW RT. *Fusion Engineering and Design*, 85(3–4):303–307, 2010.

[41] J. Heulot, M. Pelcat, J.-F. Nezan, Y. Oliva, S. Aridhi, and S. S. Bhattacharyya. Just-in-time scheduling techniques for multicore signal processing systems. In *Proceedings of the IEEE Global Conference on Signal and Information Processing*, pages 175–179, Atlanta, Georgia, December 2014.

[42] E. A. Lee. Recurrences, iteration, and conditionals in statically scheduled block diagram languages. In *Proceedings of the International Workshop on VLSI Signal Processing*, 1988.

[43] S. Lin, J. Wu, and S. S. Bhattacharyya. Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms. *ACM Transactions on Embedded Computing Systems*, 17(2):50:1–50:25, January 2018.

[44] S. Sengupta, M. Harris, and M. Garland. Efficient parallel scan algorithms for GPUs. Technical Report NVR-2008-003, NVIDIA Corporation, 2008.

[45] G. E. Blelloch. Prefix sums and their applications. Technical Report CMU-CS-90-190, Carnegie Mellon University, 1990.

[46] C. Harris, K. Haines, and L. Staveley-Smith. GPU accelerated radio astronomy signal convolution. *Experimental Astronomy*, 22(1–2):129–141, 2008.

[47] S. Fischaber, R. Woods, and J. McAllister. SoC memory hierarchy derivation from dataflow graphs. In *Proceedings of the IEEE Workshop on Signal Processing Systems*, October 2007.

[48] H. Oh and S. Ha. Fractional rate dataflow model and efficient code synthesis for multimedia applications. *ACM SIGPLAN Notices*, 37, July 2002.

[49] S. Stuijk, M. Geilen, and T. Basten. Exploring tradeoffs in buffer requirements and throughput constraints for synchronous dataflow graphs. In *Proceedings of the Design Automation Conference*, July 2006.

[50] S. S. Bhattacharyya and E. A. Lee. Memory management for synchronous dataflow programs. Technical Report UCB/ERL M92/128, Electronics Research Laboratory, University of California at Berkeley, 1992.

[51] P. K. Murthy and S. S. Bhattacharyya. Buffer merging — a powerful technique for reducing memory requirements of synchronous dataflow specifications. *ACM Transactions on Design Automation of Electronic Systems*, 9(2):212–237, April 2004.

[52] K. Desnos, M. Pelcat, J.-F. Nezan, and Slaheddine Aridhi. Buffer merging technique for minimizing memory footprints of synchronous dataflow specifications. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1111–1115, 2015.

[53] C. Hsu, M. Ko, S. S. Bhattacharyya, S. Ramasubbu, and J. L. Pino. Efficient simulation of critical synchronous dataflow graphs. *ACM Transactions on Design Automation of Electronic Systems*, 12(3):21, 2007. 28 pages.

[54] 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.

[55] S. Tripakis, D. Bui, M. Geilen, B. Rodiers, and E. A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical SDF graphs. *ACM Transactions on Embedded Computing Systems*, 12(3), 2013.

[56] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008.

[57] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya. The DSPCAD framework for modeling and synthesis of signal processing systems. In S. Ha and J. Teich, editors, *Handbook of Hardware/Software Codesign*, pages 1–35. Springer, 2017.

[58] R. Schmogrow et al. Error vector magnitude as a performance measure for advanced modulation formats. *IEEE Photonics Technology Letters*, 24(1):61–63, 2012.