Liu *et al. EURASIP Journal on Advances in Signal Processing* (2019) 2019:19
https://doi.org/10.1186/s13634-019-0615-7

EURASIP Journal on Advances
in Signal Processing

RESEARCH **Open Access**

Optimized implementation of digital signal processing applications with gapless data acquisition

Yanzhou Liu[1*], Lee Barford[2] and Shuvra S. Bhattacharyya[1,3]

Abstract

This paper presents novel models and design optimization methods for gapless deep waveform applications, where continuous streams of data must be processed reliably without dropping any samples. The approaches developed in this paper involve unified dataflow-based modeling of the interfaces and signal processing functionality of gapless deep waveform analysis. Bottleneck actors (computational modules) 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 promotes their general utility in deep waveform analysis and their retargetability to other platforms.

Keywords: Dataflow, Data acquisition, Graphics processing unit (GPU), Jitter measurement, Signal processing systems

1 Introduction

This paper 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 paper is to provide structured models and systematic methods for addressing this challenge.

We design gapless DSP applications using 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 [1].

In this paper, we discuss the models and techniques in the context of a specific gapless DSP application, which is real-time jitter measurement of deep waveforms 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*. However, the core approaches developed in our paper are not specific to this application and can be adapted to other relevant applications. 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 [2].

*Correspondence: yzliu@umd.edu
[1]Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, USA
Full list of author information is available at the end of the article

An important aspect of the techniques that we develop in this paper is the model-based integration of data acquisition (DAQ) devices into dataflow-based design processes. DAQ boards are widely used in numerous signal processing application areas, such as astronomy, environmental monitoring, biomedical instrumentation, and satellite communication (e.g., see [3, 4]).

In the deep jitter measurement system, 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 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 paper 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 paper is organized as follows. Section 2 provides introductions to dataflow modeling and the DSPCAD Lightweight Dataflow Environment (LIDE), which is a methodology and a tool, respectively, that are applied extensively in this work. In Section 3, we review related work on jitter measurement systems and dataflow graph implementation. Section 4 presents dataflow graph design approaches for efficient implementation of gapless DSP applications, using deep jitter measurement as a concrete case study. In Section 5, 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 6. Section 7 summarizes the contributions of the paper and outlines directions for future work.

2 Background

In this section, we discuss background on dataflow graph modeling that the developments of this paper depend on. We also provide background on the DSPCAD Lightweight Dataflow Environment (LIDE), which is a software tool for dataflow-based design and implementation that we apply in this work.

Dataflow is a form of model-based design that is widely used in the design and implementation of DSP applications [2]. As described in Section 1, a dataflow graph is a directed graph in which vertices are called actors and represent signal processing hardware/software components and edges specify FIFO communication of data between actors. In the form of dataflow that we apply in the work, individual actors can have arbitrary complexity. Examples of functions that are performed by dataflow actors include digital filtering, fast Fourier transform (FFT) computation, and matrix operations. Conceptually, data values are encapsulated in objects called *tokens* as they pass across dataflow graph edges.

In dataflow, execution of actors is decomposed into discrete units, which are called *firings* [1]. In a given firing, an actor consumes and produces data from its input FIFOs and onto its output FIFOs, respectively. For a given firing f and output FIFO K, the number of tokens produced onto K during f is referred to as the *production rate* associated with f and K. Similarly, we can define the *consumption rate* associated with a firing and an input FIFO. Production and consumption rates are referred to collectively as *dataflow rates*.

If for a given actor the dataflow rate on each FIFO connected to the actor is constant, then we refer to the actor as a *synchronous dataflow* (*SDF*) actor. A dataflow graph in which all actors are SDF actors is called an *SDF graph* [5].

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 [6]. If the decisions are made after compile time but prior to graph execution, the schedule is said to be a *just-in-time* schedule [7]. 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 paper, 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 paper are sufficiently predictable to enable this form of static scheduling.

The DSPCAD Lightweight Dataflow Environment (LIDE) is a software tool for dataflow-based design and implementation of signal processing systems [8, 9]. The environment is based on a compact set of application programming interfaces (APIs) for implementing design components as dataflow actors. Dataflow programming in LIDE is based on the core functional dataflow (CFDF) model of computation [10]. Each actor A in a CFDF graph has an associated set of *modes* $\mu(A)$, which can be viewed as alternative computational tasks that correspond to firings of A. Each firing of A has a unique mode associated with it. Each actor mode has constant dataflow rates on all input and output FIFOs, while the dataflow rates can vary across different modes of the same actor.

Each CFDF actor has two associated methods, called the *invoke* method and the *enable* method. The invoke method is used to execute the actor in its current mode, while the enable method is used to determine whether or not there is enough data on the input edges and enough empty space on the output edges to support firing the actor in its current mode. The separation of concerns between enable testing and invoking is an important feature of the CFDF model [10].

The LIDE APIs are formulated in terms of abstract dataflow principles and are independent of any particular programming language. This abstract formulation and the compact nature of the APIs make the core of LIDE easily retargetable to arbitrary languages for DSP implementation and simulation, such as C, C++, CUDA, MATLAB, OpenCL, Verilog, and VHDL. In this work, we use C and OpenCL versions of the LIDE APIs, which are referred to, respectively, as LIDE-C, and LIDE-OCL.

3 Related work

There are two main purposes for deep jitter measurement: (1) to increase the likelihood of capturing rare events that can cause communication errors [11] and (2) to enable estimation of tails in jitter probability distributions, as a replacement for or to improve the accuracy of distribution extrapolation [12]. Implementations of timing jitter measurement are available in instruments such as digital oscilloscopes. However, the computation time and memory requirements increase with waveform depth, and so, it is desirable to seek methods for faster yet still cost-effective jitter computation from deep waveforms.

To address this problem and help accelerate jitter measurement, researchers have introduced parallel algorithms for constant clock period computation. For example, [13] exploits multi-core processors such as Intel central processing units (CPUs) together with their *streaming single instruction multiple data extensions* (*SSE*) [14] instruction sets to enable fast and accurate jitter measurement. However, this design suffers from large memory requirements and high latency due to its "swallow and wallow" characteristic whereby the computation is started only after all input data has arrived and has been stored in memory. This limits the amount of signal data that can be measured and results in high response time for engineers to start seeing measurement results.

Another jitter measurement algorithm was demonstrated in [15] that significantly improves measurement response time 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 reformulation of jitter measurement eliminates the swallow and wallow characteristic and provides improved speed. However, a memory requirement limitation still remains: the memory required (as in the approach of [13]) 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 when memory resources are limited.

A preliminary version of this paper was presented in [16]. In this prior work, we presented a novel deep jitter measurement system that loads and processes constant-frequency signal data from an input file. The contribution of the prior work was focused on streamlining memory requirements and efficiently trading off accuracy and performance. The contribution improved the algorithm of [15] 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 paper, we go beyond the preliminary version in the following ways. First, we incorporate methods to process input from a DAQ device under the constraint of gapless processing. 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.

4 System design

In this section, we discuss our methods for dataflow graph design of gapless deep waveform analysis applications. As described in Section 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 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.

4.1 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 5.1.

4.2 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 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 Fig. 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 4.2), 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 paper as the *targeted DAQ device* (*TDD*). The implementations of the DAS and DAT actors are developed using LIDE-OCL (see Section 2).

4.2.1 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 4.1. 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

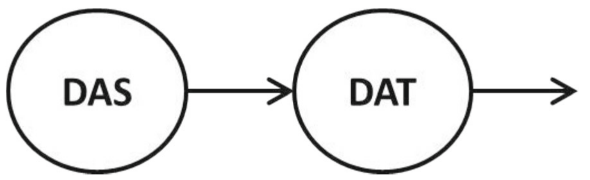

Fig. 1 Subgraph for acquiring data

identical. Self-loop edges are an established method for modeling actor state in signal processing dataflow graphs (e.g., see [17]).

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.

4.2.2 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 caries out the core signal processing functionality for deep jitter measurement. The overall dataflow graph for deep jitter measurement, including the subgraph of Fig. 1 and the subgraph for core signal processing, is presented in Section 4.3. 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.

4.3 Dataflow graph for deep jitter measurement

Figure 2 illustrates the overall dataflow graph for our deep jitter measurement system. Here, as described in Section 4.2, 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 Fig. 2 are summarized in Table 1. For further background on computations involved in jitter measurement, we refer the reader to [13, 15, 16].

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 4.2.1 for the DAS actor). Since CFDF modes must have constant production and consumption rates on all actor ports (see Section 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 Fig. 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 4.1 that W_s, which appears in the annotations associated with edge (DAS, DAT), represents the window size.

The token types associated with the edges in Fig. 2 are summarized as follows. The edge (DAS, DAT) has `long` type. The edges (GCC, SKC) and (GCT, SKT) have `double` type. All of the other edges in Fig. 2 have *OpenCL memory object* token type. A memory object in OpenCL is

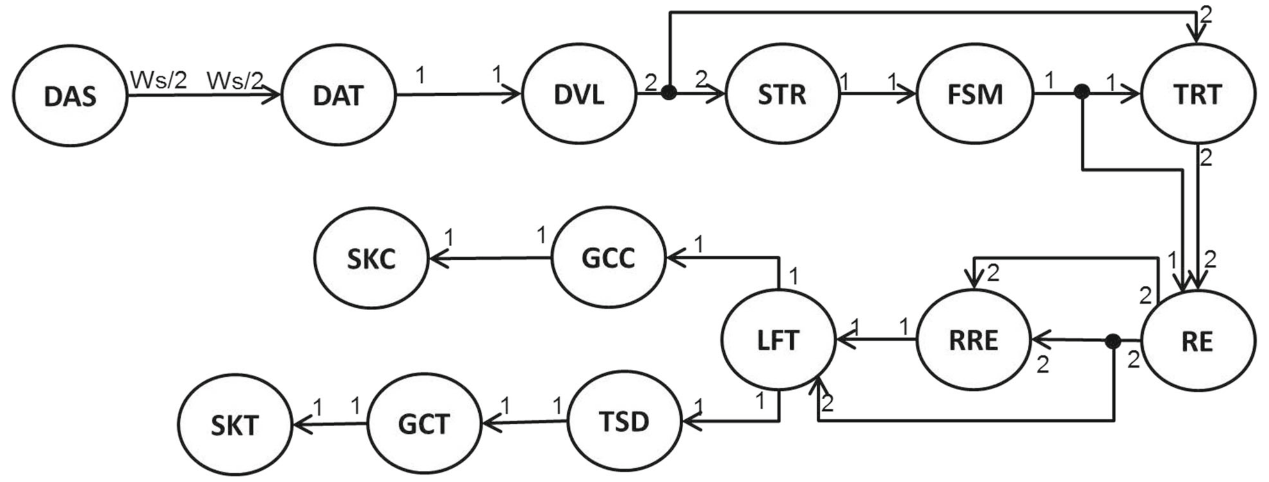

Fig. 2 Dataflow graph for deep jitter measurement system

Table 1 Actors in the dataflow graph of Fig. 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.

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.

$$\begin{array}{l}\text{DAT DVL STR FSM TRT}\\ \text{RE RRE LFT TSD GCC GCT}\\ \text{SKC SKT}\end{array} \tag{1}$$

Acquisition of a new frame of data by the TDD can then proceed concurrently with execution of the subschedule in Eq. 1. The subschedule of Eq. 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 (Eq. 1) completes and there is no input data on the (DAS, DAT) edge that is available to trigger the next subschedule iteration.

5 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 4 as a starting point, we improve the design by applying a sequence of optimizations. These optimization techniques are described in Section 5.1 through Section 5.3. Experimental results from applying these optimization are then presented in Section 6.

5.1 Window size optimization

In this section, we discuss optimized, dynamic configuration of the the window size parameter W_s, which was introduced in Section 4.1. In our deep jitter measurement system, the window size, along with sorting-related parameters (discussed in Section 5.2) that are directly influenced by W_s, has 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 [16]. 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 $C_{\text{trt\_num}}$, 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 $C_{\text{trt\_num}} = 32$ transitions per frame. The value of $C_{\text{trt\_num}}$ 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.

5.2 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 [18] 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

$$S_{\mathrm{DVL}} = \mathrm{power}(k_{\mathrm{DVL}} \times \mathrm{ceil}(W_s/N_{\mathrm{trans}})), \tag{2}$$

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 $k_{\mathrm{DVL}} = 4$, $W_s = 65,536$, and $N_{\mathrm{trans}} = 135$, then:

$$\begin{aligned} S_{\mathrm{DVL}} &= \mathrm{power}(4 \times \mathrm{ceil}(65536/135)) \\ &= \mathrm{power}(4 \times 486) = 2^{11} = 2048. \end{aligned} \tag{3}$$

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 25th 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 $N_{\mathrm{trans}} > C_{\mathrm{RE}}$, 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 $k_{\mathrm{DVL}} = 4$ and $C_{\mathrm{RE}} = 1,024$. Through experimentation, we have determined these values to provide improvements in sorting efficiency without significantly degrading jitter measurement accuracy.

5.3 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 [16]. 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 5.2 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 [19].

For the sorting operation within the RE actor, an efficient value for the vectorization degree is S_{RE}. However, as discussed in Section 5.2, 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 [20–22]) for further performance improvement.

6 Results and discussion

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

6.1 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 5.2. Figure 3 shows the throughput speedup measured for the DVL actor as the ratio R_{sort} of data used for sorting is varied. Different colors in the figure illustrate different window size configurations. For example, when $R_{\mathrm{sort}} = 0.25$ (3 out of 4 samples are ignored), a speedup of 4.63 is obtained when the window size is $W_s = 2^{20}$. The range of speedup values represented in Fig. 3 is from 2.13 (when $R = 0.5$) to 822.57 (when $R = 0.00012$) when the window size is 2^{20}. The trends of throughput speedup versus R_{sort} are similar for all of the window size configurations evaluated in this experiment.

Figure 4 shows the boxplots for the relative error of results produced by the DVL actor for varying values of R_{sort}. Figure 4a shows the relative error of the high voltage threshold, and Fig. 4b gives corresponding results for the low voltage threshold. Figure 4c and Fig. 4d show results on the recovered clock period and TIE standard deviation, respectively.

There are outliers when R_{sort} is small. The median values of the relative errors in Fig. 4 decrease nearly monotonically with increasing R_{sort}. In Fig. 4a, the median values of the relative error for the high voltage threshold do not vary in a strictly monotonic manner when R_{sort} is very small. For higher values of R_{sort}, we see a lower occurrence of outliers. We anticipate that the non-monotonicity effects in these results arise from the randomized selection of data for sorting.

The input data set for this experiment is as described in [16]. The results in Fig. 4 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 summarizes experimental results on the throughput speedup and relative error for different values of R_{sort} in the RE actor. Figure 5a demonstrates the relative error of results for different values of R_{sort}, and Fig. 5b shows the throughput speedup for different values of R_{sort}.

The data in Fig. 5 exhibits the same general trends observed in Fig. 3 and Fig. 4—significant speedups

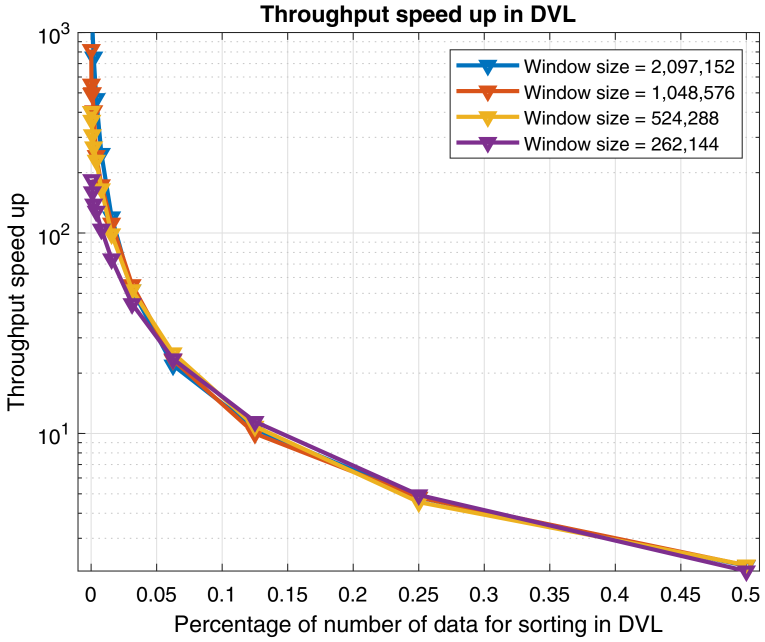

Fig. 3 Throughput speedup for the DVL actor for varying values of R_{sort}

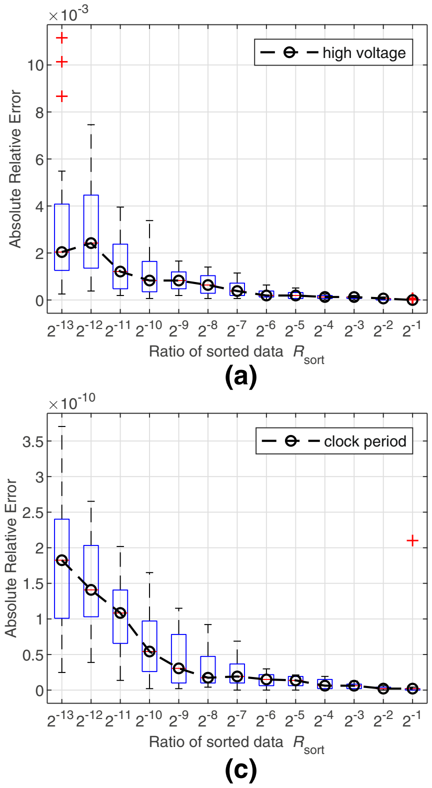

Fig. 4 a–**d** Relative error of results produced by the DVL actor for varying values of R_{sort}

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.

6.2 Optimization of reduction and prefix sum operations

Table 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 5.3. The throughput values listed in the third and fourth

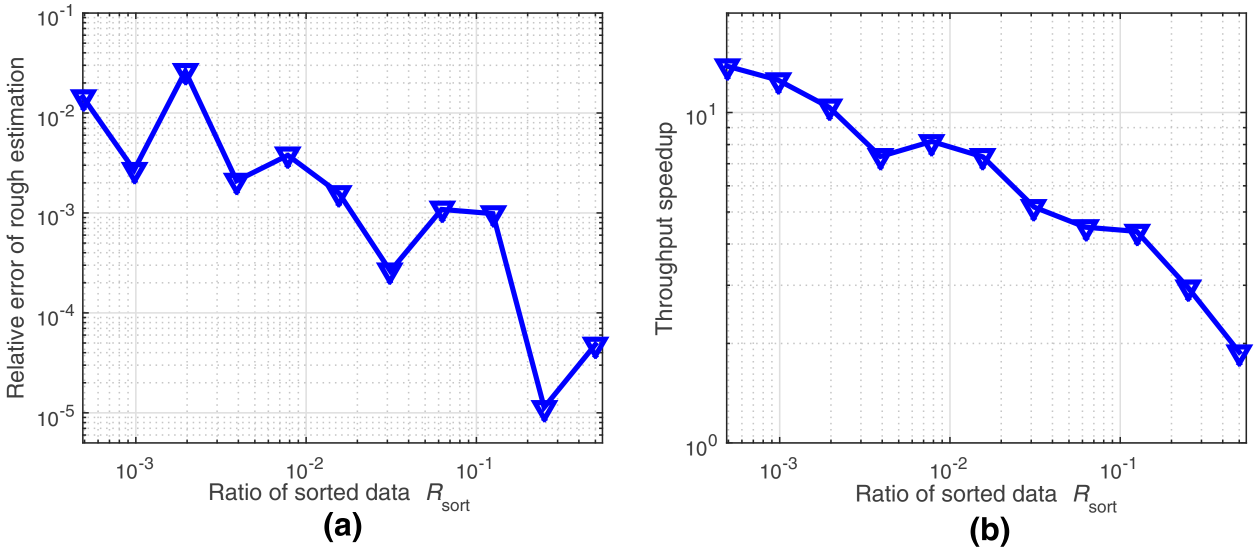

Fig. 5 a, **b** Relative error and throughput speedup for varying values of R_{sort} in the RE actor

Table 2 Throughput speedup for TRT, RRE, and LFT actors

Actor	Window size (samples)	Baseline throughput	Optimized throughput	Speed up
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

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 previous work [16], all three of these actors exhibit over 10X speedup. Unlike the optimizations related to manipulating R_{sort}, the optimizations examined in Table 2 do not affect signal processing accuracy.

6.3 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. 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 rate of 125M SPS. From our experiments involving system throughput (presented in Section 6.5), 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 $\{W_{\min}, W_{\max}\}$.

6.4 Overhead analysis for dynamic adaptation

As described in Section 5.1 and Section 5.2, 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 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 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%

6.5 System throughput

Figure 6 shows the measured throughput of the jitter measurement system for different values of the window size W_s. These results are shown for a set of representative input signal frequencies varying from 300 to 893 kHz. There is no data point when the frequency is 300 kHz and window size is 8,192. This is because if the window size is set to 8,192, the system dynamically doubles the window size to (8,192 × 2) due to insufficient numbers of intra-window transitions with the original window size setting. The implementation is tested with 10 different window sizes from 2^{13} to 2^{22} ($W_{\max}$), with each value of W_s corresponding 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 7 shows the system throughput for different signal frequencies when the window size is varied from 2^{17} to $W_s = W_{\max}$.

The results show relatively small variation in throughput for different frequencies. More specifically, the relative difference between different levels of throughput is less than 3%.

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.

7 Conclusions

In this paper, we have developed a novel framework for addressing in an integrated manner a number of

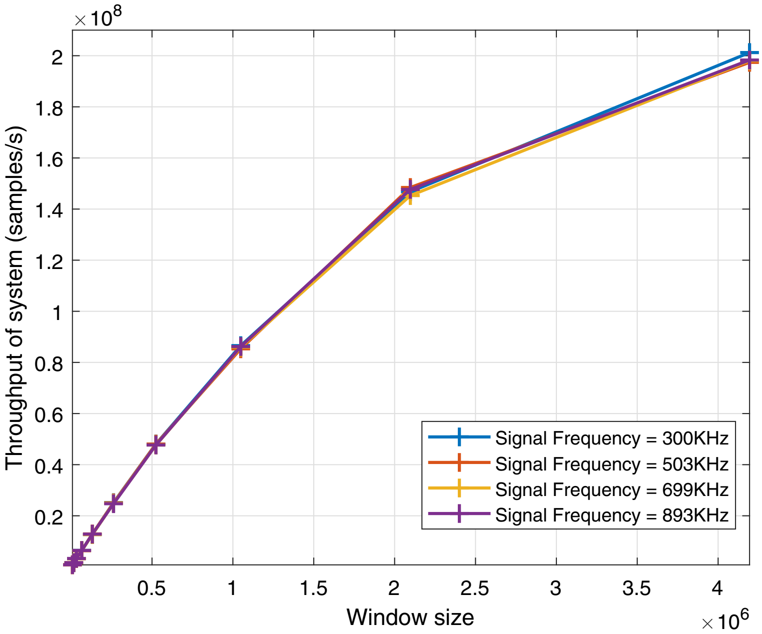

Fig. 6 System throughput versus window size

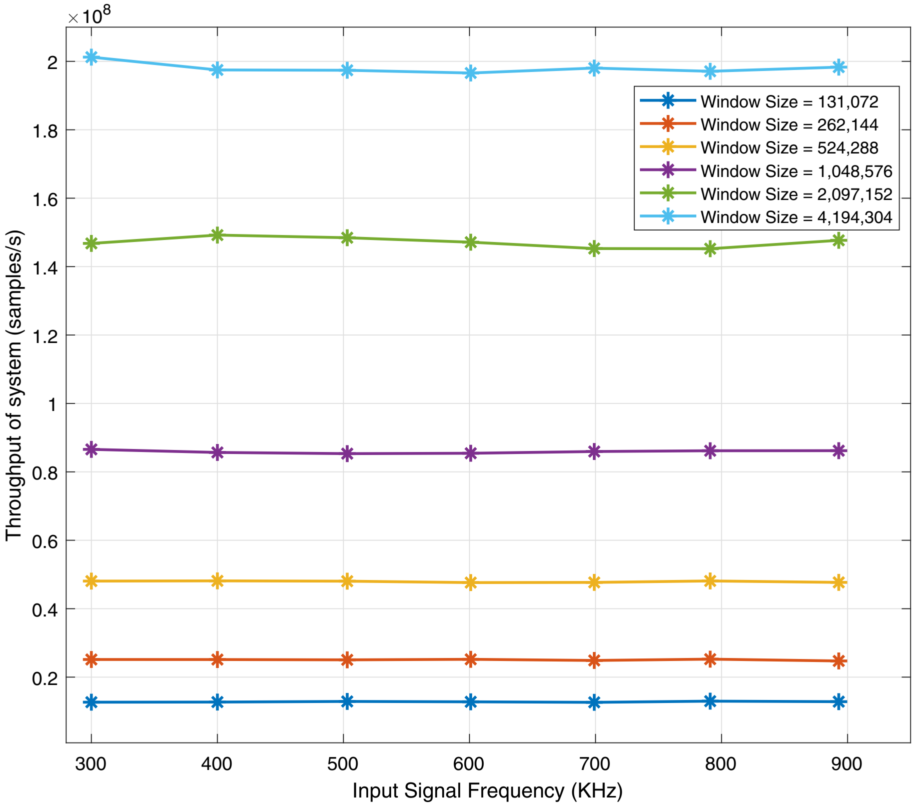

Fig. 7 System throughput versus input signal frequency

important challenges in the design and implementation of gapless signal processing systems. These challenges include the requirement for processing unbounded data streams without input buffer overflow, the need for efficient methods to trade-off signal processing accuracy and throughput subject to the constraint of gapless processing, and iterative platform-based optimization of dataflow actor implementations to maximize system throughput. The contributions of this paper also include systematic integration of data acquisition devices into dataflow-based design processes for signal processing systems. For concreteness, we have demonstrated the proposed design and implementation framework throughout the paper in the context of a specific gapless signal processing application, which is an application involving jitter measurement of deep waveforms.

In the development of the proposed deep jitter measurement system, we have applied gapless processing of input data acquired from a data acquisition (DAQ) device. Techniques were incorporated in the system to adaptively optimize window size configurations and the ratio of the selected data for sorting. These optimizations are important to improve system throughput and allow the system to keep up with high input sample rates.

In this paper, we have focused mainly on static scheduling techniques to execute dataflow graphs for gapless DSP applications. If scheduling decisions are made after compile time but prior to graph execution, the schedule is said to be a *just-in-time* schedule [7]. Extension of the static scheduling techniques proposed in this paper to just-in-time deployment contexts is an interesting direction for future work.

Abbreviations

API: Application programming interface; CFDF: Core functional dataflow; CPU: Central processing unit; DAQ: Data acquisition; DSP: Digital signal processing; FFT: Fast fourier transform; FIFO: First-in, first-out; GPU: Graphics processing unit; LIDE: The DSPCAD Lightweight Dataflow Environment; SMSS: Single-mode steady state; SPS: Samples per second; SSE: Streaming single instruction multiple data extensions; TDD: Targeted DAQ device

Acknowledgements

The authors appreciate the constructive feedback provided by the anonymous reviewers. Their comments have helped to improve the paper.

Funding

This work was sponsored in part by the U.S. National Science Foundation and Keysight Technologies.

Availability of data and materials

The data for experiments is described in [16].

Authors' contributions

YL is the primary author of the paper. She developed and conducted the experiments and wrote the initial draft of the paper. LB and SSB have supervised the research work and acted as research advisors. They provided guidance and feedback throughout the research work and throughout the process of revising the initial draft. All authors have reviewed and approved the final manuscript.

Competing interests

The authors declare that they have no competing interests.

Publisher's Note

Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Author details

[1]Department of Electrical and Computer Engineering and Institute for Advanced Computer Studies, University of Maryland, College Park, MD, USA. [2]Keysight Laboratories, Keysight Technologies, Inc., Reno, USA. [3]Laboratory of Pervasive Computing, Tampere University, Tampere, Finland.

Received: 29 November 2018 Accepted: 21 February 2019
Published online: 15 March 2019

References

1. E. A. Lee, T. M. Parks, Dataflow process networks. Proc. IEEE. **83**(5), 773–799 (1995)
2. S. S. Bhattacharyya, E. Deprettere, R. Leupers, J. Takala, *Handbook of Signal Processing Systems*, 2nd edn. (Springer, New York, 2013)
3. Keysight Technologies, *Keysight U5303A PCIe High-Speed Digitizer with On-Board Processing: Data Sheet*. (Keysight Technologies, 2015)
4. L. Giannone, et al., Data acquisition and real-time signal processing of plasma diagnostics on ASDEX upgrade using LabVIEW RT. Fusion Eng. Des. **85**(3–4), 303–307 (2010)
5. E. A. Lee, D. G. Messerschmitt, Synchronous dataflow. Proc. IEEE. **75**(9), 1235–1245 (1987)
6. E. A. Lee, S. Ha, in *Scheduling strategies for multiprocessor real time DSP*. Proceedings of the Global Telecommunications Conference, vol. 2, (Dallas, 1989), pp. 1279–1283
7. J. Heulot, M. Pelcat, J. Nezan, Y. Oliva, S. Aridhi, S. S. Bhattacharyya, in *Proceedings of the IEEE Global Conference on Signal and Information Processing*. Just-in-time scheduling techniques for multicore signal processing systems, (Atlanta, 2014), pp. 175–179
8. C. Shen, W. Plishker, H. Wu, S. S. Bhattacharyya, in *A lightweight dataflow approach for design and implementation of SDR systems*. Proceedings of the Wireless Innovation Conference and Product Exposition, (Washington, DC, 2010), pp. 640–645
9. C. Shen, L. Wang, I. Cho, S. Kim, S. Won, W. Plishker, 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
10. W. Plishker, N. Sane, M. Kiemb, K. Anand, S. S. Bhattacharyya, in *Proceedings of the International Symposium on Rapid System Prototyping*. Functional DIF for rapid prototyping, (Monterey, 2008), pp. 17–23
11. D. Murray, in *Using RF recording techniques to resolve interference problems*. Proceedings of AUTOTESTCON, (Schaumburg, 2013), pp. 1–6
12. W. Maichen, *Digital Timing Measurement: From Scopes and Probes to Timing and Jitter*. (Springer, USA, 2006). Chap. 9.3
13. T. Loken, L. Barford, F. C. Harris, in *Massively parallel jitter measurement from deep memory digital waveforms*. Proceedings of the IEEE International Instrumentation and Measurement Technology Conference, (Minneapolis, 2013), pp. 1744–1749
14. Intel Corporation, *Intel 64 and IA-32 Architectures Software Developer's Manual — Volume 3 (3A, 3B, 3C & 3D): System Programming Guide*. (Intel Corporation, 2016)
15. Y. Liu, L. Barford, S. S. Bhattacharyya, in *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*. Constant-rate clock recovery and jitter measurement on deep memory waveforms using dataflow, (Pisa, 2015), pp. 1590–1595
16. Y. Liu, L. Barford, S. S. Bhattacharyya, in *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*. Jitter measurement on deep waveforms with constant memory, (Taipei, 2016), pp. 1161–1166
17. E. A. Lee, in *Recurrences, iteration, and conditionals in statically scheduled block diagram languages*. Proceedings of the International Workshop on VLSI Signal Processing, (1988)
18. K. E. Batcher, in *Proceedings of the AFIPS Spring Joint Computer Conference*. Sorting Networks and Their Applications (Atlantic City, 1968), pp. 307–314
19. S. Lin, J. Wu, S. S. Bhattacharyya, Memory-constrained vectorization and scheduling of dataflow graphs for hybrid CPU-GPU platforms. ACM Trans. Embed. Comput. Syst. **17**(2), 50–15025 (2018)

20. S. Sengupta, M. Harris, M. Garland, Efficient parallel scan algorithms for GPUs. Technical Report NVR-2008-003 (2008). NVIDIA Corporation
21. G. E. Blelloch, Prefix sums and their applications. Technical Report CMU-CS-90-190 (1990). School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890
22. C. Harris, K. Haines, L. Staveley-Smith, GPU accelerated radio astronomy signal convolution. Exp. Astron. **22**(1–2), 129–141 (2008)