Journal of Signal Processing Systems, 66(3):285-301, 2012.
DOI: DOI 10.1007/s11265-011-0599-5.

Mapping Parameterized Cyclo-static Dataflow Graphs onto Configurable Hardware

Hojin Kee · Chung-Ching Shen · Shuvra S. Bhattacharyya · Ian Wong · Yong Rao · Jacob Kornerup

Abstract In recent years, parameterized dataflow has evolved as a useful framework for modeling synchronous and cyclo-static graphs in which arbitrary parameters can be changed dynamically. Parameterized dataflow has proven to have significant expressive power for managing dynamics of DSP applications in important ways. However, efficient hardware synthesis techniques for parameterized datafow representations are lacking. This paper addresses this void; specifically, the paper investigates efficient field programmable gate array (FPGA)-based implementation of parameterized cyclo-static dataflow (PCSDF) graphs. We develop a scheduling technique for throughput-constrained minimization of dataflow buffering requirements when mapping PCSDF representations of DSP applications onto FPGAs. The proposed scheduling technique is integrated with an existing formal schedule model, called the generalized schedule tree, to reduce schedule cost. To demonstrate our new, hardware-oriented PCSDF scheduling technique, we have designed a real-time base station emulator prototype based on a subset of *long-term evolution* (*LTE*), which is a key cellular standard.

Hojin Kee
National Instruments Corp., Austin, TX 78759, USA
E-mail: hojin.kee@ni.com

Chung-Ching Shen
Department of ECE and UMIACS, University of Maryland, College Park, MD20742, USA
E-mail: ccshen@umd.edu

Shuvra S. Bhattacharyya
Department of ECE and UMIACS, University of Maryland, College Park, MD20742, USA
E-mail: ssb@umd.edu

Ian Wong
National Instruments Corp., Austin, TX 78759, USA
E-mail: ian.wong@ni.com

Yong Rao
National Instruments Corp., Austin, TX 78759, USA
E-mail: Yong.Rao@ni.com

Jacob Kornerup
National Instruments Corp., Austin, TX 78759, USA
E-mail: jacob.kornerup@ni.com

Keywords Dataflow modeling · Scheduling · FPGA implementation · 4G Communication systems · Parameterized dataflow

1 Introduction

Synchronous dataflow (SDF) [1] has been used widely as an efficient model of computation (MOC) to analyze performance and resource requirements when implementing DSP algorithms on various kinds of target architectures (e.g., see [2], [3], and [4]). The SDF model has been incorporated in many commercial tools for DSP system design, such as ADS from Agilent, LabVIEW from National Instruments, Signal Processing Designer from CoWare, and System Studio from Synopsys. In SDF semantics, DSP applications are modeled by directed graphs in which vertices (*actors*) correspond to computational blocks, and edges represent the passage of data between blocks. SDF imposes the restriction that the number of data values (tokens) that is produced on each output edge is constant per actor execution (*firing*), and similarly, the number of tokens consumed per firing is constant for each actor/input-edge pair. Thus, SDF does not accommodate actors that can have dynamically varying token production and consumption rates. Such "dynamic dataflow" actors are employed in many modern DSP applications, including the LTE physical layer, and therefore, when developing such applications, we must explore models of computation that are more general than pure SDF.

Cyclo-static dataflow (CSDF) is a generalization of synchronous dataflow in which production and consumption rates are allowed to vary dynamically as long as the variates follow periodic patterns that are fully predictable at compile time [5]. Although CSDF production and consumption rates can vary at run-time, CSDF is typically not viewed as a dynamic dataflow model due to the predictability of the run-time variations.

Parameterized cyclo-static dataflow (*PCSDF*) further extends expressive power by allowing dynamic changes in production and consumption rates that are formulated in terms of changes to parameters of *parameterized CSDF graphs* (*PCSDF graphs*) [6]. A PCSDF graph can be viewed as a parameterized family of graphs such that each instance in the family (i.e., each specific setting of the parameters) corresponds to a CSDF graph. PCSDF significantly improves upon the expressive power of CSDF while providing a framework in which many CSDF analysis techniques can be naturally adapted into parameterized versions [7]. Thus, parameterized dataflow modeling approaches allow for dynamic capabilities without excessively compromising the key properties of existing static dataflow models (e.g., SDF and CSDF) — compile-time predictability and potential for rigorous optimizations. For example, techniques for constructing efficient parameterized looped schedules have been developed for PSDF graphs [6]. These scheduling techniques can provide for efficient simulation or software synthesis from PSDF specifications.

When describing a DSP application with a PCSDF graph, functional blocks and storage space for transferring data between adjacent blocks are modeled as graph vertices (*actors*) and edges, respectively. When mapping dataflow graph edges into storage locations, care must be taken to make effective use of limited storage locations (e.g., on-chip memory in programmable digital signal processors, and block RAM and distributed memory in FPGAs). However, reducing the storage space for transferring data between actors may result in decreased throughput due to less frequent firing of actors to prevent buffer overflow — as buffers become smaller, the frequency and dura-

tion for such overflow-avoiding idle time generally increases, which leads to decreased throughput. The limited amounts of storage available in DSP implementation targets, and the importance of meeting real-time performance constraints motivate the goal of throughput-constrained buffer minimization for PCSDF graphs. In this paper, we study this problem in the context of FPGA-based implementation.

2 Related Work

Several approaches are known to date for exploring trade-offs between throughput and buffer memory requirements in dataflow graphs. Traditionally, throughput analysis for SDF graphs is performed by solving an instance of the maximum cycle mean problem (e.g., see [8], and [9] after converting the input SDF graph into an equivalent homogeneous SDF (HSDF) graph [1]. Throughput analysis based on SDF-to-HSDF conversion suffers from high worst case complexity because neither the time nor space required to perform this conversion is polynomially bounded (e.g., see [10]).

Ghamarian et al. [11] have developed a method for SDF throughput analysis that avoids conversion to an HSDF graph, and uses state space exploration techniques — in terms of the buffer state — instead. In general, executions of actors change the buffer state by removing (consuming) tokens from input edges of the actors that fire, and inserting (producing) tokens onto output edges. Ghamarian exploits the property that when SDF graphs execute in a purely data driven ("self-timed") manner under bounded memory space, the state space is also bounded, and execution eventually settles into a periodic pattern (periodic steady state or *PSS*). In Ghamarian's method for throughput analysis, only selected states need to be stored when detecting the PSS of execution, and through Ghamarian's careful pruning technique for state storage, significant improvements can be achieved in the efficiency of performance analysis. However, the technique requires simulation of the overall schedule, and the worst case complexity is linear in the length (number of firings in) the given periodic schedule, which, as described above, is not polynomially bounded in the size of the input SDF graph.

Stuijk [12] develops a systematic approach for exploring throughput and storage trade-offs for SDF graphs. This approach applies methods developed by Geilen [13] for determining minimum storage requirements based on state-space analysis of buffer memory requirements. Stuijk's approach operates by first finding a minimal storage distribution, and then recursively increasing the storage space for each edge that has a storage dependency. This results in a family of buffer distribution-throughput pairs as a representation of Pareto solutions for the graph.

These approaches are generally based on the *self-timed execution model*, which means that each actor is fired as soon as all of its input edges have sufficient data. When actors execute and communicate on dedicated resources (so that resource contention is not an issue), this type of execution generally enhances throughput by facilitating the exploitation of parallel processing capabilities on the target hardware. Since each actor can be executed only after its input edges have certain numbers of tokens in self-timed execution, each edge has an associated bound on the required buffer memory (allocated buffer space) for avoiding deadlock [14]. In FPGA design, patterns of receiving and producing data in a functional intellectual property (IP) block are often known from associated data sheets. Even before input edges of a functional block have enough tokens to satisfy a full execution of the block, the block can typically be fired if we can

guarantee that enough tokens will be delivered to its input edges during the overall execution of the block. A schedule that takes into account such possibility for operating on partial input sets can be expected to require less buffer space in general compared to conventional self-timed execution.

Horstmannshoff et al. [15], [4] developed an SDF scheduling method for complex register-transfer level building blocks. Based on timing patterns associated with token production and consumption in each actor, this method constructs a retiming graph to generate a *stall signal* for each SDF actor such that buffer cost is minimized. This minimum-area retiming approach is applied to determine optimum actor activation times in polynomial time complexity. This work provides a motivation for us to develop a framework for generating a buffer-efficient hardware schedules for the more general family of PCSDF graphs.

In addition to minimizing the amount of buffer space required for data communication channels, minimizing the memory required for storing the generated schedules is another important implementation issue in scheduling PCSDF graphs. In this paper, we address these challenges, and develop new scheduling techniques for throughput-constrained minimization of data channel buffer requirements when mapping PCSDF representations of DSP applications onto FPGAs. In our schedule construction framework, a previously developed data structure called the *generalized schedule tree* (*GST*) is integrated in a novel way to reduce PCSDF schedule size, which in turn reduces the storage cost associated with schedule control. In this work, we impose the restriction that the given PCSDF graph is in the form of a tree-structured, directed acyclic graph.

A preliminary version of part of this work was presented in [16]. While the previous work [16] focused on developing the FPGA-based framework for modeling parametrized SDF graph, we presented a scheduling technique for throughput-constrained minimization of dataflow buffering requirements in PCSDF representations of DSP applications in this paper. This paper goes beyond the developments of [16] in that it generally requires less buffer space in data communication channels.

3 Background

3.1 Cyclo-static dataflow(CSDF)

As described earlier, cyclo-static dataflow (CSDF) is an extension of SDF that allows for static, periodic variations in token production and consumption rates [5]. Each element of such a periodic variation corresponds to a distinct *phase* of execution for the associated actor. Thus firings of CSDF actors can be viewed as progressing through periodic sequences of phases.

Fig. 1(a) illustrates an example of a CSDF representation. Actor v_{src} has an execution sequence of $f_{src}(1)$, $f_{src}(2)$, $\cdots$,$f_{src}(P_{src})$, where each $f_{src}(i)$ represents the ith phase, and P_{src} represents the length (number of phases) in the execution period of v_{src}. Given a CSDF actor x, we refer to $P(x)$ as the *phase count* of x.

The periodic pattern of production rates associated with v_{src} can be represented as $[p_{src}(1), p_{src}(2), \cdots, p_{src}(P_{src})]$. In the ith firing of v_{src}, the actor produces $p_{src}(i)$ tokens onto edge e_t. From the periodic pattern in which v_{src} executes, we have that the actor produces $p_{src}((n-1)\ mod\ (P_{src}+1))$ tokens in each n-th phase. The execution sequence and consumption rate sequence associated with Snk can be represented in a manner analogous to the representations given above for v_{src}.

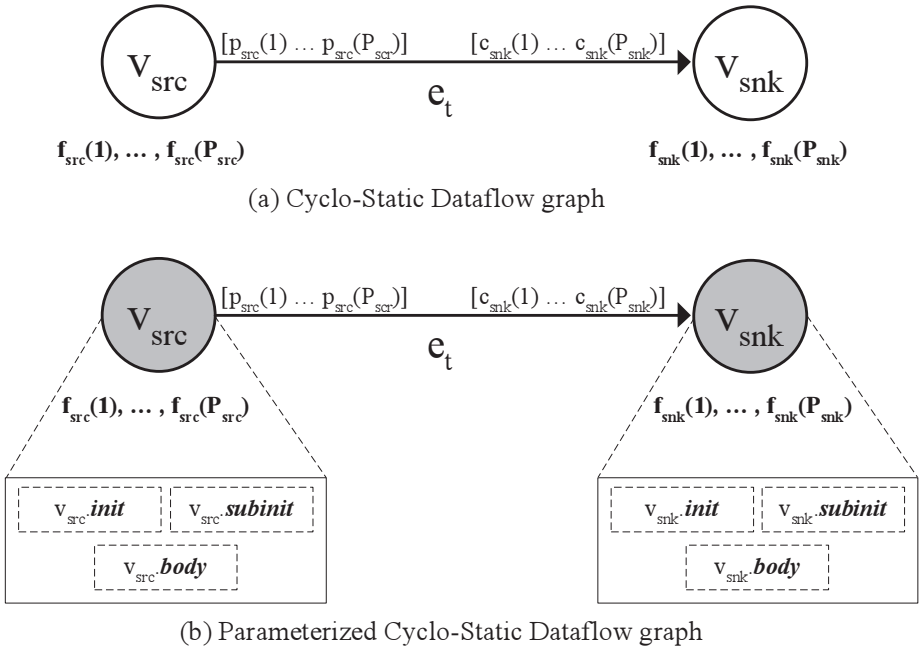


Fig. 1 CSDF graph and PCSDF graph examples.

3.2 Parameterized Cyclo-static Dataflow (PCSDF)

Parameterized Cyclo-static Dataflow (PCSDF) extends the expressive power of CSDF to manage DSP application dynamics in terms of run-time configuration of dataflow actor, edge, and subsystem parameters [6], [7]. PCSDF representations of applications are developed in terms of PCSDF *specifications*; when a PCSDF specification is encapsulated as a hierarchical actor in a higher level PCSDF graph, it is referred to as a PCSDF *subsystem*. A PCSDF specification is composed of three distinct PCSDF graphs — the *init graph*, *subinit graph*, and *body graph* [6].

The body graph models the core functional behavior of the enclosing specification, while the init and subinit graphs enable run-time configuration control for the behavior of the body graph. These configuration controllers provide two different levels of granularity in the run-time configuration processing — the init graph can form parameter configurations that are in general less restricted but also less frequent compared to the kinds of configurations that are allowed by the subinit graph.

Fig. 1(b) shows a PCSDF graph.

Conceptually, production and consumption rates in a CSDF actor have two properties — the period of the cycle of phases, and the data rate (i.e., the rates of token production and consumption) associated with each phase. In PCSDF, the value of any parameterized period and data rate values must (for valid operation) remain constant during any iteration of the underlying CSDF graph execution. However, across iterations, the length of the period and the associated data rates can be changed based on changes to parameter values that are propagated by the init or subinit graph. This

feature — in a manner analogous to other forms of paramterized dataflow [6] — allows PCSDF representations to express significant levels of application dynamics.

The modeling discipline imposed by the subinit and init graphs in PCSDF is designed to provide significant flexibility in how and when parameters are configured, while ensuring that configurations that affect the structure of subsystem schedules are allowed to occur only between iterations (in terms of CSDF repetitions vectors) of the associated subsystems. This allows each subsystem to be viewed as a dynamically evolving sequence of CSDF graphs whose SDF properties can change only at well-defined points in time (between CSDF graph iterations).

4 Scheduling Model

Given a DSP system represented as a CSDF graph that is to be mapped onto an FPGA, an important design problem is that of minimizing the buffer size (the memory requirements for the graph edges) subject to ensuring maximum throughput execution. To help address this problem, we introduce in this section a graphical schedule representation that captures relevant properties of the data transfer on edges in a given CSDF graph. This schedule representation helps to formalize our proposed synthesis approach, and ensure that the approach generates valid schedules that result in minimal buffer distributions over all maximal throughput solutions.

In our synthesis approach, we assume that every functional actor has a port to receive an internal clock control signal from a controller that enables or disables the internal clock signal. An actor is able to execute only if its internal clock signal is enabled. The controller provides a Boolean signal to each actor, and manages the firing pattern of actors based on a pre-defined schedule. Such a control port is common in commercial IP blocks for FPGAs, and our controller-based synthesis approach can therefore be incorporated naturally in practical DSP design flows that are based on existing FPGA IP blocks.

4.1 Modeling IP Blocks

To derive an efficient schedule that minimizes buffering costs, it is useful to have information about the timing patterns associated with token production and consumption in IP blocks. In our analysis, we model each IP block as a CSDF actor. In each phase of execution, the actor consumes and produces a certain number of tokens from each input and output edge, respectively. Such production and consumption is assumed to occur as an "atomic" action. That is, tokens are loaded (consumed) and unloaded (produced) from and to the associated buffers consecutively without any idle time between successive read and write operations. We refer to such atomic loading and unloading of data for a CSDF actor as the *contiguous interface* model of CSDF actor execution.

The production and consumption patterns of a CSDF actor are represented by sequences of non-negative integers, which are called *data rate signatures*. Each input and output port of a CSDF actor has a unique data rate signature, as shown in Fig. 2. Here, actors A and C have a single phase (per execution period) each and produce and consume 2 token and 1 token per firing, respectively. In contrast, B has two phases. In the first phase, B consumes 2 tokens and produces 1 token, while in the second phase, B consumes 1 token and produces 1 token.

Given a CSDF actor x, we model the patterns of loading and unloading tokens for x in terms of the *Execution Time Sequence*(ETS), the *Loading Index Sequence*(LIS), and the *Unloading Index Sequence*(UIS). Each of these sequences has $P(x)$ elements (recall that $P(x)$ represents the phase count of x).

We define the ETS of a CSDF actor v_i, denoted $ET(v_i)$, as the time duration sequence spent for each phase starting from the time instant that begins a phase execution to the time that finishes unloading the last token of the corresponding phase. The *Execution Time* of the k-th phase for an actor v_i is a positive integer that is represented by $ET(v_i, k)$. We define the *Loading Index* $LI(v_i, k)$ and *Unloading Index* $UI(v_i, k)$ as the time index (relative to the beginning of the corresponding execution phase) when the first token is loaded and unloaded, respectively, during the kth phase of an execution period of v_i.

Note that if v_i does not consume (produce) any token in kth phase, then $UI(v_i, k)$ ($LI(v_i, k)$) is defined to be infinity.

The sequences LIS and UIS are composed of the loading indices and the unloading indices, respectively. That is, for each j, the jth elements of LIS and UIS are, respectively, equal to $LI(v_i, j)$, and $UI(v_i, j)$.

An example is shown in Fig. 2. Here, $ET(B) = [6, 5]$ so the time durations required by the first and second phases of actor B's execution are 6 and 5 time units, respectively. Also, $UI(B) = [5, 4]$ so that the time indices to first unload a token in phase 0 and 1 are 5 and 4, respectively.

The formulation of LISs and UISs can be extended naturally to actors with multiple inputs and outputs (by adding an additional argument corresponding to the input/output port index). In this paper, for conciseness and clarity we focus on the single input, single output notation for conciseness and clarity — however our work is equally applicable to graphs that contain actors with multiple input and output ports.

Note that since in each phase tokens are loaded and unloaded consecutively without any idle time in between, the time index of the first loaded (unloaded) token within each phase can be used for representing the pattern of loading (unloading) tokens in through the phase That is, under our assumed scheduling model, LIS and UIS are sufficient to fully describe the loading and unloading patterns for actor firings. Profiles of loading and unloading patterns as well as actor execution times for each phase can typically be derived from data sheets of FPGA IP blocks. With this information, we can model a CSDF actor with its data rate signature, ETS, LIS, and UIS.

4.2 Contiguous Interface Scheduling Graph

In this section, we introduce a model called the *contiguous interface scheduling graph* ($CISG$) for representing periodic data transfers across a given edge ($e \in E$) in an enclosing CSDF graph $G = (V, E)$. Based on the CSDF modeling formulations developed in the previous sections, we specify attributes of vertices and edges in the CISG, and formulate buffer minimization as a linear programming problem.

We define a CISG $G_k^{SCH} = (V^{SCH}, E^{SCH})$ to represent token transfers on an edge e_k in G. In G_k^{SCH}, a *scheduling node* ($v_j^{SCH}(v_i) \in V^{SCH}$) represents the j-th firing of actor ($v_i \in V$) in a valid schedule, where v_i is connected to e_k. A *scheduling edge* ($e_t^{SCH} \in E^{SCH}$) represents either idle time between consecutive firings (phases) of v_i or token transfer on e_k.

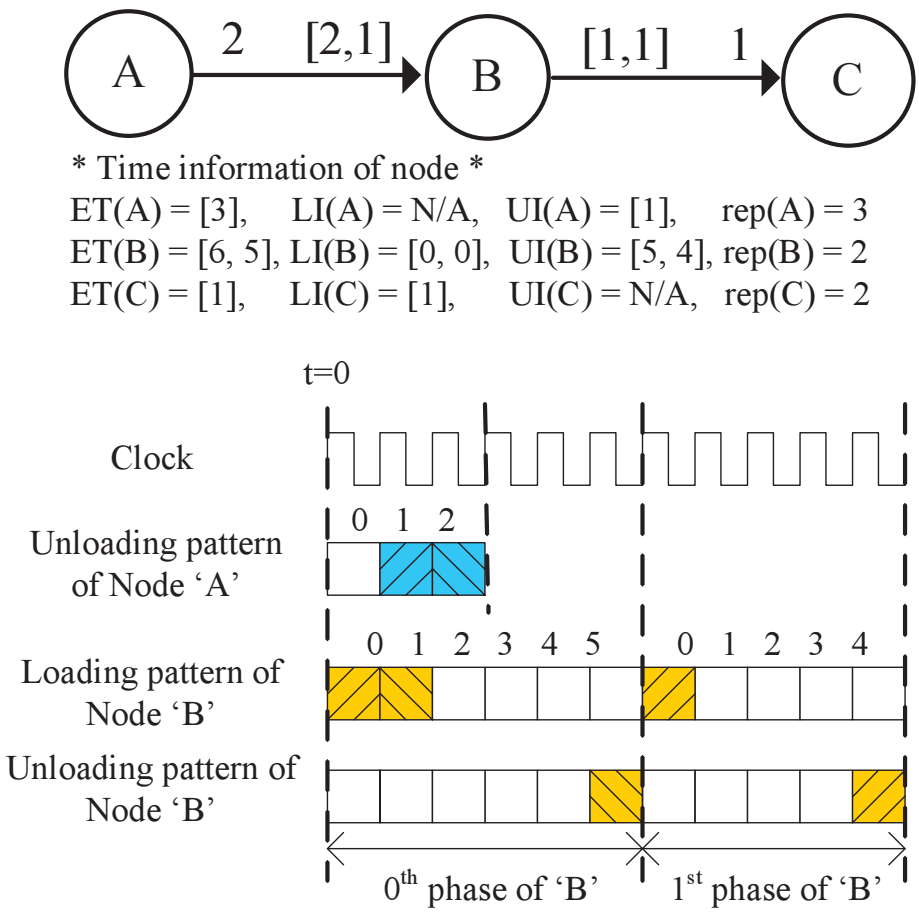


Fig. 2 An example of a CSDF graph with timing information and patterns of token loading and unloading.

The *initial node* $v_{initial}^{SCH}$ is added as an additional scheduling node. This node, which represents the beginning of the time clock, is connected to all $v_0^{SCH}(v_i)$ via scheduling edges. The *logical delay* (number of initial tokens) on each of these scheduling edges e represents the time that elapses until the first firing of the associated sink actor (i.e., the actor at the sink of e). Thus, the initial node together with its outgoing edges provides offsets among the initial starting times of actors that are connected to those outgoing edges.

A properly constructed CSDF graph has an associated *repetitions vector* $\mathbf{q_G} = [q(v_0), q(v_1), \cdots, q(v_{|V|-1})]$, where each $q(v_i)$ represents the number of firings of the lumped SDF actor v_i in a valid schedule for G [5], and a lumped SDF actor is derived from a corresponding CSDF actor v_i by merging all of the phases of v_i [17]. Thus, the total number of firings of a CSDF actor v_i in a valid schedule is $N_i (= q(v_i) * phase\#(v_i))$, where $phase\#(v_i)$ is the phase count of v_i. Also, there are N_i instances of the scheduling node $v_j^{SCH}(v_i)$ in G_k^{SCH}, where $0 \leq j \prec N_i$.

A scheduling edge can either be an *idle time delay edge* or a *token transfer edge*. An idle time delay edge connects a scheduling node $v_j^{SCH}(v_i)$ to $v_{(j+1)mod\ N_i}^{SCH}(v_i)$ with non-negative time delay $d(e_t^{SCH})$, and all nodes $v_j^{SCH}(v_i)$ representing firings of v_i are connected in a cycle by these edges. Non-negative time delays on edges represent idle times between consecutive firings of actor v_i. The unit of this time delay is one tick of the associated FPGA clock. Hence, idle time delays on this type of edge determine the schedule of the associated actor v_i.

While idle time delay edges connecting $v_j^{SCH}(v_i)$ to $v_{(j+1)mod\ N_i}^{SCH}(v_i)$ determine schedule evolution for the periodic steady state of graph execution, another schedule, which we call the *transient schedule*, is first needed to determine actor firing patterns before execution enters the steady state.

In G^{SCH}, edges connecting node $v_{initial}^{SCH}$to all $v_0^{SCH}(v_i)$ belong to the set of idle time delay edges. Because of differences among the time delays on these edges, a given actor v_i can have its initial firing earlier or later compared to the first firings of other actors in the application.

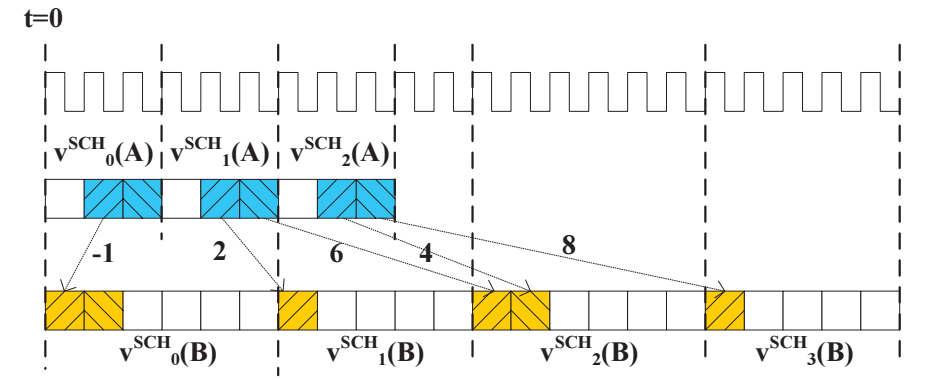
For a token transfer edge, e_t^{SCH} connects $v_i^{SCH}(v_{src})$ to $v_j^{SCH}(v_{snk})$ when a token produced from the i-th firing of a source actor v_{src} is consumed in the j-th firing of a sink actor v_{snk}, where v_{src} is connected to v_{snk} by and edge e_k in G. The time delay $d(e_t^{SCH})$ on this edge represents the time elapsed from when a token is produced by $v_i^{SCH}(v_{src})$ to when the same token is consumed by $v_j^{SCH}(v_{snk})$. For example, if a token is consumed one clock tick after being produced, the value of the time delay on the associated token transfer edge is 1. For correct operation, we must have $d(e_t^{SCH}) \geq 0$ for every token transfer edge.

In addition to assigning the edge time delay, we assign a *weight* $w(e_t^{SCH})$ to each scheduling edge e_t^{SCH}. The weight $w(e_t^{SCH})$ represents the number of tokens transferred across e_t^{SCH}. For example, when e_t^{SCH} connects $v_i^{SCH}(v_{src})$ to $v_j^{SCH}(v_{snk})$ in G_k^{SCH}, $w(e_t^{SCH})$ represents the number of tokens consumed by the jth firing of the sink actor v_{snk} out of the total number of tokens produced in the ith firing of the source actor v_{src}. A larger edge weight generally implies potential for larger buffer space requirements. Thus, the edge weight is involved in the objective function of our scheduling approach, and this weight is used to prioritize the scheduling edges so as to minimize overall buffer space requirements.

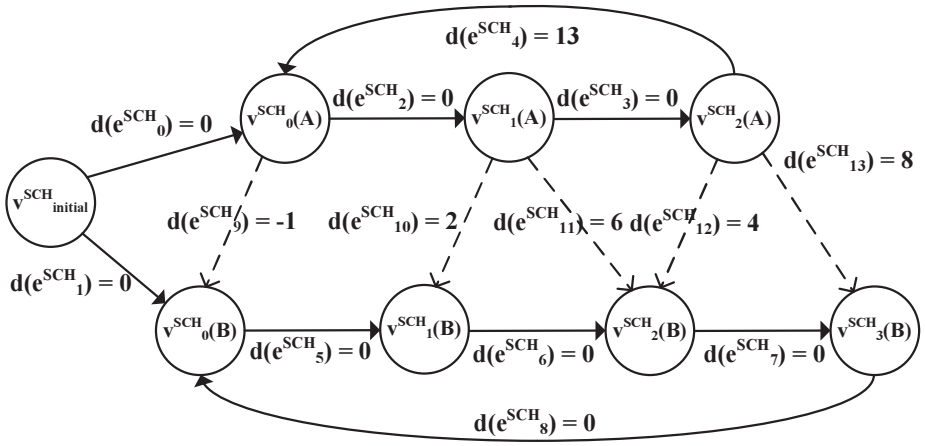
Since a token transfer edge represents the communication of data, the weight of a token transfer edge is always positive. Also, the weight of an idle time delay edge is always zero because there is no token transfer between consecutive firings of $v_j^{SCH}(v_i)$ and $v_{(j+1)mod\ N_i}^{SCH}(v_i)$. As token transfer edges and their associated weights in G^{SCH} are built from the fixed production and consumption rates of actors in a CSDF graph G, the weight of any given token transfer edge is constant. This is different from the time delay attribute of a scheduling edge, which can be changed by the schedule.

The initial schedule for actors can be generated by assigning time delays on idle time delay edges in G^{SCH}. Also, initial values of time delays on the token transfer edges are determined by this initial schedule. Since all actors are executed concurrently in FPGA implementation, a low-latency actor is generally delayed to synchronize with a high-latency actor with which it communicates. Thus, in our targeted scheduling model, the system throughput is determined by the highest-latency actor, and the maximum system throughput is achieved when this actor is executed without any idle time between consecutive firings.

In our scheduling analysis, we define the system *latency* as



(a) Loading/Unloading pattern of connected nodes in the initial schedule



(b) Generating Scheduling Graph G^{SCH}

Fig. 3 Token transfer in the initial schedule and the CISG generated from the edge e_0 in Fig. 2

.

$$L^{sys} = \max_{v_i \in V} \left\{ q(v_i) * \sum_{j=0}^{phase\#(v_i)-1} ET(v_i, j) \right\} \tag{1}$$

In general, a low-latency node must be delayed to synchronize in terms of this system latency, and the total idle time delay for each actor v_i is therefore computed as follows

$$D_i^{idle} = L^{sys} - q(v_i) * \sum_{j=0}^{phase\#(v_i)-1} ET(v_i, j) \tag{2}$$

In a valid schedule, v_i in G should have D_i^{idle} of idle time delay to achieve the maximum system throughput, and the sum of the time delays on idle time delay edges

connecting $v_j^{SCH}(v_i)$ in a cycle should be equal to D_i^{idle}. For generating G^{SCH}, we initialize the schedule by assigning zero time delay to all idle time delay edges except for the one that connects $v_{N_i-1}^{SCH}(v_i)$ to $v_0^{SCH}(v_i)$. We assign D_i^{idle} to this edge time delay to make all actors synchronized with L^{sys}. Delays on idle time delay edges are then changed by our proposed scheduling algorithm to generate a valid and optimal schedule. This scheduling algorithm is presented in Section 5.

Fig. 3 shows an example to demonstrate the process of CISG generation. This example shows the CISG for token transfer on e_0 of Fig. 2 with the initial schedule. In Fig. 3, actors A and B are connected by e_0. In a valid schedule, A and B are fired three and four times, respectively, because $q(A) = 3$, $phase\#(A) = 1$, $q(B) = 2$, and $phase\#(B) = 2$. Schedule nodes in Fig. 3(b) represent these firings in the schedule. $v_{initial}^{SCH}$is added for the initial offset between the first firings of A and B.

Solid and dotted edges in Fig. 3(b) represent the idle time delay edge and the token transfer edge, respectively. As shown in Fig. 2, the actor B has the highest latency in the system, and the time delays of idle time delay edges in the cycle for firings of B are all set to zero to achieve the maximum system throughput. Time delays on all edges that are incident to the initial node $v_{initial}^{SCH}$are initialized to zero, and time delays on edges in the cycle for A are also zero except for e_4. Since the sum of time delays in this cycle should equal the total periodic idle time delay D_A^{idle} in Eq 2, we have that $d(e_4^{SCH}) = 13$.

Time delays on token transfer edges are determined based on this initial schedule. As shown in Fig. 3(a), the first two tokens are consumed by B one tick before being produced by A. Hence, $d(e_9^{SCH}) = -1$, and $w(e_9^{SCH}) = 2$. The third and fourth tokens are produced from $v_1^{SCH}(A)$, but each of these tokens is consumed separately by $v_1^{SCH}(B)$ and $v_2^{SCH}(B)$. There are two separate token transfer edges from $v_1^{SCH}(A)$, and the weight of each of these edges is 1. As described earlier, all time delays on edges must be non-negative to satisfy schedule validity. The initial schedule shown in Fig. 3 violates this validity condition due to the negative time delay on e_9^{SCH}. The initial schedule is not a valid schedule, and thus, the system needs to be re-scheduled. In the next section, we will discuss how to systematically generate a schedule that is not only valid but also minimizes the buffer size while achieving maximum system throughput.

5 Scheduling Algorithm

Our optimization goal for generating a CISG G^{SCH} from a given CSDF graph G is to minimize the total required buffer space for graph edges subject to the maximum system throughput. The maximum system throughput can be realized when the the highest-latency actor operates without any idle time, and synchronizes with all other actors at this highest-latency, which becomes the system latency in Eq 1. Thus, when the sum of idle time delays in the schedule of each actor v_i is equal to D_i^{idle} in Eq 2, we are guaranteed that the maximum system throughput is achieved.

Two constraints , called the *CISG time delay constraints*, guide our proposed CISG-based buffer minimization approach. The first constraint is that the time delay sum of scheduling edges connecting $v_j^{SCH}(v_i)$, for $0 \leq j \prec N_i$, must be equal to D_i^{idle} in Eq 2. This comes from the synchronization requirements discussed in Section 4.2. The second constraint is that all time delays on scheduling edges must be non-negative. This ensures that dataflow semantics are respected (tokens are consumed only after

they are produced), and that multiple invocations of the same actor do not execute in parallel, which is a requirement of our targeted implementation model.

Two attributes, the time delay ($d(e_t^{SCH})$) and the weight ($w(e_t^{SCH})$), are assigned to each e_t^{SCH} in a CISG G^{SCH}. Both attributes are used in constructing the objective function of the targeted buffer minimization problem. In general, $d(e_t^{SCH})$ is equal to the required buffer space until $d(e_t^{SCH})$ reaches $w(e_t^{SCH})$, which represents the number of tokens transferred via e_t^{SCH}. If $d(e_t^{SCH}) > w(e_t^{SCH})$, then the buffer requirement is saturated at $w(e_t^{SCH})$.

In this sense, in scheduling CISG, we first try to minimize values of $d(e_t^{SCH})$ that correspond to higher values of $w(e_t^{SCH})$. Note that $w(e_t^{SCH})$ is constant in a given CISG. For example, assuming that $d(e_1^{SCH}) = 3, w(e_1^{SCH}) = 2$ for e_1^{SCH} and $d(e_2^{SCH}) = 3, w(e_2^{SCH}) = 5$ for e_2^{SCH}, the overall buffer requirement is 5. If we reduce $d(e_1^{SCH})$ by 1, the buffer requirement is not changed because it is still saturated at $w(e_1^{SCH})$. However, if we reduce $d(e_2^{SCH})$, the required buffer space is decreased.

Based on these considerations, the objective function of our minimization problem is defined as

$$f(E^{SCH}) = \sum_{e_i^{SCH} \in E^{SCH}} w(e_i^{SCH}) * d(e_i^{SCH}). \tag{3}$$

To minimize the buffer memory required for token transfer on e_k, we minimize $f(E^{SCH})$ in the CISG G_k^{SCH} generated for e_k. Since the weights of scheduling edges are constant, this objective function is a linear function of scheduling edge time delay variables. The CISG time delay constraints are taken into account in the optimization formulation. The resulting optimization problem can be formalized as a linear programming problem, and can be solved by the simplex algorithm in polynomial time [18]. We refer to the linear programming formulation associated with a CISG G_k^{SCH} as the *buffer optimization formulation* (*BOF*) for G_k^{SCH}.

5.1 Scheduling Algorithm for CSDF

In Section 4.2, we introduced our proposed approach to minimize buffering costs for a given CISG. When applying this approach, we decompose a CSDF graph G into a set of smaller graphs (*subset graphs*) $g_1, g_2, \ldots, g_N$, and generate CISGs associated with each of these smaller graphs. The buffer minimization problem in each of these CISG only provides a localized optimum solution that is associated with its corresponding g_i. To achieve a globally optimized solution for G, it is critical to process the g_is based on a strategically-determined order.

Let G_i^{ORD} represent the ith CISG that is processed based on a sequentially ordered processing of all subset graphs. The CISG G_0^{ORD} represents token transfer in a subset graph $G_0 = (V_0, E_0)$, and the resulting schedule guarantees a minimized buffer distribution for token transfers in E_0. The generated schedule for V_0 in general influences the result of buffer minimization for the CISG G_1^{ORD}, which is built for another subset graph G_1. Any shared nodes (i.e., nodes in the set $V'(= V_0 \cap V_1)$) have their schedules determined by the solution derived for G_0^{ORD}, and time delays on cyclic scheduling edges associated with V' in G_1^{ORD} are similarly determined. This kind of dependency link between schedules is exhibited in general for any pair G_i^{ORD} and $G_{(i+1)}^{ORD}$ of successive CISGs.

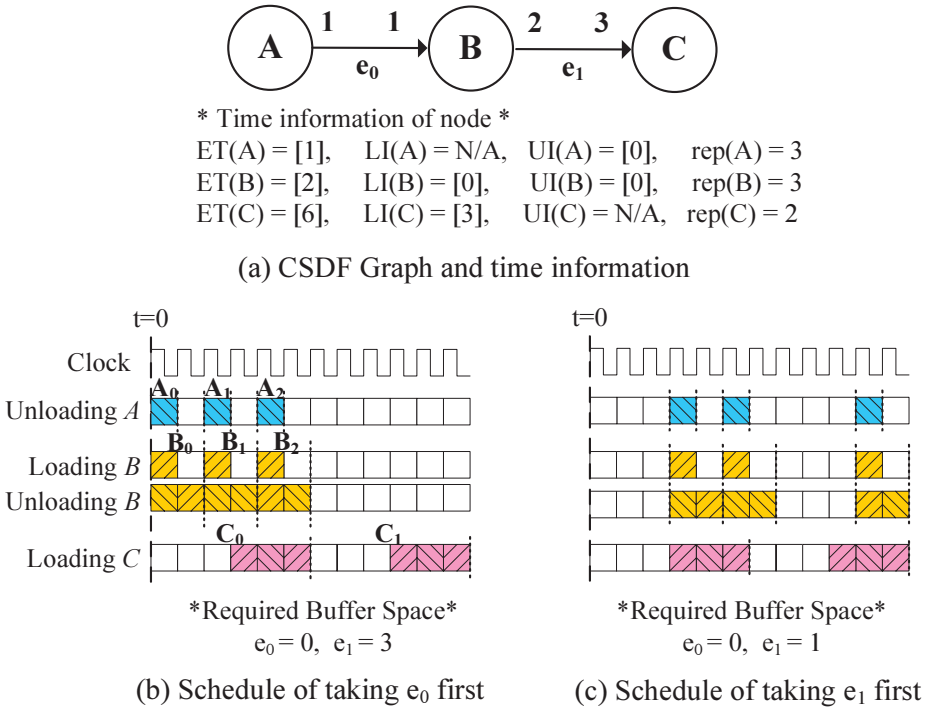


Fig. 4 Example showing the impact of edge selection order in processing CISGs.

Thus, while solving the BOF for a CISG G_i^{SCH} provides for local optimization of the associated subset graph, choosing an effective ordering G_i^{ORD} is critical to achieving global optimization of the overall CSDF graph G.

Fig. 4 illustrates the importance and impact of this ordering. Fig. 4(b) is the schedule constructed from first selecting e_0 and then e_1 in generating the CISG, and Fig. 4(c) is generated from selecting these edges in reverse order. When we choose e_1 first in generating the CISG, the overall buffer space for edges is reduced more effectively.

In the CSDF graph of Fig. 4(a), the actor C is the highest-latency actor. To achieve maximal system throughput, it is sufficient that such maximal-latency actors operate without idle time. In other words, if our goal is to to maximize throughput, the schedule for C is determined beforehand. If e_0 is first selected for scheduling A and B, the schedule of C is not considered in the CISG for e_0, and this results in a suboptimal buffer distribution. Even if the target (desired) throughput is not the maximal achievable throughput, the highest-latency actor should have the smallest total idle time delay (from Eq. 2) among all actors. In other words, the schedule of the highest-latency actor is more restricted than others due to its reduced scheduling range. Thus, to achieve a globally minimized buffer distribution, we first examine the highest latency actor.

If multiple actors are "tied" for the highest latency, we arbitrarily choose one these actors as the highest-latency actor in line 2 of algorithm 1.

Algorithm 1 shows our heuristic scheduling algorithm to for buffer-minimized, FPGA implementation of a CSDF graph G. The topologically sorted array L of nodes in line 3 is delimited by the maximum latency node and divided into two separate

subarrays l_0 and l_1 in lines 5-6. In line 7, l_0 is sorted in reverse order so that maximum latency nodes are located in the beginning of the list. The front node v_0 is first selected in each array during for-loop in line 10. Input edges to v_0 becomes the edge set E_0 and the actor set V_0 is composed of source nodes to E_0 and v_0 in lines 11-12. The scheduling graph G^{SCH} is built from this subgraph $G_0 = (V_0, E_0)$ in line 13. The simplex algorithm is applied to the constraints and the objective function from G^{SCH} and the result schedule is assigned to V_0 in lines 14-15. If a node in V_0 is included in generating another G^{SCH} during for-loop, this schedule of the node is used in assigning the value to time delays on idle time delay edges. Since we traverse nodes in G in the order of array l_0 and l_1, the complexity of the overall algorithm is polynomial. The proposed algorithm is implemented by the dataflow interchange format (*DIF*) package, which provides a standard language and associated toolset that is founded in dataflow semantics and tailored for DSP system design [19].

Algorithm 1 Scheduling CSDF graph object to get the minimized buffer distribution

```
 1: procedure SCHEDULECSDFGRAPH(G)
 2:     v_maxLatency ← getMaxLatencyNode(V)
 3:     L ← topologicalSort(G)
 4:     i ← getIndex(L, v_maxLatency)
 5:     l_0 ← getSubArray(L, 0, i)
 6:     l_1 ← getSubArray(L, i + 1, |V| − 1)
 7:     l_0 ← reverseSort(l_0)
 8:     for i = 0 to 1 do
 9:         for j = 0 to |l_i| − 1 do
10:             v_0 ← getNode(l_i, j)
11:             E_0 ← getInputEdges(G, v_0)
12:             V_0 ← v_0 ∪ sourceNodes(E_0)
13:             G^SCH ← generateScheduleGraph(V_0, E_0)
14:             S ← simplexAlgorithm(G^SCH)
15:             assignSchedule(S, V_0)
16:         end for
17:     end for
18: end procedure
```

6 PCSDF Scheduling Algorithm

In this section, we build on our scheduling technique for CSDF graphs, and develop a more powerful technique that is geared toward PCSDF graphs. As described in Section 3.2, PCSDF is significantly more expressive compared to CSDF, and is well-suited to describing application dynamics for modern wireless communication applications.

6.1 Schedule Representation

To map a PCSDF schedule into an implementation targeted on an FPGA, we determine for each actor the idle time between consecutive phases of the actor. These idle time values in turn determine the starting times of the phases, and are configured so as to minimize the numbers of tokens stored in the actor input and output buffers. In our

analysis, we refer to each block of idle time as a *Scheduling Element* (*SE*); we refer to a series of SEs as a *Scheduling Sequence* (*SS*).

In our approach to FPGA implementation, the generated SS for each actor is stored in a FIFO; the required memory space for this FIFO is $W \cdot L$, where W is the SE word length, and L is the SS length. Thus, if L is large, the scheduling logic consumes a correspondingly large amount of memory. However, if local periodicities (repetitive subsequences) can be detected within the SS, then the SS can be represented by appropriately constructed iterative constructs (loops). In each such loop, a single instance of a periodic subsequence is stored along with the associated iteration count (number of successive repetitions). Such use of looping constructs in the schedule representation can save significant amounts of memory space.

We apply the previously developed *generalized scheduling tree* (*GST*) representation [20] for modeling dataflow graph schedules that involve looping constructs. We develop a specialized form of GSTs, called the *CISG GST*, that is suitable for our proposed PCSDF scheduling model. CISG GSTs are ordered trees with leaf nodes representing SEs. An internal node of a CISG GST represents a parameterized iteration count for a parameterized schedule loop that is rooted at the internal node. Such a parameterized iteration count provides a representation for a loop iteration count that can be adapted at run-time depending on values of dynamic parameters in the application or implementation model. In our experiments, we apply two types of expressions for parameterized iteration counts: a case expression, which provides an enumerated set of alternative values, or an algebraic expression in terms of relevant parameters.

When processed at run-time, each parameterized iteration count must be evaluated into a non-negative integer $\mathbb{Z}_{\geq 0}$. Sub-trees rooted at internal nodes that evaluate to zero-valued iteration counts are effectively "hidden" while traversing the CISG GST. By allowing zero-valued iteration counts, we can therefore describe conditional behaviors within the same framework as dynamic-iteration-count looping structures.

Fig. 5 shows an example to demonstrate our CISG GST approach. In this example, the given schedule is controlled by the parameter set p_1, p_2. Fig. 5(b) shows possible scheduling sequences based on the set of different parameter combinations. From the scheduling sequences, we can observe that the scheduling elements s_1 and s_2 are only in $SS(p_1 = 1)$, while s_3 and s_4 are in $SS(p_1 = 2)$. Here, $SS(p = v)$ represents the scheduling sequence that results when parameter p has value v.

Since the parameterized iteration count of $I_1(p_1)$ and $I_2(p_1)$ in the CISG GST is formulated in the form of a "case expression", one of the sub-trees rooted at $I_j(p_1)$ is hidden depending on the value of parameter p_1. In contrast, $I_3(p_2)$ is a parameterized iteration count expressed in algebraic form; the associated expression determines the iteration count for the schedule element s_1 in terms of the parameter p_2. All other internal nodes in the CISG GST exhibit constant iteration counts for the associated looping constructs.

6.2 Scheduling Process

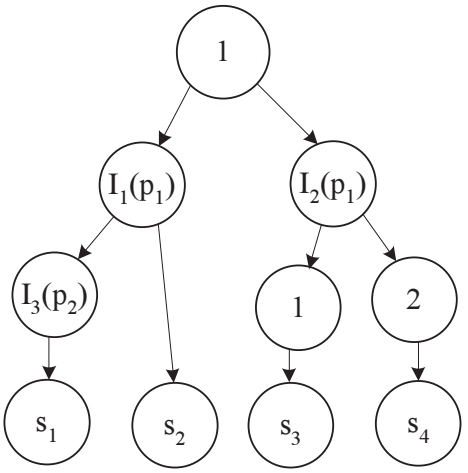
As we discussed in Section 3.2, in PCSDF semantics the production and consumption rates of each actor in a PCSDF graph are determined before an invocation of the graph, and this graph can be considered as a CSDF graph during such an invocation. With the CSDF scheduling technique developed in Section 5.1, we schedule all possible CSDF graphs defined by the set of distinct parameter combinations. The resulting set

Parameter Set = ParamSet(p_1, p_2)
p_1 = {1 ,2}
p_2 = {2, 3}

(a) Parameter Set

SS(p_1=1, p_2 =2) = [s_1, s_1, s_1, s_2]
SS(p_1=1, p_2 =3) = [s_1, s_1, s_1, s_1, s_2]
SS(p_1=2, p_2 =2) = [s_3, s_4, s_4]
SS(p_1=2, p_2 =3) = [s_3, s_4, s_4]

(b) All possible Scheduling Sequences



* Parameterized loop count
$I_1(p_1=1) = 1$, $I_1(p_1=2) = 0$
$I_2(p_1=1) = 0$, $I_2(p_1=2) = 1$
$I_3(p_2) = p_2 + 1$

(c) GSTs for (b) with parameterized loop count definitions.

Fig. 5 CISG GST example.

of CSDF schedules are then stored and switched among dynamically to implement the desired PCSDF schedule. Such an approach becomes impractical if the number of different CSDF schedules that must be stored is very large. However, in some practical applications, such as the LTE system that we study in Section 8, a relatively moderate number of CSDF schedules need to be managed dynamically; our proposed technique is geared towards exploiting the dataflow structure of such applications.

When applying this approach, the alternative CSDF schedules that are derived will in general have different buffer distributions. We determine the buffer sizes in the derived implementation based on the maximum buffer costs across the different CSDF schedules (parameter combinations).

In addition to buffer memory requirements, schedule size is also an important implementation metric for PCSDF graphs. Under our approach to implementing PCSDF actors, the memory space required for storing a schedule is the sum of the underlying CSDF schedule sizes over all valid parameter combinations.

A straightforward approach to reducing schedule size is the sharing of common schedule elements. In our CSDF scheduling approach, such sharing can be achieved by assigning common variables associated with scheduling edges in G^{SCH}. After scheduling G^{SCH}, shared variables then reference common schedule elements. However, this sharing technique not only reduces the number of independent variables in the scheduling of G^{SCH}, but it also can reduce opportunities for buffer size optimization. For

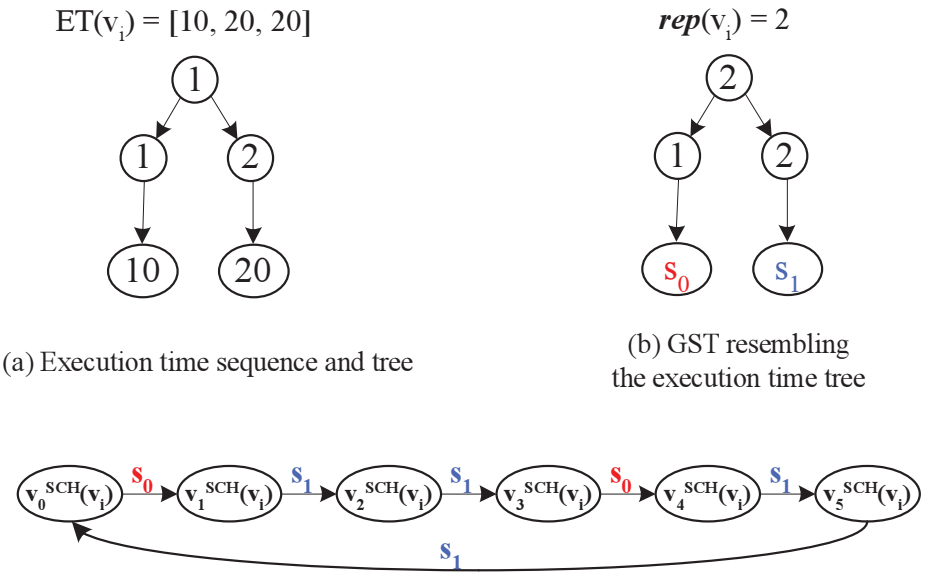


Fig. 6 An example of an ETT.

example, if a set of delay variables is defined by $[2 * v_0 \;\; v_0 \;\; v_1]$, any schedule that does not conform to this matrix can be pruned out from our search space. Therefore, special care is needed in applying the sharing technique.

In our approach for reducing schedule size, we apply a data structure called the *execution time tree* (*ETT*), which is derived from the CISG GST. In the ETT, all internal nodes represent constant iteration counts, and all leaf nodes represent execution times associated with actor phases. Fig. 6(a) illustrates an ETT associated with an actor v_i. The tree structure of the CISG GST is the same as the corresponding ETT, and all internal nodes except for the root node and leaf nodes in the CISG GST have the same iteration counts as the corresponding nodes in the ETT. The root node in the CISG GST represents $rep(v_i)$, the repetition count of v_i. All leaf nodes in the CISG GST represent independent delay variables.

We used the CISG GST to construct GST schedule structures where schedule elements share references to common schedule elements. For a given actor v_i, the variable sequence generated from traversing the CISG GST for v_i provides time delay variables. These time delay variables are associated with successive scheduling edges in G^{SCH} for the cycle that represents periodic firings of the actor v_i. Since local periodicity in an execution time sequence can be concisely represented in the CISG GST by appropriate iteration counts, variables on scheduling edges can share common storage through iteration counts in the CISG GST. Fig. 6(b) illustrates the construction of the CISG GST from the ETT, and Fig. 6(c) illustrates sharing of variables in G^{SCH} by use of the CISG GST in Fig. 6(b).

7 FPGA Implementation

In Section 6, we developed methods to derive efficient schedules for FPGA implementation of PCSDF graphs. In this section, we describe how to synthesize these schedules onto the targeted FPGA devices. In our synthesis approach, the firings of each actor v are controlled by a *schedule controller* that is dedicated to v. These schedule controllers can be viewed as hardware implementations of CISG GSTs.

7.1 Schedule Controller Implementation

In our implementation approach, a schedule controller consists mainly of an internal counter and a generated schedule. At the end of execution of an actor phase, the actor sends a "done" signal to the associated schedule controller. This signal triggers initialization of a "roll-over value" associated with the internal counter in the schedule controller. This value is based on an corresponding schedule element s_i from the derived schedule. The internal counter counts from 1 to the roll-over value, and sends a "start" signal to the actor when the count is complete (the roll-over value is reached). Hence, the actor is maintained in an idle state for s_i cycles from the end of the most recently completed phase until the start of the next phase.

During the periodic execution of an actor v_i, the actor is fired $rep(v_i) * phase\#(v_i)$ times. Hence, the length of the associated CSDF schedule is $rep(v_i) * phase\#(v_i)$ firings, and the associated schedule controller has a schedule FIFO whose size is equal to this schedule length.

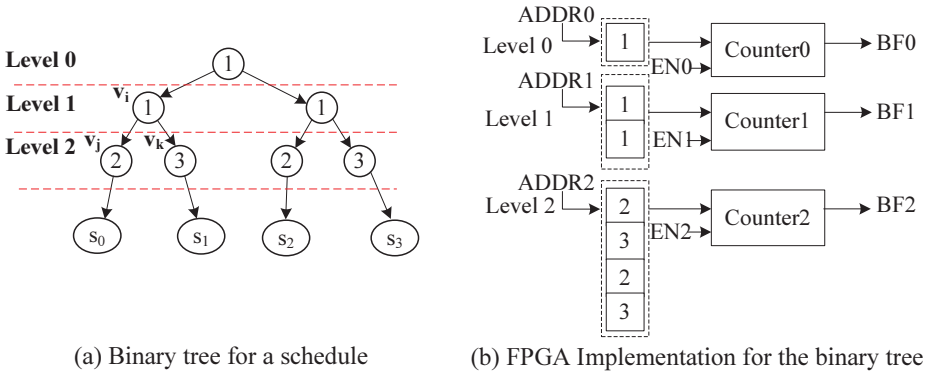
7.2 Binary Tree Implementation

In our implementation approach, a schedule controller coordinates an actor schedule in terms of its CISG GST representation. To run the schedule rooted at a node v_i in the CISG GST, all sub-schedules rooted at children nodes of v_i are executed iteratively based on the iteration counts of the nodes. Such schedule execution is performed recursively to execute the overall schedule represented by the CISG GST.

For example, the schedule S_j rooted at v_j in Fig. 7(a) is executed by running s_0 twice; the schedule S_k rooted at v_k runs s_1 three times; and the schedule S_i rooted at v_i executes S_j and S_k once each.

A binary tree is well-suited to digital hardware implementation due to its binary nature. In the remainder of this section, we develop an efficient implementation technique for binary tree structures and associated controllers for CISG GST traversal.

Fig. 7(b) illustrates our approach to hardware implementation for executing a schedule by means of CISG GST traversal. For the i-th level of tree execution, there is a single counter and a memory M having 2^i spaces. For execution at the ith level, a counter is implemented to count the iterative execution of the schedule rooted by the corresponding tree node at that level. Traversing the CISG GST requires switching among nodes at the same level; such a switch can be managed by the address $ADDR_i$ of the ith cell in M. The counter has two input ports and one output port. The first input port is used to initialize the roll-over value of the counter by the data pointed to by $ADDR_i$. The second input port receives a Boolean input that specifies whether to



(a) Binary tree for a schedule (b) FPGA Implementation for the binary tree

Fig. 7 Binary tree implementation.

increment the internal variable of the counter. When the internal variable reaches to the roll-over value, the counter produces a `true` signal on the output port BF_i.

The controller for traversing the CISG GST is composed of an address generator for $ADDR_i$, and an enable signal generator for EN_i. The address generator should reference the correct iteration count value at each level of the CISG GST. The value $ADDR_i$ is computed by the following two rules: 1) The last bit of $ADDR_i$ is flipped when a `true` value is asserted on BF_i; 2) $ADDR_k = 2^{k-i} * ADDR_i$ for $k > i$ when $ADDR_i$ is changed. When the counter reaches its roll-over value, the schedule rooted by node v_i at $ADDR_i$ has executed the same number of times as the the iteration count of node v_i. After this, the next schedule rooted by a sibling node v_s of v_i is executed. The flip of the last bit in $ADDR_i$ changes the address to reference v_s, and initialize the roll-over value by the iteration count associated with v_s. When a change of the i-th level node occurs, the schedule rooted by v_i is correspondingly changed.

The enable generator determines when to increment the internal value of the counter. This generator asserts a `true` value to EN_i when the last bit of $ADDR_{i+1}$ is flipped from `true` to `false`. This is because the schedule rooted by the node v_i is executed once both schedules rooted by the left and right children have executed the number of times specified by their respective iteration counts. When the schedule rooted by the right child is complete, execution moves to the schedule of the sibling node, and this move is represented by a flipping of the last bit in $ADDR_{i+1}$.

We apply the above approach to implementing *balanced* binary trees. If the given binary tree is unbalanced, some slots of the iteration count memory in Fig. 7(b) are empty. In order to handle unbalanced trees in our scheme, we balance such a tree by adding dummy nodes with zero-valued iteration counts, and initializing empty spaces with zero values. Because of their zero-valued iteration counts, such dummy nodes are effectively ignored when the tree is traversed.

8 Case Study: Long-Term Evolution(LTE) Base Station Transmitter

To demonstrate the properties and capabilities of PCSDF representations, we applied our synthesis techniques to the physical layer for 3GPP-Long Term Evolution (LTE), an important next generation cellular standard.

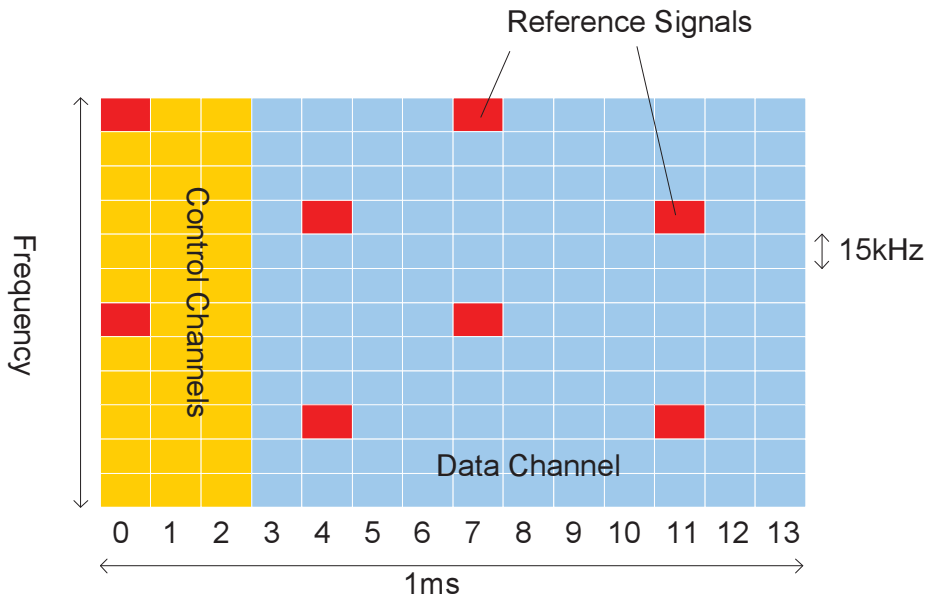


Fig. 8 Example LTE subframe showing multiplexing of various channels on a 2D time-frequency grid (not to scale).

The LTE downlink physical layer is based on the modulation and multiple access scheme called Orthogonal Frequency Division Multiple Access (*OFDMA*) OFDMA [21]. OFDMA uses an inverse fast Fourier transform (IFFT) to divide a wideband channel into multiple narrowband channels. This creates a two-dimensional resource grid in frequency and time. In LTE, each element of this grid is called a *resource element*. This 2D grid allows multiplexing various physical channels, e.g., data and control channels, which can be intended for multiple users.

An example of a 1ms LTE subframe comprising 14 OFDMA symbols in the normal cyclic prefix mode is shown in Fig. 8. LTE can be configured for 4 different bandwidths, namely 5, 10, 15, and 20 MHz, but still maintain a constant 15 kHz subcarrier spacing. The LTE physical layer can also support multiple antenna transmission schemes, including transmit diversity, beamforming, and spatial multiplexing, but our paper primarily focuses on implementation for the single-antenna transmission mode.

Fig. 9 illustrates a PCSDF model for a single-antenna LTE Base Station (BS) Modulator. Each of the solid blocks corresponds to a PCSDF actor whose production and consumption rates (annotated on the solid edges) can change in terms of graph parameters. These parameters are indicated by the dashed blocks that are communicated by the dashed edges. The data, control, and reference symbol generation blocks provide QPSK, 16-, or 64-QAM symbols that are multiplexed via the Resource Element (RE) mapper. The RE mapper takes in different numbers of symbols s_1, s_2, and s_3 from the available input ports as a function of the number of control symbols ($N_{ctrl} \in \{1, 2, 3, 4\}$), subframe index ($Sfidx \in \{0, .., 9\}$), bandwidth configuration ($BW \in \{1.4, 3, 5, 10, 15, 20\}$), cyclic prefix mode ($CP_{mode} \in \{Normal, Extended\}$), and symbol index ($SymbIdx \in \{0, .., 13\}$)(Fig. 10). These symbols are multiplexed into $N_u \in \{72, 180, 300, 600, 900, 1200\}$ used subcarriers, which is a direct map from

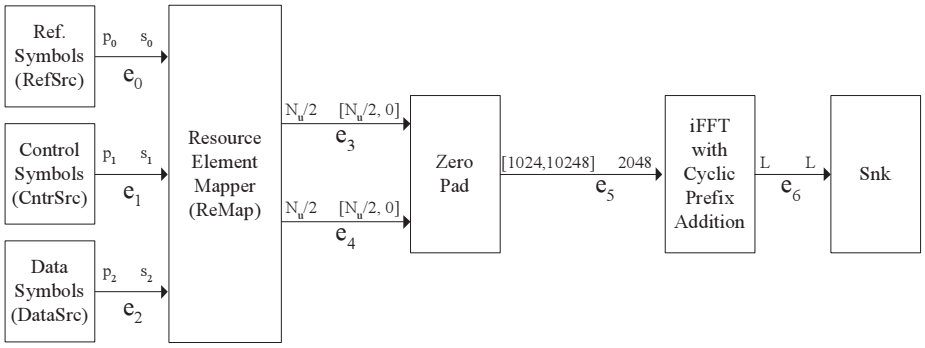


Fig. 9 PCSDF Model for LTE BS Modulator.

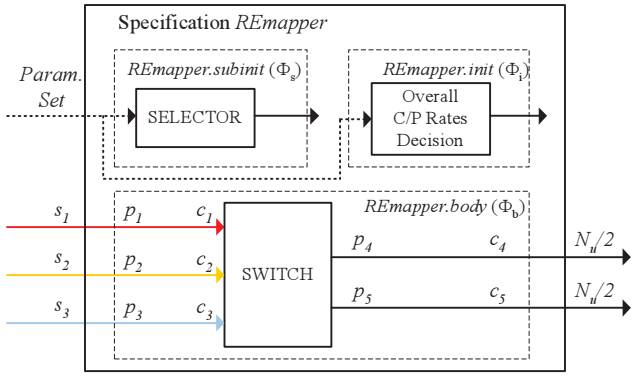


Fig. 10 PCSDF specification for RE Mapper.

Table 1 Schedule size of all actors without introducing GST

Param.	rSrc	cSrc	dSrc	ReMap	ZeroPad	iFFT	Snk	Param.	rSrc	cSrc	dSrc	ReMap	ZeroPad	iFFT	Snk
p=0	200	1	13	420	28	14	14	p=16	200	1	11	416	24	12	12
p=1	200	2	12	420	28	14	14	p=17	200	2	10	416	24	12	12
p=2	200	3	11	420	28	14	14	p=18	200	3	9	416	24	12	12
p=3	200	4	10	420	28	14	14	p=19	200	4	8	416	24	12	12
p=4	400	1	13	820	28	14	14	p=20	400	1	11	816	24	12	12
p=5	400	2	12	820	28	14	14	p=21	400	2	10	816	24	12	12
p=6	400	3	11	820	28	14	14	p=22	400	3	9	816	24	12	12
p=7	400	4	10	820	28	14	14	p=23	400	4	8	816	24	12	12
p=8	600	1	13	1220	28	14	14	p=24	600	1	11	1216	24	12	12
p=9	600	2	12	1220	28	14	14	p=25	600	2	10	1216	24	12	12
p=10	600	3	11	1220	28	14	14	p=26	600	3	9	1216	24	12	12
p=11	600	4	10	1220	28	14	14	p=27	600	4	8	1216	24	12	12
p=12	800	1	13	1620	28	14	14	p=28	800	1	11	1616	24	12	12
p=13	800	2	12	1620	28	14	14	p=29	800	2	10	1616	24	12	12
p=14	800	3	11	1620	28	14	14	p=30	800	3	9	1616	24	12	12
p=15	800	4	10	1620	28	14	14	p=31	800	4	8	1616	24	12	12

the bandwidth configuration BW. The Zero Pad block then takes in N_u symbols and appends zeros at the DC and edge subcarriers forming 2048 frequency domain complex values. The following block then performs a 2048-pt IFFT, and appends a cyclic prefix of length that is a function of the CP_{mode} and $SymbIdx$ parameters.

9 Experimental Results

In our experiments, we modeled the targeted LTE BS transmitter application using the PCSDF graph shown in Fig. 9, and we generated PCSDF schedules based on our proposed scheduling techniques. We first scheduled the PCSDF graph without incorporating GSTs into the derived scheduling structures. This approach is expected to result in a minimal data buffer distribution at the expense of increased memory requirements for schedule storage.

Next, we scheduled the PCSDF graph with GSTs, which enables sharing of schedule elements. To demonstrate the associated trade-offs, we compared the two scheduling approaches in terms of implementation costs for data buffer and schedule storage. We then derived a hybrid schedule that applies GSTs selectively based on analysis of the schedules constructed with and without GSTs.

There are three parameters affecting the functionality of the LTE transmitter system — the cyclic prefix mode($CP_{mode} \in \{Normal, Extended\}$), number of subcarriers ($N_u \in \{300, 600, 900, 1200\}$), and number of control symbols ($N_{ctrl} \in \{1, 2, 3, 4\}$). These parameters are encoded into "composite parameter" p, where CP_{mode}, N_u, and N_{ctrl} are assigned to the most significant bit; second and third bits; and fourth and fifth bits, respectively. For instance, $p = 5'b01001$ represents that the LTE transmitter is configured to have the normal cyclic prefix mode, 900 subcarriers, and 2 control symbols.

In an iteration of its PCSDF execution, the top dataflow graph in Fig. 9 produces the LTE sub-frame in Fig. 8. Table 1 shows schedule sizes of actors when the schedule structure is not confined by GSTs during scheduling.

Table 2 shows FIFO buffer distributions associated with different scheduling techniques. The word length for data on each edge is specified under e_i in the first row of the table. The FIFO length for each e_i under each scheduling technique is represented in the remaining rows. The second row in Table 2 shows the buffer distribution associated with the schedule in Table 1. The FIFO length on each edge is determined to accommodate the the worst case among the the schedules. According to this buffer distribution, edges e_3, e_5 and e_6 can be implemented by wires instead of FIFOs. Thus, the generated schedules guarantee that any token produced from a source actor connected to these edges is not stored in a FIFO; instead such a token is consumed immediately by a "downstream" actor. In contrast, the edge e_4 requires a FIFO of size 600.

The third row in Table 2 shows the buffer distribution associated with the schedule derived using GSTs. Compared to the second row in Table 2, e_2 and e_5 require more buffer spaces in this approach. These edges exhibit the trade-off of decreased schedule storage cost under GST-based implementation at the expense of increased buffer cost.

However, the overall buffer size increase due to GST usage is only 2% of the total memory required for implementing the FIFOs associated with dataflow graph edges. This result shows that in the targeted LTE application, our GST-based scheduling technique shares schedule elements efficiently with only a small overhead in buffering cost.

Table 3 shows the height of the binary tree for each actor schedule when we use GSTs in the scheduling process. Nodes in these trees represent either parameterized iteration counts or parameterized schedule elements. The numbers of nodes in the schedule trees for CntrSrc and DataSrc in Table 3 are 3 and 4, respectively. The total number of nodes in each tree is $(2^h - 1)$, where h is the tree height.

Table 2 FIFO Buffer distribution

Method	e0 2bit	e1 16bit	e2 4bit	e3 16bit	e4 16bit	e5 16bit	e6 32bit
W/O GST	1	166	166	0	600	0	0
W/ GST	1	166	171	0	600	21	0
Hybrid	1	166	166	0	600	21	0

Table 3 Height of schedule tree when incorporating to GST

RefSrc	CntrSrc	DataSrc	ReMap	ZeroPad	iFFT	Snk
3	3	4	5	2	2	2

When we compare these numbers to the schedule sizes in Table 1, the memory costs for the schedule trees are greater than those of the corresponding schedule sequences in Table 1. This is because the iteration counts in the schedule trees are less than or equal to 2 for the schedules of CntrSrc and DataSrc, and the tree representations are consequently more expensive than the sequence representations. In other words, it is more efficient not to incorporate GSTs in the scheduling of CntrSrc and DataSrc.

The last row in Table 2 shows the buffer distribution when we use GSTs for all actors except for CntrSrc and DataSrc. Since the schedule for DataSrc is not constrained to be GST-based, we can achieve a FIFO length on e_3 that is equal to that in the second row of the table. Since we still use GSTs in scheduling iFFT and Snk, the FIFO length on e_5 is same as that in the third row.

10 Conclusion

We have presented a scheduling algorithm that jointly minimizes data buffer distributions and schedule storage costs for synthesis of parameterized cyclo-static dataflow graph models onto FPGAs. This minimization is performed subject to the constraint that the maximum achievable throughput is maintained. We apply generalized schedule trees (GSTs) for realize compact scheduling representations. Incorporating GSTs in a schedule generally increases data buffering costs, but for the targeted LTE transmitter, we show that the increased buffering cost is relatively small. After analyzing costs for schedule implementation with and without GSTs, we realized a hybrid schedule implementation in which GSTs are applied only to edges that benefit from their application. This hybrid implementation is shown to provide an especially useful trade-off between schedule and buffer storage costs. Our work appears promising for integration into high-level design processes for FPGA-based DSP system implementation, especially in the domain of fourth generation wireless communication systems.

References

1. Edward Ashford Lee and David G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.
2. C. Hsu, S. Ramasubbu, M. Ko, J. L. Pino, and S. S. Bhattacharyya, "Efficient simulation of critical synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, San Francisco, California, July 2006, pp. 893–898.

3. C. B. Robbins, *Autocoding Toolset Software Tools for Automatic Generation of Parallel Application Software*, Technical report, Management Communications and Control, Inc., 2002.
4. Jens Horstmannshoff and Heinrich Meyr, "Efficient building block based rtl code generation from synchronous data flow graphs," in *DAC '00: Proceedings of the 37th Annual Design Automation Conference*, New York, NY, USA, 2000, pp. 552–555, ACM.
5. G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.
6. B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.
7. S. Saha, S. Puthenpurayil, and S. S. Bhattacharyya, "Dataflow transformations in high-level DSP system design," in *Proceedings of the International Symposium on System-on-Chip*, Tampere, Finland, November 2006, pp. 131–136, Invited paper.
8. R. Reiter, "Scheduling parallel computations," *J. ACM*, vol. 15, pp. 590–599, October 1968.
9. Ali Dasdan, Ali Dasdan, Rajesh K. Gupta, and Rajesh K. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, pp. 889–899, 1998.
10. J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1995, pp. 122–126 vol.1.
11. A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. G. Bekooij, "Throughput analysis of synchronous data flow graphs," in *ACSD '06: Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, Washington, DC, USA, 2006, pp. 25–36, IEEE Computer Society.
12. Er Stuijk, Marc Geilen, and Twan Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *In DAC*. 2006, pp. 899–904, ACM Press.
13. Marc Geilen, Twan Basten, and Er Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *in Proceedings of the Design Automation Conference*. 2005, pp. 819–824, ACM.
14. S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.
15. Jens Horstmannshoff and Heinrich Meyr, "Optimized system synthesis of complex rt level building blocks from multirate dataflow graphs," in *ISSS '99: Proceedings of the 12th international symposium on System synthesis*, Washington, DC, USA, 1999, p. 38, IEEE Computer Society.
16. H. Kee, I. Wong, Y. Rao, and S. S. Bhattacharyya, "FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Dallas, Texas, March 2010.
17. S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, CRC Press, second edition, 2009.
18. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, 2nd edition*, MIT Press, McGraw-Hill Book Company, 2000.
19. C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, Dallas, Texas, September 2005, pp. 37–49.
20. M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, "Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation," *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, June 2007.
21. 3G Americas, *The Mobile Broadband Evolution: 3GPP Release 8 and beyond*, Feb. 2009.