In Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, pages 136-143, 2010.

Efficient Static Buffering to Guarantee Throughput-Optimal FPGA Implementation of Synchronous Dataflow Graphs

Hojin Kee and Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering,
University of Maryland
College Park, MD 20742, USA
Email: {hjkee,ssb}@umd.edu

Jacob Kornerup
National Instruments Corp.
Austin, TX 78759, USA
Email: {jacob.kornerup}@ni.com

***Abstract*—When designing DSP applications for implementation on field programmable gate arrays (FPGAs), it is often important to minimize consumption of limited FPGA resources while satisfying real-time performance constraints. In this paper, we develop efficient techniques to determine dataflow graph buffer sizes that guarantee throughput-optimal execution when mapping synchronous dataflow (SDF) representations of DSP applications onto FPGAs. Our techniques are based on a novel two-actor SDF graph Model (*TASM*), which efficiently captures the behavior and costs associated with SDF graph edges (flow-graph connections). With our proposed techniques, designers can automatically generate upper bounds on SDF graph buffer distributions that realize maximum achievable throughput performance for the corresponding applications. Furthermore, our proposed techinque is characterized by low polynomial time complexity, which is useful for rapid prototyping in DSP system design.**

***Keywords*-Synchronous dataflow, throughput, buffer memory, FPGA, signal processing systems.**

I. Introduction and related work

In the design of DSP applications, it is important to consider real-time constraints as well as optimization of hardware resources. Synchronous dataflow (SDF) [1] has been used widely as an efficient model of computation for analyzing performance and resource requirements of DSP applications that are implemented on various target architectures (e.g., see [2], [3], [4], [5], [6]).

When describing a DSP application with an SDF 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 idle time that is required to prevent buffer overflow — as buffers become smaller, the frequency and duration 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 guaranteed, throughput-optimal buffer configuration for SDF graphs. In this paper, we study this problem in the context of FPGA-based implementation.

Traditionally, throughput analysis for SDF graphs is performed by solving an instance of the maximum mean cycle problem (e.g., see [7], [8]) after converting the input SDF graph into an equivalent homogeneous SDF (HSDF) graph [1]. HSDF is a special case of SDF in which the production and consumption rates are identically equal to unity for all input and output ports of all actors. These rates are in terms of data values (*tokens*) per actor execution (*firing*). 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 [9]).

This complexity arises from the nature of *periodic schedules* of SDF graphs, which are used for static scheduling. A periodic schedule for an SDF graph is a schedule that produces no net change in the *buffer state* — i.e., the numbers of tokens that are queued on the buffers associated with the graph edges. The total number of actor firings in a periodic schedule can scale exponentially even for simple classes of SDF graphs [9]. Since each actor firing corresponds to a separate vertex in the HSDF version of an SDF graph, the SDF-to-HSDF transformation process can result in similar exponential growth.

Oh [10] develops a new single appearance schedule (SAS) approach for SDF graphs to jointly minimize data memory and code memory size. While this approach provides powerful analysis for software synthesis that is targeted to single processor architectures, it does not allow for simultaneous firings of multiple actors. In our context of hardware synthesis to FPGA targets, where each actor is mapped to a separate computing unit, allowing simultaneous firings is a key feature in maximizing the achieved throughput.

Stuijk [11] develops a systematic approach for exploring throughput and storage trade-offs for SDF graphs. This approach applies methods developed in [12] for determining

minimum storage requirements based on state-space analysis of buffer states. 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. Although this approach prunes the search space to reduce complexity, schedule simulation is still required in the search process, so again, worst case complexity is not polynomially bounded.

Wiggers [13] presents an algorithm with linear computational complexity to determine close-to-minimum buffer capacities for a given throughput constraint. However, this approach imposes a form of strictly periodic scheduling that requires a counter in every functional block, which leads to resource overhead in FPGA and other hardware-oriented implementations. Also, since Wiggers's approach assumes that execution will enter the required periodic steady-state only with the timely availability of sufficient starting tokens for every actor, it may not adequately handle irregular streaming inputs, where token arrival times are less predictable.

In contrast to the related prior work, we propose a heuristic algorithm with low polynomial time complexity that provides upper bounds on buffer requirements to guarantee throughput-optimal FPGA realizations of SDF graphs. Our approach focuses on the restricted class of tree-structured SDF graphs — that is, the input application model (*application graph*) must be in the form of an SDF tree. We emphasize that our algorithm is a heuristic only in the sense of the buffer sizes that are computed; in terms of achieved throughput performance, our approach guarantees optimality.

We first analyze relationships of firing patterns between actors and buffer requirements for the *two-actor SDF graph model* (*TASM*), which is a specialized form of SDF graph that we propose for efficient analysis of data communication on individual edges in a given SDF application graph. We then apply this two-actor firing pattern analysis repeatedly when traversing an application graph to determine buffer configurations that guarantee maximum achievable throughput.

In our buffer optimization scenario and our associated TASM analysis, we consider self-timed dataflow graph execution (e.g., see [14], [15]), which means that an actor is fired as soon as all of its input edges have enough tokens — that is, as soon as the number of tokens on each input edge e is at least $c(e_i)$. If each actor is mapped to a separate hardware resource, and the overhead of communication and synchronization between actors is negligible, then self-timed execution leads to the maximum achievable throughput (e.g., see [16], [15]). Moreover, this form of execution does not require any global schedule, and therefore storage, performance, and interconnect overhead associated with implementing a global schedule is avoided.

With predominantly coarse-grained dataflow actors (e.g., digital filters, and transform computations as opposed to adders and multiplers), and streamlined implementation of dataflow edges, one can reduce the relative overhead of inter-actor communication and synchronization significantly so that self-timed scheduling becomes an effective approach. This context of coarse-grain actors and streamlined edge implementation is the form in which we explore self-time implementation and associated buffer configuration strategies in this paper.

We first present precise definition and notations related to buffer analysis of SDF-based implementations. Using these concepts, we analyze the data transfer behavior on an SDF edge by the TASM model described earlier. Based on this analysis, we develop an algorithm for buffer analysis based on the TASM model, and we show an overall design flow for applying this algorithm for efficient synthesis of FPGA implementations. The proposed algorithm is implemented in 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 [5].

II. Background

A. Application representation

We represent a DSP application with a dataflow graph $G = (V, E)$, where each computational module is mapped to a vertex (*actor*) $v \in V$ and each directed edge $e \in E$ corresponds to a FIFO buffer for communicating data from the source actor $src(e)$ to the sink actor $snk(e)$ of e. We assume that the given dataflow model adheres to the assumptions of SDF, which require that the production and consumption rates of all actor output and input ports, respectively, are constant [1]. The SDF model is used widely in tools for DSP system design, and powerful analysis techniques have been develop for mapping SDF representations into various kinds of platforms (e.g., see [15]).

Given an SDF edge e, we represent the associated production rate of $src(e)$ by $p(e_i)$, and we represent the associated consumption rate of $snk(e)$ by $c(e_i)$. An SDF edge e also has associated with it a non-negative *delay*, denoted $del(e)$, which represents the number of *initial tokens* that reside on the corresponding buffer at the start of execution.

A necessary condition for executing (*firing*) an SDF actor v is that the number of tokens on every input edge e_{in} of v is greater than or equal to $c(e_{in})$. While v consumes $c(e_{in})$ tokens from each input edge e_{in} during its execution — i.e., during the execution of a single *invocation* or *firing* of the actor — it produces $p(e_{out})$ tokens onto each output edge e_{out}.

III. Target platform model

Since resource sharing is often avoided in FPGA implementation due to the relatively high cost of multiplexing and routing resources (e.g., see [17]), we assume that each computational block (SDF actor) is assigned to a dedicated set of FPGA logic cells without any sharing. Integrating resource sharing considerations into the developments of this paper is an interesting direction for future work, and may be useful in cases where resources are limited compared to the amount of required computation.

FPGAs provide two ways of implementing memory space between functional blocks — such memory space can be implemented using *block RAMs*, which provide dedicated

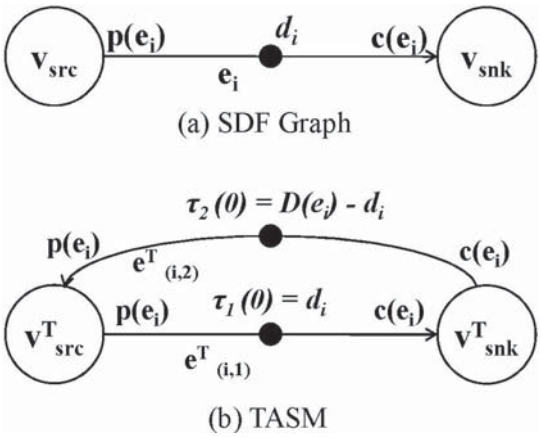

Fig. 1. An example of an SDF edge and its TASM model.

memory hardware within an FPGA, and *distributed RAM* using FPGA slices. The number of ports for reading (writing) data from (into) both forms of RAM is limited, and these limitations must be taken into account carefully for correct buffer management. In the Xilinx Virtex-II Pro FPGA, which we target in this paper, the number of ports is limited to two, and therefore, only a single pair of simultaneous read/write operations to each RAM subsystem is possible.

Our overall mapping approach therefore maps each actor in the SDF application model to a single actor (dedicated hardware resource) in the architecture model, along with a self-loop connection for that actor. Thus, we allow for concurrent execution of distinct actors, while serializing successive invocations of the same actor since such successive invocations must access the same memory ports for buffer access.

IV. Two-actor SDF graph model (TASM)

We assume a *static buffering* approach for SDF graphs, which means that for each SDF edge we allocate a fixed amount of memory space at compile time. We refer to the fixed amount of space that is allocated for an edge e_i as the *buffer size* of e_i, and we denote this buffer size by the symbol $D(e_i)$. For real-time implementation of SDF graphs, static buffering is often preferable due to its enhanced predictability and elimination of overhead due to dynamic memory allocation.

In this section, we introduce a model called the *Two-Actor SDF Graph Model* (TASM). For any edge $e_i \in E$ in an arbitrary SDF graph $G = (V, E)$, the TASM for e_i, analyzed in terms of in SDF semantics, accurately captures the token transfer across e_i as the enclosing SDF graph G executes under bounded memory. Also, TASM facilitates the formalization of our proposed synthesis approach, and its feature of computing buffer space requirements for throughput-optimal implementation.

A. Two-actor SDF graph model (TASM)

Suppose that edge e_i, shown in Fig. 1, is part of some arbitrary enclosing SDF graph $G = (V, E)$ (i.e., $e_i \in E$), and suppose that $src(e_i) = v_{src}$, $snk(e_i) = v_{snk}$, $del(e_i) = d_i$, and the production and consumption rates of e_i are denoted by $p(e_i)$ and $c(e_i)$, respectively. Suppose also that e_i is assigned a pre-specified buffer size $D(e_i)$. Then the TASM graph associated with e_i, which we denote by G_i^T, is defined as illustrated in Fig. 1(b). Here, $v_{src}^T = v_{src}$ and $v_{snk}^T = v_{src}$. Furthermore, $e_{(i,\,1)}^T$ connects v_{src}^T to v_{snk}^T with delay d_i, and $e_{(i,\,2)}^T$ connects v_{snk}^T to v_{src}^T with delay $(D(e_i) - d_i)$. The production and consumption rates for edges in the TASM graph are set as follows.

$$p(e_{(i,\,1)}^T) = c(e_{(i,\,2)}^T) = p(e_i) \tag{1}$$

and

$$c(e_{(i,\,1)}^T) = p(e_{(i,\,2)}^T) = c(e_i). \tag{2}$$

At any given time, buffer slots (cells in the memory that are allocated for the buffer) are categorized into two types based on whether they contain live data (*filled*) or whether they are available for storing new data (*empty* or *free*). The filled space in the buffer for e_i is modeled by $e_{(i,\,1)}^T$ in TASM. Thus as G_i^T executes, each token on $e_{(i,\,1)}^T$ represents a live token in the buffer associated with e_i in a corresponding execution of G. Since the source actor $src(e_i)$ can be fired only when e_i has enough free space to store all of the tokens produced by a firing of $src(e_i)$, each firing of $src(e_i)$ can be viewed as consuming $p(e_i)$ free cells from the buffer space available on e_i. Conversely, each execution of $snk(e_i)$ expands the free space on e_i by $c(e_i)$ cells. Hence, the free space on e_i can be modeled by the edge $e_{(i,\,2)}^T$ shown in Fig. 1(b), where each token on $e_{(i,\,2)}^T$ during an execution of G_i^T represents an empty cell in the buffer associated with e_i in a corresponding execution of G.

B. Modified self-timed execution (MSTE) in TASM

We use the self-timed execution model when mapping the input SDF graph into an FPGA implementation. Self-timed execution of SDF graphs can in general lead to execution periods (the patterns in which actors execute on the available resources) that are of exponential length in terms of the size of the of the graph (e.g., see [15]). Such exponential growth of execution periods can significantly complicate static analysis. To help address this difficulty, we add an additional firing rule, which we call the *MSTE firing rule*: actor v_{src}^T in G_i^T (see Fig. 1(b)) of the TASM model cannot be fired if

$$\tau_1(t) \geq max(p(e_i),\, c(e_i)), \tag{3}$$

where $\tau_j(t)$ represents the number of tokens on $e_{(i,\,j)}^T$ at time t for $j = 1, 2$. By imposing the MSTE firing rule, we obtain a modified form of self-timed execution, which we refer to in the remainder of this paper as *modified self-timed execution* (*MSTE*).

We have empirically observed that this additional firing rule usually results in only relatively minor deviations from self-timed execution. However, imposing the rule leads to a periodic execution pattern S_P that is defined by the repetition vector of G. More precisely, by S_P in this context, we mean a finite-duration schedule onto the disjoint subsets, r_{src} and r_{snk}, of FPGA resources that are occupied by the actors v_{src}

and v_{snk}, respectively. In other words, S_P can be viewed as a mapping

$$S_P : [0, 1, \ldots, (t_i - 1)] \times \{r_{src}, r_{snk}\} \rightarrow \{v_{src}^T, v_{snk}^T, v_{idle}\}, \quad (4)$$

where t_i is the length of the schedule (the period of the periodic pattern), and v_{idle} represents a void computation (idle resource). Note that even though S_P is formulated as a fully static schedule, it is implemented using our modified form of self-timed execution — i.e., the constraints imposed by our modified form of self-timed execution lead naturally to this kind of periodic pattern in the steady state. If t_s represents the time when this steady state pattern first emerges, (i.e., just after then point when the transient ends), then the schedules for all time intervals of the form $[(t_s+kt_i), (t_s+(k+1)t_i-1)]$, for $k = 0, 1, \ldots$, can be obtained by appropriately-shifted versions of the schedule defined by (4).

Thus, if $\vec{q}$ represents the repetitions vector of G, then one period of S_P contains $q[v_{src}]$ and $q[v_{snk}]$ firings of v_{src}^T and v_{snk}^T, respectively. This kind of periodic schedule helps to significantly reduce the complexity of performance analysis since the iterative dataflow execution is characterized by a relatively compact periodic structure.

C. Subperiods in TASM

The entire firing pattern in an iteration (i.e., a single execution in the periodic repetition) of S_P can be expressed as a sequence of *subperiod*s, where by a subperiod (*SP*), we mean a smaller firing pattern within S_P. From the additional firing rule that we introduce in our implementation model, we are able to constrain execution so that it becomes more structured, which leads to potential for more efficient static analysis. Fortunately, the constraints imposed by the MSTE firing rule do not impose significant performance limitations, which we will demonstrate in the experiments that we present in Section IX.

An *SP* is defined as the time period between two consecutive *breakpoints* of actor execution, where the breakpoints are derived from two key conditions. The first condition, which we denote by $c_1(t)$, is

$$c_1(t) = c_{1,a}(t) \text{ and } c_{1,b}(t), \quad (5)$$

where

$$c_{1,a}(t) = (\tau_1(t) \geq max(p(e_i), c(e_i))), \text{ and} \quad (6)$$

$$c_{1,b}(t) = (\tau_1(t) < p(e_i) + c(e_i)). \quad (7)$$

We say that the first condition, Condition 1, "holds" or "is true" at a given time instant θ if (6) is satisfied for $t = \theta$ (i.e., if both $c_{1,a}(\theta)$ and $c_{2,a}(\theta)$ hold). As we see from the MSTE firing rule, the source actor cannot be fired when Condition 1 is satisfied.

To introduce Condition 2, denoted by $c_2(t)$, it is useful to first define the following notion of *inter* and *intra* firing times of an actor. The set of inTRA firing times of an actor X, denoted by $TRA(X)$, is defined as the set of time instants during which actor X is executing. This set can be formulated as follows.

$$TRA(X) = \cup_{j=1}^{\infty} \{t \mid start(X, j) \leq t < end(X, j)\},$$

where $start(X, j)$ and $end(X, j)$ are the start and end time, respectively, of actor X's j-th firing in a periodic schedule.

Similarly, the set of inTER firing times of an actor X is defined as the set of time instants during which actor X is not executing ("idle"). This set can be expressed as the complement of the inTRA firing times of X with respect to the set $\mathbb{Z}^+$ of non-negative integers:

$$TER(X) = \mathbb{Z}^+ - TRA(X).$$

Condition 2 associated with the definition of breakpoints is defined as

$$c_2(t) = \begin{cases} \text{true}, & \text{if } t \in \left\{ TER(v_{src}^T) \cap TER(v_{snk}^T) \right\} \\ \text{false}, & \text{otherwise} \end{cases} \quad (8)$$

where v_{src}^T and v_{snk}^T are TASM actors in Fig. 1. This condition represents that breakpoints do not occur during the execution time of either actor in TASM — breakpoints occur only "between" executions of v_{src}^T and v_{snk}^T. Based on the two conditions, $c_1(t)$ and $c_2(t)$, the k-th breakpoint, denoted $BP(k)$, is defined by

$$BP(k) = min \{t \mid c_1(t) \wedge c_2(t) \wedge (t > BP(k-1))\}. \quad (9)$$

V. Properties of subperiods in TASM

As described in Section IV-B and IV-C , MSTE leads to efficient static analysis because an execution pattern under MSTE can be decomposed into a periodic pattern, and such a pattern can be further decomposed into a sequence of subperiods (smaller patterns). A subperiod can be more precisely defined as the time between successive breakpoints. For convenience in this discussion, let the greatest common divisor (GCD) of $p(e_i)$ and $c(e_i)$ be denoted by $g(e_i)$, and consider the following two mutually exclusive scenarios:

$$g(e_i) \neq min(p(e_i), c(e_i)), \quad (10)$$

and

$$g(e_i) = min(p(e_i), c(e_i)). \quad (11)$$

Under Scenario (10), we distinguish between two different types of subperiods that occur, and we refer to these types as SP_α and SP_β. Each of these two types consists of a fixed number of firings of v_{src}^T and v_{snk}^T. Thus, an iteration of S_P is a sequence of subperiods, where each subperiod in the sequence takes on one of two statically-known forms — SP_α and SP_β. The specific numbers of firings are summarized in Table I.

TABLE I
THE NUMBER OF FIRINGS OF v_{src}^T AND v_{snk}^T IN SUBPERIOD α AND β OF TASM

	# of firings	$p(e_i) \geq c(e_i)$	$p(e_i) < c(e_i)$
Type α	$f_{SP_\alpha}(v_{src}^T)$	1	$\lceil c(e_i)/p(e_i) \rceil$
	$f_{SP_\alpha}(v_{snk}^T)$	$\lfloor p(e_i)/c(e_i) \rfloor$	1
Type β	$f_{SP_\beta}(v_{src}^T)$	1	$\lfloor c(e_i)/p(e_i) \rfloor$
	$f_{SP_\beta}(v_{snk}^T)$	$\lceil p(e_i)/c(e_i) \rceil$	1

Here, $f_{SP_\lambda}(X)$ represents the number of firings of actor X that occur in a subperiod of type $\lambda \in \{\alpha, \beta\}$.

Under Scenario (11), there exists only one type of subperiod in S_P. In this case, $p(e_i)$ divides $c(e_i)$ or $c(e_i)$ divides $p(e_i)$, and it follows that the numbers of firings in Table I for the source and sink actors are the same between the rows corresponding to type α and type β. In other words, under Scenario (11), SP_α and SP_β are identical, and thus, execution proceeds based on only one type of subperiod.

In summary, there are in general two types of subperiods to consider — SP_α and SP_β, and these forms are identical under Scenario (11). Execution within S_P can always be broken down into a succession of subperiods, where each individual subperiod conforms to one of these two forms. This is established by the following two lemmas. Basic notation related to TASM, which is used in our formulation of these lemmas, is summarized in Fig. 1(b). Proofs of theorems and lemmas are omitted throughout the paper due to lack of space.

Lemma 1. *Suppose that we are given a TASM G_i^T under MSTE. If $p(e_i) \geq c(e_i)$, then in each subperiod, v_{src}^T has exactly one firing, and v_{snk}^T has either $\lfloor p(e_i)/c(e_i) \rfloor$ or $\lceil p(e_i)/c(e_i) \rceil$ firings.*

Lemma 2. *Suppose that we are given a TASM G_i^T under MSTE. If $p(e_i) < c(e_i)$, then in each subperiod, v_{snk}^T has exactly one firing, and v_{src}^T has either $\lfloor c(e_i)/p(e_i) \rfloor$ or $\lceil c(e_i)/p(e_i) \rceil$ firings.*

From Lemma 1 and 2, it follows that the numbers of firings of v_{src}^T and v_{snk}^T in a subperiod can be determined as shown Table I.

VI. THROUGHPUT ANALYSIS IN TASM

In this section, we analyze the pattern of actor firings in TASM and analyze the impact of allocated buffer sizes on the achieved throughput.

A. Firing pattern analysis

We begin with the following lemma, which relates tokens produced and consumed by the source and sink actors, respectively, in TASM.

Lemma 3. *Given a TASM G_i^T under MSTE, tokens produced by v_{src}^T in a subperiod are never consumed by v_{snk}^T in the same subperiod.*

Lemma 3 states that firing of v_{snk}^T is never *delayed* in a subperiod (i.e., it is not preceded by any idle time at the *start* of the subperiod). This is because v_{snk}^T does not need to wait for tokens produced from v_{src}^T in the same subperiod. While $(\tau_1(BP(k))$ tokens are always sufficient to avoid delaying v_{snk}^T during each subperiod k, v_{src}^T may be delayed (due to a value of $(\tau_2(BP(k))$ that is too small) if the allocated buffer size $D(e_i)$ is not sufficient. In other words, v_{src}^T can be delayed to wait for tokens on $e_{(i,\,2)}^T$ that must be produced by v_{snk}^T or equivalently, v_{src} waits until one or more firings of v_{snk} generate sufficient empty space in the buffer shown in Fig. 1(a). Hence, the firing pattern of v_{src}^T in a subperiod is in general a function of the allocated buffer size $D(e_i)$.

From the second breakpoint condition $c_2(t)$ in (8), the size of buffer($D(e_i)$) allocated on e_i of Fig. 1(a) can be represented by

$$D(e_i) = (\tau_1(BP(k)) + \tau_2(BP(k))) \text{ for all } k. \quad (12)$$

We first derive a buffer size that is sufficient to guarantee that firings of v_{src}^T are never delayed. This derivation is general in the sense that it holds in the absence of information about the execution times of v_{src}^T and v_{snk}^T (beyond the assumption that the execution times are constant). Thus, this execution pattern analysis is useful for applications in which actor execution times are known to be constant, but whose constant values are not known exactly. Furthermore, this analysis provides a foundation for computing more tight buffer size requirements in the presence of known (constant) execution times (as we show in Section VII).

Theorem 1. *Suppose that we are given a TASM G_i^T under MSTE, and suppose that the buffer size is given by*

$$D(e_i) = max(p(e_i),\, c(e_i)) + p(e_i) + c(e_i) - g(e_i). \quad (13)$$

Then firings of v_{src}^T are never delayed in any subperiod.

In the next two theorems, we derive buffer size levels that are sufficient to guarantee certain kinds of firing patterns for v_{src}^T and v_{snk}^T. These firing patterns are useful in throughput analysis.

Theorem 2. *Suppose that G_i^T is executed under MTSE; $p(e_i) \geq c(e_i)$; γ is an integer satisfying $0 \leq \gamma \leq f_{SP_\alpha}(v_{snk}^T)$; and*

$$D(e_i) = 2 * p(e_i) + (1 - \gamma) * c(e_i) - g(e_i). \quad (14)$$

Then in any given subperiod, there is exactly one firing of v_{src}^T. Furthermore, if n firings of v_{snk}^T precede v_{src}^T in a given subperiod, then $n \leq \gamma$. In other words, the single firing of v_{src}^T in a subperiod occurs after at most γ firings of v_{snk}^T.

Theorem 2 tells us how long a single firing of v_{src}^T is delayed in each subperiod when the buffer size for e_i in Fig. 1(a) is bounded.

Theorem 3. *Suppose that G_i^T is executed under MTSE; $p(e_i) < c(e_i)$; δ is an integer satisfying $0 \leq \delta \leq f_{SP_\beta}(v_{src}^T)$; and*

$$D(e_i) = (1+\delta)*p(e_i)+c(e_i)-g(e_i). \quad (15)$$

Then in any given subperiod, there is exactly one firing of v_{snk}^T. Furthermore, if n firings of v_{src}^T occur before the end of the v_{snk}^T firing in a given subperiod, then $n \leq \delta$. In other words, v_{src}^T is fired at most δ times in a subperiod before the single firing of v_{snk}^T completes.

Theorem 3 tells us, for a given bounded buffer size, how many times v_{src}^T can be fired independently of v_{snk}^T within a given subperiod. After firing v_{src}^T δ times in a subperiod, any remaining firings of v_{src}^T are delayed until v_{snk}^T completes its execution.

B. Saturated TASM systems

In this section, we assume that the execution times of actors are constant and known apriori, and we develop methods for throughput analysis of MSTE under this assumption.

We begin by defining some notation.

Definition 1. *Suppose that we are given a TASM G_i^T. Then the execution times of v_{src}^T and v_{snk}^T are denoted by T(v_{src}^T) and T(v_{snk}^T), respectively.*

Intuitively, T(v_{src}^T) and T(v_{snk}^T) give the time required for each actor to complete a single firing on r_{src} and r_{snk}, respectively. Our development of throughput analysis for MSTE also involves the following definition.

Definition 2. *Suppose that we are given a TASM G_i^T that executes under MSTE, and suppose that in each subperiod, the resource r_{src} operates without any idle time — that is, $S_P(t, r_{src}) = v_{src}^T$ for all $t \in 0, 1, \ldots, (t_i - 1)$. Then we say that G_i^T is* source-saturated*. Similarly, if r_{snk} executes without any idle time, then we say that G_i^T is* sink-saturated*.*

Clearly, since the net production and consumption rates of v_{src}^T and v_{snk}^T are balanced across S_P, it follows that the original SDF graph (Fig. 1(a)) executes at its maximum achievable throughput if G_i^T is source- or sink-saturated. This is summarized in the following property.

Property 1. *A TASM that is source-saturated or sink-saturated executes at its maximum achievable throughput when it executes under MSTE.*

Due to the additional firing rule of MTSE (see (3)), the execution of TASM under MTSE has the following property.

Property 2. *Suppose that we denote the total non-idle time of the resource r_η in a type λ subperiod by*

$$T_{SP_\lambda}(r_\eta) = f_{SP_\lambda}(v_\eta^T)*T(v_\eta^T). \quad (16)$$

Then G_i^T is either source- or sink-saturated if

$$T_{SP_\lambda}(r_{src}) \geq T_{SP_\lambda}(r_{snk}) \text{ for all } \lambda \in \{\alpha, \beta\} \quad (17)$$

or

$$T_{SP_\lambda}(r_{src}) < T_{SP_\lambda}(r_{snk}) \text{ for all } \lambda \in \{\alpha, \beta\}. \quad (18)$$

In particular, G_i^T can be neither source- nor sink-saturated under two *corner cases*, which we denote as corner case 1 (CC1) and corner case 2 (CC2). CC1 corresponds to the condition that the following two inequalities both hold:

$$T_{SP_\alpha}(r_{src}) \geq T_{SP_\alpha}(r_{snk}) \quad (19)$$

and

$$T_{SP_\beta}(r_{src}) < T_{SP_\beta}(r_{snk}). \quad (20)$$

Similarly, CC2 corresponds to the condition that (21) and (22) both hold:

$$T_{SP_\alpha}(r_{src}) < T_{SP_\alpha}(r_{snk}) \quad (21)$$

and

$$T_{SP_\beta}(r_{src}) \geq T_{SP_\beta}(r_{snk}). \quad (22)$$

Equation (19) means that r_{snk} has nonzero idle time in each α-type subperiod, while (20) holds if r_{src} has nonzero idle time in each β-type subperiod. Clearly, neither r_{src} nor r_{snk} is saturated in such a system.

The corner cases CC1 and CC2 represent limitations in our MSTE approach since our guarantee of maximal throughput, as given by Property 1, does not apply under these cases. However, we observe that CC1 and CC2 do not apply to a broad class of practical systems — in particular, systems that contain functional blocks that perform as bottlenecks, where by a "bottleneck", we mean a block whose computational complexity is dominant over other functional blocks. For example, in the dataflow-based 3GPP-Long Term Evolution (LTE) protocol application developed in [18], the FFT block can be observed to be a bottleneck.

In Section IX we present detailed experimental studies with three practical applications, all of which involve bottleneck actors and corresponding avoidance of the corner cases (CC1 and CC2) that prevent source- and sink-saturated execution.

VII. Analysis of saturated systems

Motivated by our discussion on bottleneck actors and the practical relevance of source- and sink-saturated systems, we develop in this section a detailed analysis of throughput-constrained buffer optimization for such systems. Throughout the remainder of this section, we assume that we are working with a source- or sink-saturated TASM — i.e., we assume that the corner cases CC1 and CC2 (defined in Section VI-B) do not hold.

Definition 3. *Suppose that we are given an SDF Graph $G = (V, E)$ that executes under MSTE, and suppose that the time duration of S_P (i.e., a single iteration of the periodic schedule) is denoted by t_i. Then by the throughput of an actor $v \in V$, which we represent by $\Phi(v)$, we mean the number of firings of v that execute per unit time. Since $q[v]$ firings of an actor v execute in each iteration of S_P, we have that*

$$\Phi(v) = q[v]/t_i \text{ for all } v \in V. \quad (23)$$

Furthermore, by the throughput of G_i^T, which we refer to as the *TASM throughput*, we mean the reciprocal of the time duration of S_P — i.e., $(1/t_i)$.

In this section, we show how to determine an upper bound on the buffer size required to execute G_i^T at its maximum achievable throughput. Henceforth, we refer to the reciprocal this maximum achievable throughput as t_{min}.

We remind the reader that although this analysis is developed for two-actor SDF graphs, the methods can be applied to arbitrary tree-structured SDF graphs, as described in Section I, by using them on each edge (and the underlying two-actor subgraph) separately and combining the results.

Property 3. *Suppose that we are given a TASM G_i^T. From the definitions of S_P, t_i, t_{min}, and actor throughput, we have that*

$$(\Phi(v_{src}^T) \geq q[v_{src}^T]/t_{min}) \text{ and } (\Phi(v_{snk}^T) \geq q[v_{snk}^T]/t_{min}) \Rightarrow t_i \leq t_{min}$$

From Lemma 3, firings of v_{snk}^T are never delayed in a subperiod. Also, from Theorem 1, firings of v_{src}^T are not delayed in a subperiod if $D(e_i)$ is set according to (13). Thus, under such a setting for $D(e_i)$, each actor in G_i^T is fired throughout a subperiod without any dependency on the other TASM actor. Hence, we establish the following lower bound on the throughput of an actor in G_i^T under the buffer size given by (13):

$$\Phi(v_\eta^T) \geq \frac{min_{\lambda \in \{\alpha,\beta\}} f_{SP_\lambda}(v_\eta^T)}{max(T_{SP_\alpha}(r_\eta), T_{SP_\beta}(r_\eta))} \text{ for } \eta \in \{v_{src}^T, v_{snk}^T\}. \quad (24)$$

If the execution times of v_{src}^T and v_{snk}^T are known, then it may be possible to exploit this knowledge to relax the buffering requirements, and thereby save resources on the target FPGA device. In particular, we can reduce buffering requirements if after applying the reduced buffer size given by Theorem 2 or Theorem 3 (based on whether $p(e_i) \geq c(e_i)$ or $p(e_i) < c(e_i)$, respectively), the resulting throughput given by (24) still meets the given throughput constraint.

In a given enclosing SDF graph $G = (V, E)$, the minimum achievable iteration period for S_P is given by

$$t_{min} = T(v_{btlneck}) * q[v_{btlneck}], \quad (25)$$

where $v_{btlneck} = max_{v \in V}\{v|q[v]*T(v)\}$. If an iteration of S_P completes exactly every t_{min} time units, then we can conclude that $v_{btlneck}$ is source- or sink-saturated, and the overall TASM throughput cannot be increased further.

VIII. Application to general tree-structured SDF graphs

Our TASM analysis can be applied iteratively to determine buffer sizes for all edges in an arbitrary tree-structured SDF graph. This assumption of a tree-structured graph is needed to ensure that the "extra (feedback) edges" added by the TASM models for different SDF graph edges do not "interact" (i.e., introduce new directed cycles in the overall graph model). Many practical SDF graphs or subsystem models are tree-structured, including models for multi-stage sample rate conversion, and various kinds of filterbanks, as well as the JPEG and OFDM transmitter applications that we examine in Section IX.

Algorithm 1 provides a systematic procedure for determining buffer sizes for an arbitrary, tree-structured SDF graph in a way that guarantees that the achieved performance will satisfy a given throughput constraint. The output of this procedure is a buffer size function $D : E \rightarrow Z_{pos}$, where Z_{pos} denotes the set of positive integers. The complexity of this algorithm is $O(E)$, which renders the approach practical for DSP and FPGA design tools.

Algorithm 1

```
INPUT : Tree-structured SDF graph G = (V, E)
      : Actor execution times T : V → Z_pos
OUTPUT: Buffer sizes, D : E → Z_pos
procedure TASM-BUFFERING(G)
    for each e ∈ E do
        p ← p(e); c ← c(e);
        t_src ← T(src(e)); t_snk ← T(snk(e))
        if p ≥ c then
            γ = f_α(snk(e)) − ⌈t_src/t_snk⌉
            D(e) ← apply Theorem 2 with γ
        else
            δ = ⌊t_snk/t_src⌋
            D(e) ← apply Theorem 3 with δ
        end if
    end for
end procedure
```

IX. Experimental results

We have implemented Algorithm 1 in the DIF environment [5], and applied it to three relevant signal processing applications — a CDtoDAT (compact disc to digital audio tape) sample rate converter, JPEG encoder, and DVB-T OFDM transmitter for digital video broadcasting. Using National Instruments LabVIEW FPGA 8.5, we have developed FPGA implementations of these three applications along with corresponding buffer size computation results from Algorithm 1.

LabVIEW is a graphical, dataflow-based programming environment for embedded system design. LabVIEW features for HDL (hardware description language) synthesis along with LabVIEW's dataflow orientation make the tool well-suited to FPGA-based design of signal processing applications. We have targeted the Xilinx Virtex II Pro P30 embedded in the National Instruments PCI-5640R digital system prototyping board to synthesize SDF-based application graphs with the buffer size functions computed by Algorithm 1. The base clock rate for our experiments is 40 MHz.

In the CDtoDAT application, the FIR filter in the first conversion stage becomes the bottleneck ($v_{btlneck}$ in (25))

TABLE II
SUM OF RESULT BUFFER DISTRIBUTION UNDER THE MAXIMUM THROUGHPUT(SAMPLES/CYCLE) AND ITS SYNTHESIS RESULT

		CDtoDAT	JPEG Encoder	DVB-T OFDM
Algorithm Result	Throughput	$5.6 * 10^{-3}$	.159	.191
	Buffer Sum	32	34112	9179
Synthesis Result	FPGA Slices	5438	8105	2810
	Block RAM	5	41	36
	18x18 MULT	5	41	36

of the system due to the high number of taps. Similarly, a discrete cosine transform (DCT) block in the JPEG encoder and the inverse FFT block in the DVB-T OFDM transmitter are bottlenecks for their respective applications.

Table II shows the results of Algorithm 1 and the associated synthesis results on targeted FPGA. Based on the synthesis results for the three applications, we verified that all of the solutions operate at the corresponding maximum achievable throughput levels, which correspond to the absence of idle time in the execution profiles of the resources that execute the associated bottleneck actors. Our results are therefore consistent with our theoretical results, which guarantee throughput optimality under the buffer sizes derived from Algorithm 1.

X. CONCLUSION

We have presented a novel algorithm to provide upper bounds on FPGA buffer distributions for throughput-optimal execution of synchronous dataflow graphs that are in the form of tree-structured, directed acyclic graphs. The resulting bounds can be employed directly as buffer sizes when mapping SDF graphs into digital hardware. A distinguishing aspect of our proposed algorithm is that it has low polynomial time complexity, which makes it especially useful for rapid prototyping and for implementation of large scale or heavily multirate designs. Our work appears promising for integration into high-level design processes for FPGA-based DSP system implementation, as our experiments with the LabVIEW FPGA demonstrate.

REFERENCES

[1] E. A. Lee and D. 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] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for dsp using ptolemy," *Journal of VLSI Signal Processing*, vol. 9, pp. 7–21, 1995.

[3] W. Sung, M. Oh, C. Im, and S. Ha, "Demonstration of codesign workflow in peace," in *in Proc. of International Conference of VLSI Circuit, Seoul, Koera*, 1997.

[4] J. Buck and R. Vaidyanathan, "Heterogeneous modeling and simulation of embedded systems in el greco," in *CODES '00: Proceedings of the eighth international workshop on Hardware/software codesign*. New York, NY, USA: ACM, 2000, pp. 142–146.

[5] 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.

[6] C. Hsu, J. L. Pino, and S. S. Bhattacharyya, "Multithreaded simulation for synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, Anaheim, California, June 2008, pp. 331–336.

[7] R. Reiter, "Scheduling parallel computations," *Journal of the Association for Computing Machinery*, October 1968.

[8] A. Dasdan, A. Dasdan, R. K. Gupta, and R. 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.

[9] 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.

[10] H. Oh, N. D. Dutt, and S. Ha, "Single appearance schedule with dynamic loop count for minimum data buffer from synchronous dataflow graphs," in *CASES*, 2005, pp. 157–165.

[11] E. Stuijk, M. Geilen, and T. Basten, "Exploring trade-offs in buffer requirements and throughput constraints for synchronous dataflow graphs," in *In DAC*. ACM Press, 2006, pp. 899–904.

[12] M. Geilen, T. Basten, and E. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *in Proceedings of the Design Automation Conference*. ACM, 2005, pp. 819–824.

[13] M. Wiggers, M. Bekooij, P. G. Jansen, and G. J. M. Smit, "Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure," in *CODES+ISSS*, 2006, pp. 10–15.

[14] E. A. Lee and S. Ha, "Scheduling strategies for multiprocessor real time DSP," in *Proceedings of the Global Telecommunications Conference*, November 1989.

[15] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*, 2nd ed. CRC Press, 2009.

[16] S. Y. Kung, P. S. Lewis, and S. C. Lo, "Performance analysis and optimization of VLSI dataflow arrays," *Journal of Parallel and Distributed Computing*, pp. 592–618, 1987.

[17] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "Fpga pipeline synthesis design exploration using module selection and resource sharing," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.

[18] 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, pp. 1510–1513.