In Proceedings of the International Heterogeneity in Computing Workshop, pages 66-77, Anchorage, Alaska, May 2011. Published through the IEEE Computer Society Press as part of the IPDPS CD-ROM.

A Model-based Schedule Representation for Heterogeneous Mapping of Dataflow Graphs

Hsiang-Huang Wu, Chung-Ching Shen, Nimish Sane, William Plishker, Shuvra S. Bhattacharyya
Department of Electrical & Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland, USA
{hhwu, ccshen, nsane, plishker, ssb}@umd.edu

***Abstract*—Dataflow-based application specifications are widely used in model-based design methodologies for signal processing systems. In this paper, we develop a new model called the *dataflow schedule graph* (DSG) for representing a broad class of dataflow graph schedules. The DSG provides a graphical representation of schedules based on dataflow semantics. In conventional approaches, applications are represented using dataflow graphs, whereas schedules for the graphs are represented using specialized notations, such as various kinds of sequences or looping constructs. In contrast, the DSG approach employs dataflow graphs for representing both application models *and* schedules that are derived from them.**

Our DSG approach provides a precise, formal framework for unambiguously representing, analyzing, manipulating, and interchanging schedules. We develop detailed formulations of the DSG representation, and present examples and experimental results that demonstrate the utility of DSGs in the context of heterogeneous signal processing system design.

***Keywords*-dataflow graphs, heterogeneous computing, models of computation, scheduling.**

I. Introduction

Dataflow models of computation are widely used for expressing the functionality of digital signal processing (DSP) applications (e.g., see [1]). In DSP-oriented dataflow models of computation, applications are modeled as directed graphs, where vertices (*actors*) represent computational modules for executing (or *firing*) tasks, and edges represent first-in-first-out channels for storing data values (*tokens*), and imposing data dependencies between actors. Whenever an actor fires, it consumes and produces tokens from its input and output edges, respectively.

Scheduling has been studied extensively in the context of dataflow-based modeling of DSP systems. Dataflow graph scheduling involves assigning actors to processors, and sequencing subsets of actors that share common processing resources. For dataflow scheduling of DSP systems, a "processor" in this context is typically taken to be a hardware resource on which execution is time-multiplexed by actors that are assigned to it. In addition to ensuring that dataflow graph dependencies are respected, scheduling is often geared towards exploiting parallelism (performance improvement) and efficient memory utilization (buffer management). Given the fundamental role of scheduling in dataflow-based design flows, and its heavy impact on key implementation metrics, a wide variety of techniques has evolved over the years and continues to evolve for scheduling DSP dataflow graphs. Such techniques target objectives such as buffer optimization [2], joint code and data minimization [3], quasi-static scheduling [4], adaptive scheduling [5], [6], and throughput optimization [7].

As the range of dataflow graph scheduling techniques continues to expand, based on the heterogeneity of application modeling styles and implementation objectives, and the increasing degree of dynamics in applications, it becomes increasingly important to develop a common representation for modeling and working with dataflow schedules. Such a representation is desirable to enable systematic reuse of design tool code, analysis techniques, and back-end implementation methodologies across various scheduling strategies. Furthermore, a formal representation helps to integrate different scheduling techniques so that they can be mixed and matched across different subsystems of a design based on characteristics and objectives associated with those subsystems.

In this paper, we address this problem by introducing a formal framework, called the *dataflow schedule graph* (DSG), for precisely representing, analyzing, manipulating, and interchanging schedules. We have designed the DSG representation with two major objectives — 1) it should be rooted in formal dataflow semantics, and 2) it should accommodate a wide range of schedule classes, including static, quasi-static, and dynamic schedules, as well as both sequential and parallel schedule formats. Furthermore, because they are based on the same dataflow semantic framework as the application representations from which the schedules are derived, DSGs can naturally represent structures in which schedules are adapted dynamically (e.g., in response to changes in input data characteristics).

II. Related Work

A number of dataflow schedule representations have been explored previously. The *generalized schedule tree* (*GST*) representation provides a tree-based representation of arbitrary looped schedules [8]. A novel schedule format based on dynamic loop counts that is geared towards SDF buffer memory minimization is developed in [9]. The *interprocessor communication graph* and *synchronization graph* models provide dataflow-based schedule representations for parallel schedules of homogeneous SDF (HSDF) graphs [10]. HSDF is a restricted form of SDF in which the dataflow rate on each input and output port is always equal to 1 [11].

A distinguishing characteristic of our proposed DSG representation is that it is both dataflow based, and capable of handling dynamic schedule structures as well as dynamic dataflow application models. This is in contrast to execution-sequence based representations, which can usually be characterized formally but lack dataflow semantics and are often restricted to static schedules.

The most closely related modeling technique is the synchronization graph model. In this model, self-timed multiprocessor schedules are represented as interacting dataflow graph cycles, where each cycle corresponds to the periodic execution of the actors that are assigned to a given processor [10]. A significant body of theory and algorithms has been developed for this model. We are therefore motivated to generalize the synchronization graph concept beyond self-timed schedules, and HSDF graphs.

The DSG can be viewed as such a generalization. The DSG model can represent dynamic schedules, which can be applied to static or dynamic application models to improve flexibility (e.g., load balancing robustness or data dependent control structures). Furthermore, the model is fully based on dataflow principles, which together with its accommodation of dynamic dataflow semantics, allows for integration with dynamic parameter control methods for dataflow graphs, such as those provided by parameterized dataflow [5] and scenario-aware dataflow [6].

The DSG representation can be used in conjunction with existing task graph scheduling techniques, such as those developed in [12], [13], [14], [15], [16]. For example, the DSG can be used to model the sequencing structures derived by the scheduling techniques (e.g., as a standard interface for code generation) or to bridge subsystems that are scheduled using different techniques. Indeed, exploring the optimized integration of DSG based schedule control with new and existing task graph scheduling techniques is an interesting direction for further investigation, and one that is especially relevant in the area of heterogeneous computing systems.

III. Core Functional Dataflow

For concreteness, we develop the DSG in the context of a specific form of dataflow — the *core functional dataflow* (*CFDF*) model of computation, which can be viewed as a deterministic sub-class of *enable-invoke dataflow graphs* [17]. CFDF is a highly expressive (Turing complete), dynamic dataflow model. In Section XI, we discuss how the DSG model can be adapted to other forms of dataflow (beyond CFDF).

In CFDF, actors are specified as sets of *modes*, where each mode has a fixed production and consumption rate associated with each input and output port, respectively. Each actor has an associated *current mode*, which is maintained as part of its state. When an actor is invoked, it executes its current mode, produces and consumes data (as in other dataflow models), and updates its current mode. Since different modes of an actor can have different production and consumption rates, dynamic dataflow can be modeled flexibly in CFDF.

A distinguishing aspect of CFDF (and the non-deterministic superset EIDF) is that *separation* of enable and invoke functionality for actors is defined as a first class characteristic of the model. Specifically, each actor has an associated *enable* function, which can be called at any time between firings (e.g., by a run-time scheduler), and returns a Boolean value indicating whether or not there is sufficient data available on the actor input ports to fire (invoke) the actor in its current mode. Since such an isolated enable check is available, the invoke function of an actor assumes that sufficient data is present, and reads its input data without blocking reads.

In the implementation of dataflow tools, functionalities corresponding to the enable and invoke methods are often interleaved — for example, an actor firing may have computations that are interleaved with blocking reads of data that provide successive inputs to those computations. In contrast, there is a clean separation of enable and invoke capabilities in EIDF. This separation helps to improve the predictability of an actor invocation (since availability of the required data can be guaranteed in advance by the enable method), and in prototyping efficient scheduling and synthesis techniques (since enable and invoke functionality can be called separately by the scheduler). This separation also leads naturally to a concept of *guarded execution*, whereby an actor firing is conditionally executed depending on whether or not it is enabled.

IV. The Dataflow Schedule Graph Representation

Given a CFDF representation G_A of an application, a *dataflow schedule graph* (*DSG*) is a dataflow graph that satisfies certain technical constraints (described later in this section), and represents the time-multiplexed execution of G_A across a set of hardware resources. Here, a hardware resource represents an arbitrary computational resource, such as a processor core, dedicated accelerator or FPGA subsystem, that executes actors sequentially. Constraints imposed on the DSG ensure that each hardware resource can execute at most one actor from G_A at any given time. Tokens that flow along edges of the DSG serve to enable actors for execution (as it becomes their turn to execute). DSG tokens can also contain values that are manipulated and queried during execution of the DSG to achieve various forms of data- or parameter-dependent schedule control.

In DSGs, special actors, called *schedule control actors* (*SCAs*) and *reference actors* (*RAs*), are selected or developed as an integral part of the schedule modeling framework. In contrast to conventional dataflow actors, which represent functional components from the original application specification (*application actors*), *SCAs* are dataflow actors that are dedicated to coordinating control flow in derived schedules. On the other hand, *RAs* can be viewed as "pointers" to application actors. These pointers are equipped with optional auxiliary computations. Intuitively, an RA represents a scheduling "wrapper" that specifies the computation that is executed when the corresponding actor is "visited" during schedule execution. The simplest form of RA is one that simply performs a guarded execution of the actor that it points to. However, more capabilities can be incorporated into RAs using the optional auxiliary computations mentioned above.

V. Reference Actors

An RA has a single input port and a single output port. An RA is a homogeneous synchronous dataflow actor in the enclosing DSG — that is, it consumes a single token on each firing from its input, and produces a single token on its output.

Given an RA A, we represent the application graph actor pointed to by A with the symbol $ref(A)$, and we refer to $ref(A)$ as the *referenced actor* of A.

As illustrated in Figure 1, an RA A consists of two functions pre_A and $post_A$, which are executed, respectively, before and after the *guarded execution* phase of A. This guarded execution phase, represented by the block labeled "guarded firing" in Figure 1, represents the guarded execution of A in terms of CFDF semantics (see Section III).

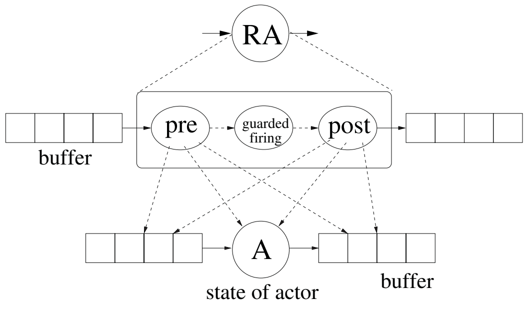


Figure 1. The internal structure of an RA.

We refer to the functions pre_A and $post_A$ as *subfunctions* of the enclosing RA. Intuitively, the RA subfunctions provide a mechanism to process and manipulate data that is used throughout the graph to control execution of actors (e.g., to facilitate conditional execution or data dependent iteration in various parts of the graph). The data manipulated by RA subfunctions is encapsulated within the DSG tokens that are produced and consumed by the enclosing RA.

To clarify the operational structure of DSGs, it is useful to emphasize that the tokens flowing on a DSG are strictly for schedule control purposes. Furthermore, because actors in the application graph are allowed to execute only when they have sufficient data (as specified by the CFDF enabling conditions), and CFDF is a deterministic dataflow model, schedule control by DSGs does not violate determinacy — such control only dictates how actors are time multiplexed when they are mapped to the same hardware resource.

RAs can contain internal state. Such local (actor-specific) state is widely known to be compatible with dataflow representations since in dataflow graphs, state can be modeled as self loops with delays (initial tokens) [11], [18]. Thus, the use of state in RAs does not violate our ability to interpret DSGs as genuine dataflow representations.

The following categories of data can be used as inputs in RA subfunctions:

- The value represented by the current DSG token — i.e., the DSG token that is consumed by the enclosing RA firing (pre_A only). This value can be of any type. The type is a design issue of the particular DSG control structure that is being developed for a specific schedule or the particular class of control structures that is being targeted by a particular scheduling tool.
- The state of the enclosing RA.
- The state of the referenced actor.

The following categories of data serve as outputs for (i.e., can be modified by) RA subfunctions:

- The state of the enclosing RA.
- The value of the token that is produced by the RA ($post_A$ only).

Firing of an RA involves the following sequence of steps:

1) The RA consumes a token from its input edge. This token is passed as input to pre_A, which executes, and updates the state of RA.
2) A guarded execution of ref_A is carried out. That is, ref_A is fired once if it is enabled.
3) An execution of $post_A$ is carried out. This execution operates on the state of the RA. The output value from this execution is produced as the output of the RA firing.

The general purpose of pre_A and $post_A$ is to manipulate DSG tokens. The values of DSG tokens, in conjunction with SCAs, contribute to overall schedule control. Computations in pre_A and $post_A$ are optional. For example, an RA can simply execute the referenced actor unconditionally, maintain no internal (RA) state, and pass input DSG values from input to output without modification. Such "lightweight" RAs are typical in the construction of static scheduling structures, as well as in dynamic structures where dynamic schedule control is managed by SCAs. When code is generated from DSGs, such lightweight RAs can easily be detected and "optimized away" so that they do not result in run-time overhead.

An example of a non-lightweight RA is one that updates DSG tokens with estimates of the amount of energy or execution time taken by the associated firings. Such information can then be used by the enclosing DSG to adapt overall schedule control — e.g., when the DSG is embedded within a parameterized dataflow system or other kind of reconfigurable dataflow graph framework (e.g., see [5], [6]).

VI. Schedule Control Actors

To model dynamic scheduling structures, SCAs generally play an important role in conjunction with RAs. An SCA is an actor that can have any positive number of input ports and any positive number of output ports. In other words, an SCA must have at least one input port and output port, and may have any number of additional input or output ports. The dataflow behavior of an SCA exhibits the following *lumped homogeneous synchronous dataflow* (*LHSDF*) condition: for every firing f of an SCA C, we have that $n_c = n_p = 1$, where n_c represents the total number of tokens consumed by C across all input ports during f, and n_p represents the total number of tokens produced across all output ports during f.

Note that an SCA C can have internal state, and if we model that state as a self-loop edge for C, then this edge is treated independently of the LHSDF condition — i.e., such a self-loop edge is a standard HSDF edge whose dataflow does not "count towards" the values of n_c and n_p.

A token in a DSG can be interpreted loosely as an "actor level program counter" for a given target processor. The LHSDF condition for SCAs along with the HSDF semantics of RAs guarantee that there is only one such program counter (thread of control) that is "demanded of" each target processor. This ensures that the schedule execution modeled by the DSG conforms to the assumption that individual target processors execute actors sequentially.

Note that while our proposed DSG model is used to model schedules for CFDF graphs, SCAs and hence DSGs do not necessarily conform to CFDF semantics. The primary requirement for SCAs in the context of the associated actor level program counter concept is most naturally captured by LHSDF semantics as opposed to CFDF.

We introduce several types of SCA actors that will be used in this paper. Table I summarizes properties of these actors. The *loop* actor has two pairs of inputs and outputs. One pair is used to perform computations within the loop repeatedly, while the other pair is used for conditionally branching into and exiting the loop based on certain control conditions. Since there is only one DSG token, execution always proceeds unambiguously either inside or outside the loop.

SCA actors can be paired with other SCA actors to provide special control functions that involve their coordination. For example, *if* and *fi* provide DSGs with the capability of selecting computations conditionally. The number of outputs for a given *if* actor must match the number of inputs to the corresponding *fi* actor to provide conditional selection of the computations that are enclosed by the matching *if* and *fi* pair.

The pair *snd* and *rec* is used for interprocessor communication and synchronization in concurrent DSGs (CDSGs), which are discussed further in Section VIII.

Table I
Examples of SCAs.

SCA	# of inputs	# of outputs
loop	2	2
if	1	≥ 2
fi	$\geq$2	1
snd	1	2
rec	2	1

VII. Sequential Dataflow Schedule Graphs

A DSG for a single-processor schedule represents the time-multiplexed (sequential) execution of a set of actors on a single processing resource. Execution of the DSG models the evolution of actor firings in the associated sequential schedule. To preserve this sequential execution property, a *sequential DSG* (SDSG) imposes the restriction that *at most one token* can be present in the entire DSG at any given time. This requirement formally captures the interpretation of DSG tokens as actor level program counters in the context of single-processor schedules. Just as the program counter in a conventional processor "points to" a single instruction at any given time, the unique SDSG token points to a single SDSG actor, which is the next actor to execute.

For example, consider the class of single appearance schedules for SDF graphs [3]. These schedules are represented in terms of *looped schedules* such that each actor appears exactly once, implying, for example, minimal code size under inline implementation. For example, the looped schedule $(3(2ab)c)$, involving 3 actors a, b, c, and 2 loops represented by the two nested, parenthesized terms, represents the firing sequence *ababcababcababc*.

To demonstrate SDSGs for single appearance schedules, we apply the *loop* SCA that was introduced in Section VI. Figure 2(a) shows an SDF graph (G_A) and an associated single appearance schedule $(A(2B)C)$. A simple SDSG (G_S) is shown in Figure 2(b). In this example, $loop_1$, which is an instance of the *loop* actor, implements an outer loop that models a finite *blocking factor* J. This blocking factor value gives the number of times that the schedule is to be repeated. If the schedule is to be repeated indefinitely ($J = \infty$), then $loop_1$ should be removed, and the output of R_C should be connected directly to R_A.

The actor $loop_2$, which is also an instance of the *loop* SCA defined in Section VI, implements control for an inner loop that corresponds to the nested subschedule $(2B)$. A token in this SDSG does not carry any values; it simply points to the next actor in the SDSG that is to be executed.

The "D" symbols on the graph in Figure 2 correspond to *delays*, and are implemented as initial tokens in the graph. Functionally, a delay corresponds to the z^{-1} operator in signal processing.

Execution of the SDSG shown in Figure 2(b) proceeds as follows. The delay (initial token) on the edge $(R_C, loop_1)$ causes execution to begin with a firing of $loop_1$. This actor $loop_1$ has one input port, one output port, and an internal state that maintains a loop iteration count n_o, which corresponds to the number of remaining schedule iterations, and is initialized to the blocking factor value J. Each time $loop_1$ fires, it first checks the value of n_o. If $n_o = 0$, then the firing completes with an output token produced on the output edge that is connected to END. On the other hand, if $n_o > 0$, then the value of n_o is decremented, and the firing completes with a token produced on the output edge that is connected to R_A.

This token has the effect of passing processor control to R_A, which then fires the referenced actor A once and passes control (through its output token) to $loop_2$.

The actor $loop_2$ has two input ports *in1* and *in2* and two output ports *out1* and *out2*, as shown in Figure 2(b). $loop_2$ also has a state variable n_i, which maintains the number of iterations remaining in the current inner loop invocation.

When $loop_2$ consumes a DSG token from *in1*, it resets n_i to 2, and produces an output token on *out1* to enable R_B. On the other hand, when $loop_2$ consumes its input from *in2*, it first decrements the value of n_i. If after this decrement operation $n_i > 0$, then it again produces an output token on *out1*; otherwise, it produces an output token on *out2*, which effectively exits the inner loop, and passes control to R_C.

Actors R_B and R_C, like R_A, operate by consuming a single token each from their unique input edges, firing their associated referenced actors, and producing a single output token on their unique output edges. In the case of R_C, the output token produced has the effect of passing control to the next invocation of the outer loop iteration control.

We emphasize that under correct operation, an SDSG contains at most one token. Thus, for an enabled SCA that has multiple input edges, there is never ambiguity about which input edge the next firing will consume data from — the SCA will simply consume the input token from the unique edge that has a nonzero buffer population.

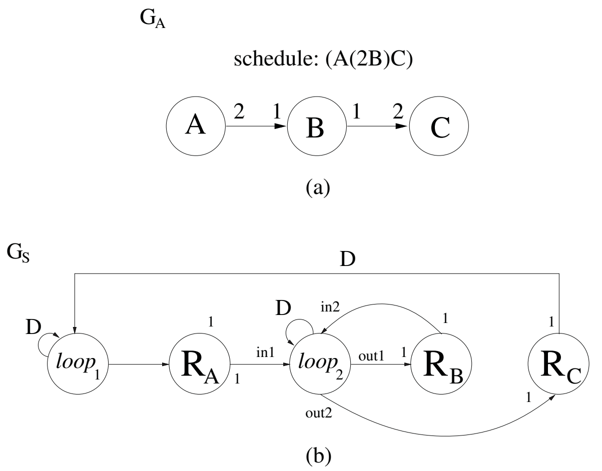


Figure 2. (a) An SDF graph (b) A design example of an SDSG for the single appearance schedule $(A(2B)C)$.

VIII. Concurrent Dataflow Schedule Graphs

Efficient parallel computation is an important motivation for use of dataflow graphs in many implementation contexts. For this purpose, the concept of the DSG can be naturally extended to handle concurrent execution of multiple SDSG "threads". Multiple SDSGs can be integrated to execute concurrently through the use of a special kind of actor called an *inter-SDSG coordination actor* (*ICA*). We refer to the resulting class of communicating, concurrent SDSGs as *concurrent DSGs* (*CDSGs*).

Two specific ICAs are snd and rec, which perform communication and associated synchronization of data that is passed between different processors. As shown in Figure 3, snd and rec both have one pair of input and output ports each — IN_{PC} and OUT_{PC} — for the execution-enabling SDSG token (i.e., the token that is analogous to a program counter or "PC", as described in Section VII). Additionally, the snd actor has a second output port that is used to send data to another processor, and similarly, the rec actor has a second input port that is used to receive interprocessor communication (IPC) data. We refer to these output and input ports as OUT_{IPC} and IN_{IPC}, respectively.

Every instance of a snd actor is paired with a corresponding rec actor in the sense that the OUT_{IPC} port of each snd actor is connected to the IN_{IPC} port of the corresponding rec actor. The snd represents the communication of a single token, including any necessary synchronization functionality (e.g., checking for available buffer space) from the sending processor to the processor on which the corresponding rec actor resides. Similarly, the rec represents receipt of a single token, including any associated synchronization functionality (e.g., to check whether the corresponding interprocessor communication buffer is non-empty before reading).

In general, the synchronization and data communication features of the rec and snd actors can be decoupled into more specialized ICAs that separately perform communication and synchronization. Such decoupling of synchronization and IPC operations can lead to opportunities for significantly reduced synchronization overhead (e.g., see [10]). Design and application of ICAs for such decoupled synchronization and IPC is a useful direction for further work.

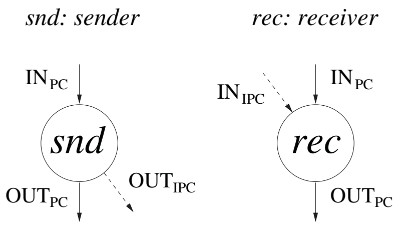


Figure 3. The snd and rec actors.

Figure 4 illustrates an HSDF application graph, and a partitioning of this graph across four processors. Figure 5 illustrates a CDSG representation of a multiprocessor schedule that is based on this partitioning result. In Figure 5, the schedule for each processor is embedded within an infinite loop to achieve an iterative execution of indefinite duration, which is a common execution format for DSP dataflow graph applications. Such infinite loops can easily be replaced by finite-iteration loops if needed by appropriate reconfiguration of the four loop SCAs.

Recall that the "D" symbols in our dataflow graph drawings correspond to delays, which are equivalent to initial tokens. Note also that each of the four concurrent SDSGs in Figure 5 has an edge directed from the last actor in the associated actor chain back to the first actor, which is a loop actor. This "feedback edge" represents the transfer of execution from the end of a given loop iteration on the processor back to the beginning of the next iteration. The delay on each of these feedback edges indicates that the execution on the given processor starts with the loop actor.

Each edge in Figure 4 that crosses the boundary of two processors can be viewed as an *interprocessor communication edge* (*IPC edge*), and is mapped to a corresponding pair of snd and rec actors in the CDSG of Figure 5. For example, the edge (E, I) in Figure 4 represents an IPC edge between Processor 1 and Processor 4. In the CDSG, this IPC edge is implemented by snd_1 and rec_4, which are connected, respectively to the output of the reference actor for E and the input of the reference actor for I.

In summary, the CDSG provides a formal, dataflow-based representation for modeling multiprocessor schedules of dataflow application graphs. Although other representations exist for managing schedules, the CDSG provides a novel combination of features — in particular, 1) full adherence to dataflow semantics, which helps to unify the model with the associated application representation, and 2) flexible integration of control constructs (through SCAs), which

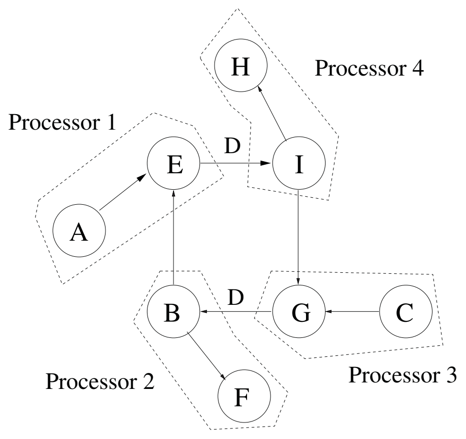


Figure 4. An application graph and a partitioning of the graph across four processors.

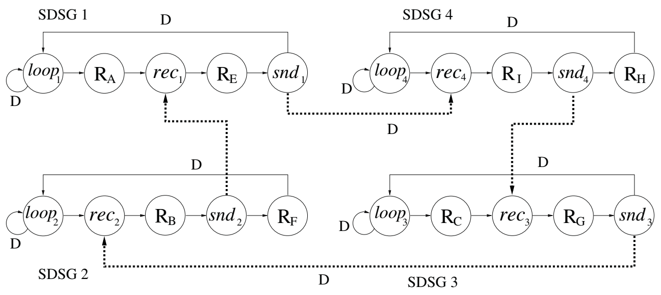


Figure 5. A CDSG representation of a multiprocessor schedule that corresponds to the partitioning result shown in Figure 4.

allows for modeling of a wide range of static, quasi-static and dynamic schedules.

IX. Adaptive Dataflow Schedule Graphs

A major benefit of the SDSG model is that in addition to accommodating static schedules, it provides a common, formal framework for representing a wide variety of dynamic dataflow schedules — i.e., schedules in which firing sequences are adapted dynamically, based on characteristics of the input data or operating environment.

We refer to an SDSG model of a dynamic dataflow schedule as an *adaptive dataflow schedule graph* (*ADSG*). Since ADSGs form a subclass of SDSGs, an ADSG can contain at most one token across all of its edges at any given time.

As a simple example, consider the dataflow-based `if-then-else` construct illustrated in Figure 6(a). All actors in Figure 6(a) produce and consume one token each except for the `switch` (S_W) and `select` (S_E) actors. Although the `switch` and `select` actors are commonly associated with the Boolean dataflow model [19], they can be mapped conveniently into CFDF semantics [20].

Both S_W and S_E consume the Boolean token produced by actor E to determine whether Path 1 or Path 2 will be followed subsequently. Although the path is determined at run time, a schedule for each path can be determined at compile time — $(AES_W BS_E D)$ and $(AES_W CS_E D)$ are schedules corresponding to Path 1 and Path 2, respectively.

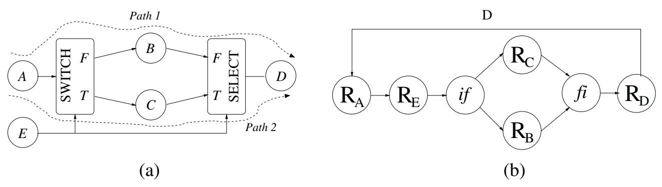


Figure 6. (a) A dataflow-based `if-then-else` construct. (b) An adaptive DSG for this construct.

Figure 6(b) shows a design example of an ADSG for the application graph shown in Figure 6(a). In other words, Figure 6(b) shows an ADSG model of a specific quasi-static schedule for the application graph in Figure 6(a).

Intuitively, the cycle in Figure 6 that encapsulates the actors (with feedback edge from R_D to R_A) models an infinite, quasi-periodic schedule.

In this ADSG, the output token that is produced by R_E encapsulates the data value that is produced by the corresponding firing of E. This is different from the DSG tokens in our earlier examples, where the tokens carried only control (enabling) information and had no values associated with them. The data encapsulation at the output of R_E can be ensured by the *post* function associated with R_E.

The `switch` actor, modeled by the SCA if, examines the output value v from E (through the DSG token that encapsulates its value), and produces a token on one of its output edges depending on whether v is `true` or `false`. This output token, like all other tokens in this DSG except for those at the output of R_E, does not have any associated data value.

The RAs R_B and R_C are "minimal" RAs that simply perform guarded executions of their associated referenced actors. By design of the quasi-static schedule that is modeled by the enclosing DSG, the enabling conditions for these guarded executions will be satisfied whenever the corresponding reference actors are fired.

On each firing, the SCA *fi* consumes the token from its unique, non-empty input edge (which is determined by the "output path" taken by the preceding invocation of if), and passes control to the RA R_D.

X. Experimental Results

Heterogeneous computing systems integrate different kinds of hardware and software to work together based on the given application requirements. Benefits of heterogeneous computing are often achieved at the expense of ad-hoc, error prone integration processes due to diverse code bases and the lack of unifying formal models. The DSG representation developed in this paper helps to alleviate this integration problem, leading to more systematic design and implementation of solutions that leverage heterogeneous computing platforms.

In this section, we demonstrate through design examples that the DSG is an efficient schedule representation, which provides robustness and flexibility to the back-end of dataflow-based design processes. Our experiments examine the application of DSGs to improving simulation performance of dataflow graphs, as well as to improving the processes of hardware mapping and software implementation from dataflow graphs. Overall, the experiments show the utility of DSG-based design and implementation across a heterogeneous variety of platforms.

A. Simulation Time Improvement

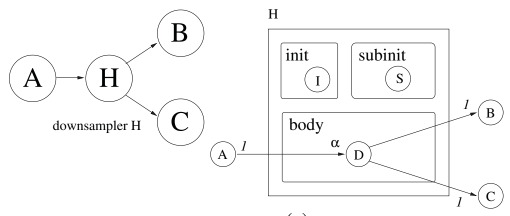
High level system simulation is a useful application of dataflow graphs in DSP system design. Simulation time for complex dataflow models is often dominated by the computation time of the schedule [21]. For some applications, this overhead can be reduced with well-designed quasi-static schedules, which trade off relatively large amounts of static schedule computations with relatively small amounts of run-time schedule adjustments [10].

In this section, we apply the DSG representation to model a quasi-static schedule, and demonstrate improvement in simulation time achieved by this schedule.

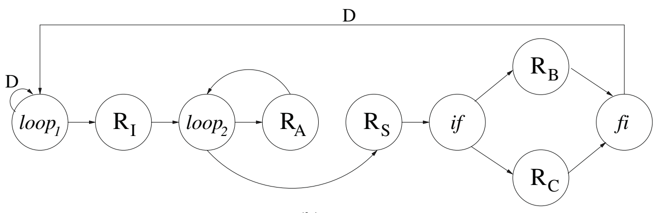
Figure 7(a) demonstrates a Boolean-parameterized down-sampler H, which can be used to achieve dynamic changes in sampling rate for different parts of a data stream. The actor H consumes α tokens and then sends one of the consumed tokens to either actor B or C, as determined by the value of the actor's `selection` parameter. The values of parameter α and the selection parameter are generated by actor I and S, which are enclosed within the subsystems labeled `init` and `subinit`. The operation of these subsystems as well as the periodic generation and updating of new parameter values are based on parameterized dataflow semantics [5]. For more details on parameterized dataflow, we refer the reader to [5].

To accommodate dynamic changes to α and dynamic selections between actors B or C based on the `selection` parameter, we construct the ADSG representation shown in Figure 7(b). Here, we utilize the ability to embed control information within DSG tokens to achieve the dynamic reconfiguration required by the given application.

The RA R_I determines the updated value of the parameter α, which we denote (with a minor abuse of notation) by $\alpha(t)$, and embeds this value in the DSG token that is output by R_I. This value is then used to control the number of iterations in the nested loop SCA labeled as $loop_2$. The *if* and *fi* SCAs perform conditional execution of actor B or C based on the current value of the `selection` parameter. The current value of this parameter is embedded in the DSG control token that is output by R_S so that it can be queried by the subsequent execution of the *if* SCA.



(a)



(b)

Figure 7. (a) A PSDF model for a reconfigurable phase shift keying application; (b) an ADSG representation for implementing this application.

Experimental results with different numbers of application graph iterations (processed blocks of data samples) are given in Table 8(a), and a corresponding chart is shown in Figure 8(b). The experiments are performed using the PSDFSim simulation environment, which can be adapted to implement and experiment with different types of schedules for PSDF graphs [21].

The quasi-static schedule provided by the DSG is compared to the standard PSDF scheduling approach, which can be viewed as a *dynamic scheduling approach*, of recomputing the schedule dynamically every time graph parameters change. The dynamic scheduling approach is more general and easier to apply, while a quasi-static approach has the potential for significant performance improvements by exploiting application-specific structure in the schedule. The DSG representation helps to capture this structure in a standard, dataflow-based format that is easily integrated into the PSDFSim environment.

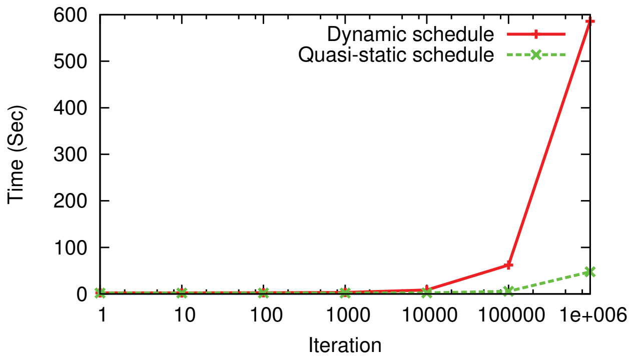
The performance of the quasi-static schedule is consistently better than the performance of the dynamic schedule. The degree of performance improvement generally increases with increasing numbers of iterations, which correspond to increasing numbers of input samples that are processed in the simulation. This is due to overhead in construction of the DSG representation that is more effectively amortized across the input data set as the size of the data set increases. Thus, for larger numbers of iterations, the DSG-based quasi-static schedule significantly outperforms the dynamic schedule.

B. Hardware Architecture Mapping from a DSG

In this section, we experiment with a *reconfigurable phase-shift keying* (*RPSK*) modulator application, which can be configured as binary PSK (BPSK), quadrature PSK

Dynamic schedule (Sec.)							
Iteration	1	10	10^2	10^3	10^4	10^5	10^6
CPU Time	1.00	1.06	0.97	0.97	1.16	1.19	320.70
Total Time	**2.29**	**2.35**	**2.60**	**3.21**	**8.65**	**62.08**	**585.97**
Quasi-static schedule (DSG) (Sec.)							
CPU Time	0.60	0.59	0.59	0.59	0.59	0.61	0.64
Total Time	**1.86**	**1.87**	**1.93**	**1.95**	**2.28**	**5.67**	**47.45**

(a) Simulation results for DSG-based quasi-static scheduling.



(b) Performance chart from simulation.

Figure 8. Performance comparison between DSG-based quasi-static scheduling, and dynamic scheduling.

(QPSK) or 8PSK based on the desired trade-off between communication quality and performance. As in the previous section, we apply the parameterized synchronous dataflow (PSDF) model of computation for application modeling and scheduling.

Figure 9 shows our PSDF-based model of the RPSK modulator. Two parameters are employed for dynamic reconfiguration — β (analogous to the α parameter in Section X-A) provides the consumption rate of actor T, and ν, a parameter of actor X_{12}, provides the modulation frequency. Since ν does not affect the dataflow (production and consumption) rate of its associated actor, it does not show up in the dataflow rate annotations of Figure 9.

In a previous study with this RPSK application, we defined a general methodology for mapping PSDF graphs into hardware, and demonstrated synthesis results for the RPSK application using this methodology [21]. Analogous to the dynamic scheduling approach described in Section X-A, this methodology is easy to apply due to its generality, and is also useful as it provides a standard method to realize hardware implementations of PSDF graphs. The DSG provides a complementary method, which can be used (e.g., in later stages of the design process) to specialize the hardware mapping for a specific application, and capture the structure of such specialized mappings in an abstract form that can be targeted subsequently to platform-specific, hardware control structures.

From its formal, dataflow-based structure, the DSG is well-suited for transformation into optimized finite state

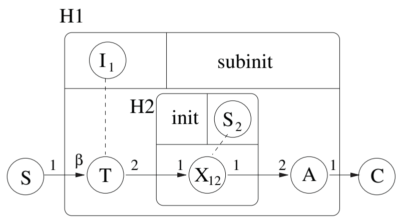
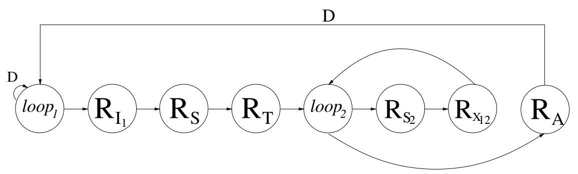
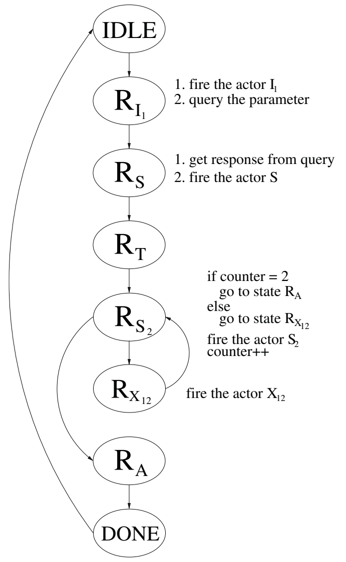


Figure 9. RPSK modulator.

machine (FSM) structures that provide control logic for hardware implementation of the associated schedules. Figure 10(b) illustrates a DSG representation for the RPSK targeted application, along with an FSM that is derived from the DSG. Most of the states map to distinct RAs, and execute the functionality associated with the associated RAs. Since the loop iteration count of $loop_2$ is fixed, the state R_{S_2} is designed to implement loop control as well as firing the actor S_2.



(a) A DSG for the RPSK modulator of Figure 9.



(b) An FSM for the DSG in Figure 10(a).

Figure 10. Hardware architecture mapping for a DSG.

In our experiments with hardware mapping, we targeted ASIC implementation using the Cadence Encounter RTL Compiler for back-end synthesis. The results reported here are synthesis results only (the design was tested thoroughly but not actually fabricated). Table II shows the improvement in area that is achieved by the streamlined DSG representation compared to the general-purpose PSDF-to-hardware mapping approach of [21]. This improvement is accompanied by a formal, dataflow based representation of schedule logic, which can be retargeted systematically to other types of platforms for rapid prototyping and experimentation with platform-specific implementation trade-offs.

Table II
AREA COMPARISON FOR RPSK MODULATOR UNDER CONSTANT SPEED (100 MHZ).

	DSG	General-purpose mapping	Reduction
Area (cell)	18949	20004	5.27%

C. Application to Software Implementation

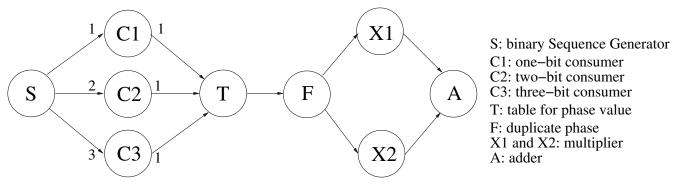
We use the core functional dataflow (CFDF) model of computation (see Section III), and the lightweight dataflow (LWDF) programming method [22] for software design and implementation of the RPSK application described in Section X-B, and for DSG-based experimentation with alternative schedules in this software context. LWDF can be viewed as a “minimalistic” approach for integrating coarse grain dataflow programming structures into arbitrary simulation- or platform-oriented languages, such as C, C++, CUDA, Java, Verilog, and VHDL. For more details on LWDF programming, we refer the reader to [22].

Figure 11(a) illustrates a C language implementation using CFDF and LWDF. In Figure 11(a), actors S and T are CFDF actors with three modes each. The variable M is set to 1, 2 or 3 depending on whether the current communication mode is is BPSK, QPSK or 8PSK, respectively. Depending on the modes of S and T, data is routed to one of the actors $C1$, $C2$ or $C3$. The consumption rates of these actors are different, as the annotations in Figure 11(a) show. In Figure 12, the function `guarded_execution` carries out a CFDF guarded execution of the given actor, and returns `true` if the associated actor firing was carried out (i.e., if the actor was enabled to begin with).

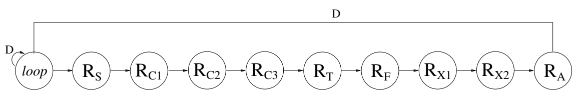
A DSG representation of a *canonical schedule* is shown in Figure 11(b). A canonical schedule for a CFDF graph can be viewed as a simple, brute force way to schedule the graph [20]. Canonical schedules usually have high run-time overhead, but can be useful for rapid prototyping purposes because they can be constructed very easily and quickly.

Compared to the canonical schedule, the schedule modeled by the DSG in Figure 11(c) is more efficient. This schedule model employs SCAs to direct control flow based on the active communication mode, and minimize run-time overhead due to fireability (enable condition) checking.

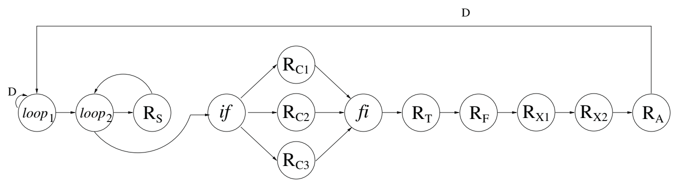
Experiments with these schedules were carried out on a Windows-based desktop computer with a 2.8 GHz CPU



(a) Application graph for RPSK system.



(b) Canonical schedule represented by an SDSG.



(c) A more efficient schedule represented by an ADSG.

Figure 11. RPSK system and alternative DSGs.

```
do {
    progress ← 0
    M = query(S)
    for 1 to M{progress ← progress OR guarded_execution(S)}
    switch(M){
        case 1: progress ← progress OR guarded_execution(C1) break
        case 2: progress ← progress OR guarded_execution(C2) break
        case 3: progress ← progress OR guarded_execution(C3) break
    }
    progress ← progress OR guarded_execution(T)
    progress ← progress OR guarded_execution(F)
    progress ← progress OR guarded_execution(X1)
    progress ← progress OR guarded_execution(X2)
    progress ← progress OR guarded_execution(A)
} while (progress)
```

Figure 12. Outline of software implementation structure for the DSG shown in Figure 11(c).

and 1GB RAM. The `gcc version 3.4.4` compiler was used in the back end of the implementation process.

Table III compares the performance of the canonical schedule DSG (denoted by $C.Sched$), and the DSG of the more efficient schedule (denoted by $E.Sched$). The overall performance measurement is performed using pre_S and $post_A$, which record the starting and stopping time for execution, respectively. Such implementation of performance measurement functionality represents a useful application of RA subfunctions, which in this case help to modularize, cleanly separate, and formally connect performance instrumentation code with respect to application (actor) and schedule code. The difference in performance between the two schedules is largely due to the higher frequency of guarded execution failures (i.e., calls to the `guarded_execution` function that return `false`) that result from the canonical schedule.

Table III
PERFORMANCE COMPARISON BETWEEN ALTERNATIVE SCHEDULES BASED ON DSG MODELING. THE UNITS OF TIME IN THIS TABLE ARE SECONDS.

# of bits	3×10^3	3×10^4	3×10^5	3×10^6	3×10^7
M = 1 (BPSK)					
C.Sched	0.64	0.89	3.49	28.05	275.96
E.Sched	0.63	0.83	3.21	25.69	251.72
Improv.	**1.56%**	**6.74%**	**8.02%**	**8.41%**	**8.78%**
M = 2 (QPSK)					
C.Sched	0.64	0.83	3.10	25.16	264.46
E.Sched	0.63	0.73	2.05	14.79	142.26
Improv.	**1.56%**	**12.05%**	**33.87%**	**41.22%**	**46.21%**
M = 3 (8PSK)					
C.Sched	0.62	0.81	2.91	23.82	234.18
E.Sched	0.62	0.71	1.60	10.83	103.88
Improv.	**0.00%**	**12.35%**	**45.02%**	**54.53%**	**55.64%**

This experiment helps to demonstrate how the DSG representation can be used as a common framework for experimenting with alternative schedules for software implementation. In this case, the DSG representation is used for initial functional validation using the canonical schedule, followed by a natural progression to a more sophisticated schedule, which provides opportunities for performance optimization once initial functional validation has been achieved. Recall from the formal semantics of dataflow graphs that for all valid schedules (i.e., schedules that respect the dataflow properties of the application), functional correctness is independent of the schedule. Thus, such a progressive or incremental approach to schedule exploration is attractive from the viewpoints of separating concerns, structuring the design process, and improving overall productivity. These are all useful viewpoints to help designers leverage the power of heterogeneous computing platforms.

XI. EXTENSIONS

For concreteness, we have presented the DSG model in the context of CFDF semantics. However, the DSG model can be adapted to other dataflow models or environments that can support a notion of *guarded execution* — i.e., a check for fireability followed by execution of the associated actor if it is found to be fireable.

In contrast, models in which fireability checking and actor invocation are interleaved (with blocking reads) cannot be integrated directly into the proposed DSG framework. However, a more restricted form of DSG can be employed in which RAs fire their associated actors unconditionally. Such DSGs require more care in their construction (to avoid run-time deadlock), and can be useful for modeling static or quasi-static scheduling structures where significant information is available at compile time for DSG derivation.

XII. CONCLUSIONS

In this paper, we have introduced the *dataflow schedule graph* (*DSG*) as a formal, dataflow-based model for representing and interpreting schedules for dataflow graphs. We have shown that both sequential and parallel schedules can be accommodated in the DSG framework. The DSG is not restricted to any specific dataflow model, and provides for a wide range of static, quasi-static, and dynamic scheduling structures. Furthermore, the model is easily extended with new types of schedule control actors and reference actor subfunctions so that the structures of the represented schedules can be flexibly customized by designers, tool developers, and adaptive scheduling strategies. We have demonstrated the utility of the DSG representation through various examples with emphasis on demonstrating the utility across a heterogeneous variety of computing platforms. Useful directions for future work include the application of DSGs as a substrate for optimized integration of hybrid scheduling techniques.

XIII. Acknowledgments

This research was sponsored in part by the US Air Force Research Laboratory, and the Laboratory for Telecommunication Sciences.

References

[1] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, Eds., *Handbook of Signal Processing Systems*. Springer, 2010.

[2] M. Ade, R. Lauwereins, and J. Peperstraete, "Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets," in *Proceedings of the Design Automation Conference*, June 1997, pp. 64–69.

[3] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for DSP," *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 849–875, September 2000.

[4] S. Ha and E. A. Lee, "Compile-time scheduling of dynamic constructs in dataflow program graphs," *IEEE Transactions on Computers*, vol. 46, no. 7, July 1997.

[5] 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.

[6] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Proceedings of the International Conference on Formal Methods and Models for Codesign*, July 2006.

[7] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi, "Throughput analysis of synchronous data flow graphs," in *Proceedings of the International Conference on Application of Concurrency to System Design*, June 2006.

[8] 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.

[9] H. Oh, N. Dutt, and S. Ha, "Memory optimal single appearance schedule with dynamic loop count for synchronous dataflow graphs," in *Proceedings of the Asia South Pacific Design Automation Conference*, 2006, pp. 497–502.

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

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, September 1987.

[12] A. Gerasoulis and T. Yang, "On the granularity and clustering of directed acyclic task graphs," *IEEE Transactions on Parallel and Distributed Systems*, pp. 686–701, June 1993.

[13] V. Kianzad and S. S. Bhattacharyya, "Efficient techniques for clustering and scheduling onto embedded multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 7, pp. 667–680, July 2006.

[14] Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *Journal of the Association for Computing Machinery*, vol. 31, no. 4, pp. 406–471, December 1999.

[15] H. J. Siegel, J. K. Antonio, R. C. Metzger, M. Tan, and Y. A. Li, "Heterogeneous computing," in *Parallel and Distributed Computing Handbook*, A. Y. Zomaya, Ed. McGraw-Hill, 1996.

[16] L. Wang, H. J. Siegel, and V. Roychowdhury, "A genetic-algorithm-based approach for task matching and scheduling in heterogeneous environments," in *Proceedings of the Hetergeneous Computing Workshop*, April 1996, pp. 72–85.

[17] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya, "Functional DIF for rapid prototyping," in *Proceedings of the International Symposium on Rapid System Prototyping*, Monterey, California, June 2008, pp. 17–23.

[18] B. Kienhuis and E. F. Deprettere, "Modeling stream-based applications using the SBF model of computation," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, September 2001, pp. 385–394.

[19] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs using the token flow model," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, April 1993.

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

[21] H. Wu, H. Kee, N. Sane, W. Plishker, and S. S. Bhattacharyya, “Rapid prototyping for digital signal processing systems using parameterized synchronous dataflow graphs,” in *Proceedings of the International Symposium on Rapid System Prototyping*, Fairfax, Virginia, June 2010.

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