In Transactions on High-Performance Embedded Architectures and Compilers, Volume 4, Issue 4, 2009.

Heterogeneous Design in Functional DIF

William Plishker, Nimish Sane, Mary Kiemb, and Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies,
University of Maryland at College Park, USA
{plishker,nsane,kiemb,ssb}@umd.edu
http://www.ece.umd.edu/DSPCAD

Abstract. Dataflow formalisms have provided designers of digital signal processing (DSP) systems with analysis and optimizations for many years. As system complexity increases, designers are relying on more types of dataflow models to describe applications while retaining these implementation benefits. The semantic range of DSP-oriented dataflow models has expanded to cover heterogeneous models and dynamic applications, but efficient design, simulation, and scheduling of such applications has not. To facilitate implementing heterogeneous applications, we utilize a new dataflow model of computation and show how actors designed in other dataflow models are directly supported by this framework, allowing system designers to immediately compose and simulate actors from different models. Using examples, we show how this approach can be applied to quickly describe and functionally simulate a heterogeneous dataflow-based application such that a designer may analyze and tune trade-offs among different models and schedules for simulation time, memory consumption, and schedule size.

Key words: Dataflow, Heterogeneous, Signal Processing

1 Introduction

For a number of years, dataflow models have proven invaluable for application areas such as digital signal processing. Their graph-based formalisms allow designers to describe applications in a natural yet semantically rigorous way. Such a semantic foundation has permitted the development of a variety of analysis tools, including determining buffer bounds and efficient scheduling [1]. As a result, dataflow languages are increasingly popular. Their diversity, portability, and intuitive appeal have extended them to many application areas with a variety of targets (e.g., [2][3][4])

As system complexity and the diversity of components in digital signal processing platforms increases, designers are expressing more types of behavior in dataflow languages to retain these implementation benefits. While the semantic range of dataflow has expanded to cover quasi-static and dynamic interactions, efficient functional simulation and the ability to experiment with more flexible

scheduling techniques has not. Complexity in scheduling and modeling has impeded efforts of a functional simulation that matches the final implementation. Instead, designers are often forced to go all the way to implementation to verify that dynamic behavior and complex interaction with various domains are correct. Correcting functional behavior in the application creates a developmental bottleneck, slowing the time to implementation on a heterogeneous platform.

To understand complex interactions properly, designers should be able to describe their applications in a single environment. In the context of dataflow programming, this involves describing not only the top level connectivity and hierarchy of the application graph, but also the functionality of the graph actors (the functional modules that correspond to non-hierarchical graph vertices), preferably in a natural way that integrates with the semantics of the dataflow model they are embedded in. Once the application is captured, designers need to be able to evaluate static schedules (for high performance) alongside dynamic behavior without loosing semantic ground. With such a feature set, designers should arrive at heterogeneous implementations faster.

To address these designer needs, we present a new dataflow model called *enable-invoke dataflow* (EIDF), a preliminary version of which was published in [5]. EIDF permits the natural description of actors for a variety of dynamic (and static) dataflow models. Even if the actors adhere to different models of dataflow, being described in EIDF ensures that they may be composed and functionally simulated together. Leveraging our existing *dataflow interchange format* (DIF) package [6], we implement an extension to DIF based on a restricted form of EIDF, called *core functional dataflow* (CFDF), that facilitates the simulation of heterogeneous applications. This extension to DIF, called *functional DIF*, allows designers to verify the functionality of their application immediately as we showed in preliminary work in [7]. From this working application, designers may focus on efficient schedules and buffer sizing, and thus are able to arrive at quality implementations of heterogeneous systems quickly.

This article is organized in the following fashion: relevant background is covered in Section 2 and related research is discussed in Section 3. The semantics for our dataflow formalisms are presented in Section 4 while their relationship to other dataflow models is covered in Section 5. The implementation of functional DIF is covered in Section 6. Two design examples are discussed in Section 7 and Section 8 summarizes and concludes the work.

2 Background

2.1 Dataflow Modeling

Modeling DSP applications through coarse-grain dataflow graphs is widespread in the DSP design community, and a variety of dataflow models has been developed for dataflow-based design. A growing set of DSP design tools support such dataflow semantics [8][9]. Ideally, designers are able to find a match between their application and one of the well studied models, including cyclo-static

dataflow (CSDF) [10] , synchronous dataflow (SDF) [1], single-rate dataflow, homogeneous synchronous dataflow (HSDF), or a more complicated model such as boolean dataflow (BDF) [11].

Common to each of these modeling paradigms is the representation of computational behavior as a dataflow graph. A dataflow graph G is an ordered pair (V, E), where V is a set of vertices (or nodes), and E is a set of directed edges. A directed edge $e = (v_1, v_2) \in E$ is an ordered pair of a source vertex $v_1 \in V$ and a sink vertex $v_2 \in V$. A *source function*, $src : E \to V$, maps edges to their source vertex, and a *sink function*, $snk : E \to V$ gives the sink vertex for an edge. Given a directed graph G and a vertex $v \in V$, the set of incoming edges of v is denoted as $in(v) = \{e \in E | snk(e) = v\}$, and similarly, the set of outgoing edges of v is denoted as $out(v) = \{e \in E | src(e) = v\}$.

2.2 Dataflow Interchange Format

To describe the dataflow applications for this wide range of dataflow models, application developers can use the *dataflow interchange format* (DIF) [6], an approach founded in dataflow semantics and tailored for DSP system design. *The DIF language* (TDL) provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification. From a dataflow point of view, TDL is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related and actor-specific information. The dataflow semantic specification is based on dataflow modeling theory and independent of any design tool.

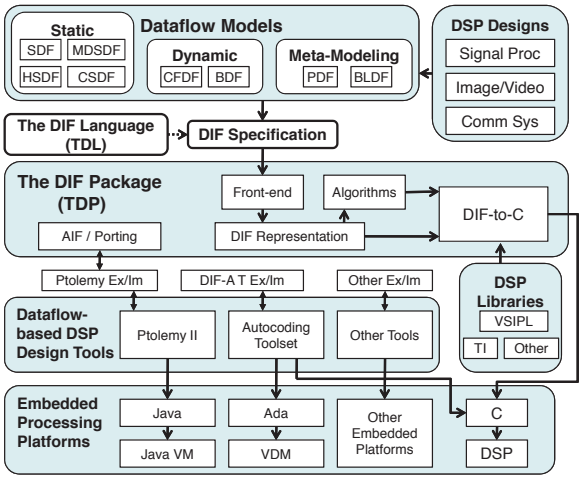

Fig. 1. DIF based design flow

To utilize the DIF language, *the DIF package* (TDP) has been built. Along with the ability to transform DIF descriptions into a manipulable internal rep-

resentation, TDP contains graph utilities, optimization engines, algorithms that may prove useful properties of the application, and a C synthesis framework [12]. These facilities make the DIF package an effective environment for modeling dataflow applications, providing interoperability with other design environments, and developing new tools. An overview of the DIF design flow using TDP is shown in Figure 1.

Beyond these features, TDP is also suitable as a design environment for implementing dataflow-based application representations. Describing an application graph is done by listing nodes and edges, and then annotating dataflow specific information. TDP also has an infrastructure for porting applications from other dataflow tools to DIF [13]. What is lacking in TDP is the ability to simulate functional designs in the design environment. Such a feature would streamline the design process, allowing applications to be verified without having to go to implementation.

3 Related Work

A number of development environments utilize dataflow models to aid in the capture and optimization of functional application descriptions. Ptolemy II encompasses a diversity of dataflow-oriented and other kinds of models of computation [14]. To describe an application subsystem, developers employ a director that controls the communication and execution schedule of an associated application graph. If an application developer is able to write the functionality of an actor in a prescribed manner, it will be polymorphic with respect to other models of computation. To describe an application with multiple models of computation, developers can insert a "composite actor" that represents a subgraph operating with a different model of computation (and therefore its own director). In such hierarchical representations, directors manage the actors only at their associated levels, and directors of composite actors only invoke their actors when higher level directors execute the composite actors. This paradigm works well for developers who know *a priori* the modeling techniques with which they plan to represent their applications.

Other techniques employ SystemC to capture actors as composed of input ports, output ports, functionality, and an execution FSM, which determines the communication behavior of the actor [15]. Other languages specifically targeting actor descriptions such as CAL [16]. For complete functionality in Simulink [9], actors are described in the form of "S-functions." By describing them in a specific format, actors can be used in continuous, discrete-time, and hybrid systems. LABVIEW [8] even gives designers a way of programmatically describing graphical blocks for dataflow systems.

Semantically, perhaps the most related work is the Stream Based Function (SBF) model of computation [17]. In SBF, an actor is represented by a set of functions, a controller, state, and transition function. Each function is sequentially enabled by the controller, and uses on each invocation a blocking read for

each input to consume a single token. Once a function is done executing, the transition function defines the next function in the set to be enabled.

Functional DIF differs from these related efforts in dataflow-based design in its integrated emphasis on a minimally-restricted specification of actor functionality, and support for efficient static, quasi-static, and dynamic scheduling techniques. Each may be critical to prototyping overall dataflow graph functionality. Compared to models such as SBF, functional DIF allows a designer to describe actor functionality in an arbitrary set of fixed modes, instead of parceling out actor behavior as side-effect free functions, a controller, and a transition function. Functional DIF is also more general than SBF as it permits multi-token reads and can enable actors based on application state. As designers experiment with different dataflow representations with different levels of actor dynamics, they need corresponding capabilities to experiment with compatible scheduling techniques. This is a key motivation for the integrated actor- and scheduler-level prototyping considerations in functional DIF.

4 Semantic Foundation

4.1 Enable-Invoke Dataflow

We propose a new dataflow model in which an actor specification is divided into enable and invoke. We call this model *enable-invoke dataflow* (EIDF). Any application based on EIDF also adheres to the dataflow formalism described in Section 2.1, where each of the vertices are actors that implement separate enable and invoke capabilities. These capabilities correspond, respectively, to testing for sufficient input data, and executing a single quantum (invocation) of execution for a given actor.

Each actor also has a set of *modes* in which it can execute. This set of modes can depend upon the type of dataflow model being employed or it may be user-defined. Each mode, when executed, consumes and produces a fixed number of tokens. This set of modes can depend upon the type of dataflow model being employed or it may be user-defined. Given an actor $a \in V$ in a dataflow graph, The *invoking function* for an actor a is defined as:

$$\kappa_a : (I_a \times M_a) \rightarrow (O_a \times Pow(M_a)), \tag{1}$$

where $I_a = X_1 \times X_2 \times \ldots \times X_{|in(a)|}$ is the set of all possible inputs to a, where X_i is the set of possible tokens on the edge on input port i of actor a. After a executes, it produces outputs $O_a = Y_1 \times Y_2 \times \ldots \times Y_{|out(a)|}$, where Y_i is the set of possible tokens on the edge connected to port i of actor a, where $|out(a)|$ is the number of output ports. In general, invoking an actor can change the mode of execution of the actor, so the invoking function also produces the set of modes that are valid next. If no mode is returned (i.e., an empty mode set is returned), the actor is forever disabled.

To ensure that an invoking mode has the necessary inputs to run to completion, we utilize the *enabling function* which for a is defined as:

$$\varepsilon_a : (T_a \times M_a) \rightarrow \mathbb{B}, \tag{2}$$

where $T_a = \aleph^{|in(a)|}$ is a tuple of the number of tokens on each of the input edges to actor a (here, $|in(a)|$ is the number of input edges to actor a); M_a is the set of modes associated with actor a; and the output is $true$ when an actor $a \in V$ has an appropriate number of tokens for mode $m \in M_a$ available on each input edge, and $false$ otherwise. An actor can be executed in a given mode at a given point in time if and only if the enabling function is true-valued.

The separation of enable and invoke capabilities helps in prototyping efficient scheduling techniques. Scheduling is key to executing dataflow models, since minimal emphasis is placed on execution ordering in the paradigm of dataflow-based application specification. Scheduling is therefore a necessary part of dataflow graph execution, and furthermore, scheduling has major impact on key implementation metrics, including memory requirements, performance, and power consumption (e.g., see [18]).

Dynamic dataflow behaviors require special attention to schedule to retain efficiency and minimize the loss of predictability. The enable function is designed so that if desired, one can use it as a "hook" for dynamic or quasi-static scheduling techniques to rapidly query actors at runtime to see if they are executable. For this purpose, it is especially useful to separate the enable functionality from the remaining parts of actor execution.

These remaining parts are left for the invoke function, which is carefully defined to avoid computation that is redundant with the enable function. The restriction that the enable method operates only on token counts within buffers and not on token values further promotes the separation of enable and invoke functionality while minimizing redundant computation between them. At the same time, this restriction does not limit the overall expressive power of EIDF, which is Turing complete, as enabling and invoking functions can be formulated to describe BDF actors. Since BDF is known to be Turing complete, and EIDF is at least as expressible as BDF, EIDF can express any computable function and important dynamic dataflow models.

The restrictions in EIDF can therefore be viewed as design principles imposed in the architecting of dataflow actors rather than restrictions in functionality. Thus, flexible and efficient support for prototyping with dynamic and quasi-static scheduling techniques is an important motivation for EIDF. Such techniques are generally needed when dynamic dataflow behavior is present, and may be convenient for early-stage prototyping of static dataflow behaviors by simplifying the construction of schedulers.

In summary, the EIDF model is tailored to natural actor design and also facilitates dataflow modeling for rapid prototyping. EIDF is a generic model with its semantics independent of the underlying dataflow model used to describe a particular application. Thus, one can efficiently experiment with different specialized dataflow formats in the context of a given application. It is also possible to integrate dynamic parameterization (i.e., parameters whose values can be set and changed dynamically) into EIDF for example, through the meta-modeling framework of parametrized dataflow [19], to yield parameterized EIDF (PEIDF).

Such rigorous integration of dynamic parameterization into EIDF is a useful topic for future work.

4.2 Core Function Dataflow

For a formalism customized to functional DIF, we derive a special case of the EIDF model that we refer to as *core functional dataflow* (CFDF). In the case of the EIDF model, the invoking function returns a set of valid modes of execution for an actor. This allows for non-determinism as an actor can be invoked in any of these valid modes. In the deterministic CFDF model, actors must proceed deterministically to one particular mode of execution whenever they are enabled. Hence, the invoking function should return only a single valid mode of execution instead of a set. The generic definition of the invoking function can be modified as

$$\kappa_a^* : (I_a \times M_a) \rightarrow (O_a \times M_a), \tag{3}$$

With this restricted form of invoking function, only one mode can meaningfully be interrogated by the enabling function. As long as the modes of an actor are themselves deterministic, requiring a single, unique next mode ensures that the resulting application is deterministic.

4.3 Functional DIF Semantic Hierarchy

Functional DIF is the realization of the CFDF semantics in our DIF package. Figure 2 shows the new hierarchical structure of functional DIF semantics with respect to the DIF package. DIF Graph is at the highest hierarchical level in the DIF semantics. EIDF represents our generalized, enable-invoke dataflow abstraction for applications specified by non-deterministic dataflow models. It provides a mechanism to handle generic methods that form a basis for many common dataflow models.

As described earlier, CFDF is a restricted form of EIDF for modeling deterministic dataflow applications. CFDF is the most general form of dataflow that we consider in our development of dataflow representations in DIF that have functional capabilities (as opposed to abstract dataflow graphs for application analysis). We are actively expanding the sub-tree rooted at CFDF to enable an increasing set of specialized dataflow modeling techniques available.

5 Translation to CFDF/EIDF

Many common dataflow models may be directly translated to CFDF in an efficient and intuitive manner. In this section we show such constructions, demonstrating the expressibility of CFDF and how the burden of design is eased when starting from an existing dataflow model.

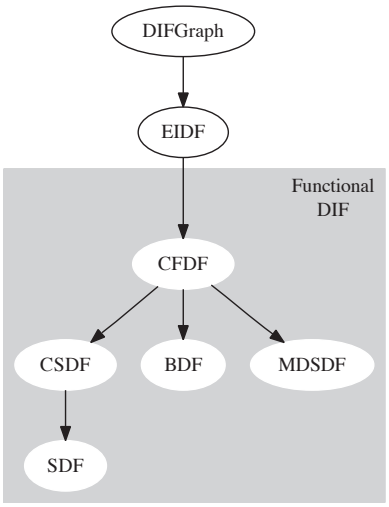

Fig. 2. Functional DIF semantic hierarchy

5.1 Static dataflow

SDF, CSDF, and other static dataflow-actor behaviors can be translated into finite sequences of CFDF modes for equivalent operation. Consider, for example, CSDF, in which the production and consumption behavior of each actor a is divided into a finite sequence of periodic phases $P = (1, 2, ..., n_a)$. Each phase has a particular production and consumption behavior. The pattern of production and consumption across phases can captured by a function ϕ_a whose domain is P_a. Given a phase $i \in P_a$, $\phi_a(i) = (G_i, H_i)$, where G_i and H_i are vectors indexed by the input and output ports of a, respectively, that give the numbers of tokens produced and consumed on these edges for each port during the ith phase in the execution of actor a.

To construct a CFDF actor from such a model, a mode is created for each phase, and we denote the set of all modes created in this way by M_a. Given a mode $m \in M_a$ corresponding to phase $p \in P_a$, the enable method for this mode checks the input edges of the actor for sufficient numbers of tokens based on what the phase requires in terms of the associated CSDF semantics. Thus, for each input port z of a, mode m checks for the availability of at least $G_p(z)$ tokens on that port, where $\phi(p) = (G_p, H_p)$. For the complementary invoke method, the consumption of input ports is fixed to G_p, the production of output ports is fixed to H_p. The next mode returned by the invoke method must be the mode corresponding to the next phase in the CSDF phase sequence. Since any SDF actor can be viewed as a single-phase CSDF actor, the CFDF construction process for SDF is a specialization of the CSDF-to-CFDF construction process described above in which there is only one mode created.

5.2 Boolean dataflow

Boolean dataflow (BDF) adds dynamic behavior to dataflow. The two fundamental elements of BDF are Switch and Select. Switch routes a token from its input to one of two outputs based on the Boolean value of a token on its control input. The concept of a control input is also utilized for Select, in which the value of the control token determines which input port will have a token read and forwarded to its one output.

To construct a CFDF actor that implements BDF semantics, we create a mode that is dedicated to reading that input value, which we call the control mode. The result of this examination sends the actor into either a true mode or a false mode that corresponds to that control port. In the case of Switch, this implies three modes with behavior described in Table 1. Note that a single invocation of a Switch in BDF corresponds to two modes being invoked in the CFDF framework. For a strict construction of BDF, only the Switch and Select actors are needed for implementation, but CFDF does permit more flexibility, allowing designers to specify arbitrary behavior of true and false modes as long as each mode has a fixed production and consumption behavior.

Table 1. The behavior of modes in a Switch actor

mode	consumes		produces	
	Control	Data	True	False
Control	1	0	0	0
True	0	1	1	0
False	0	1	0	1

5.3 Gustav Function

While EIDF is designed to handle non-deterministic applications, it is also useful for certain cases of deterministic behavior in which CFDF is not expressible enough. The EIDF generalization of multiple next modes provides the expressibility necessary for some of these cases. Consider the Gustav function [20] the behavior of which is described in Table 2. Like the Switch actor in BDF, certain behaviors are tied to the value of tokens. The behavior is deterministic, but conditioned on the values presented at the inputs. To determine which two tokens should be processed, the values on the inputs must be known. This presents a problem for blocking read semantics like that of CFDF. Switch is implementable by splitting the token production and consumption into separate phases such that the dataflow behavior can be conditioned on the value of tokens. In the case of the Gustav function, it cannot be known *a priori* which token to read to indicate which token to read next. For example, the actor cannot simply wait for the value of input I_1 because the next mode to be processed might be m_3 implying a token might not appear on I_1 leaving the actor forever waiting while valid data is ready to be processed on the other two inputs.

Table 2. The behavior of the Gustav Function

modes	consumes		
	I_1	I_2	I_3
m_1	T	F	
m_2	F		T
m_3		T	F

Fortunately, EIDF is capable of processing these inputs by leveraging multiple possible next modes. Consider the subset of Gustav traces presented in Figure 3. Initially, the actor attempts to read a single token off any port, which is represented by the set of initial valid modes: $ReadI_1$, $ReadI_2$, and $ReadI_3$. By allowing any one of those modes to become enabled (and then successfully invoked), we avoid blocking on any one port. Once a token is read in (in our example, a token from I_1), the actor waits for a token from one of the other two ports. Once two tokens are received, the data may be ready for processing, or may need to wait for a final token on the last remaining port. Once an action mode has been successfully invoked, it returns an appropriate set of read modes to continue processing.

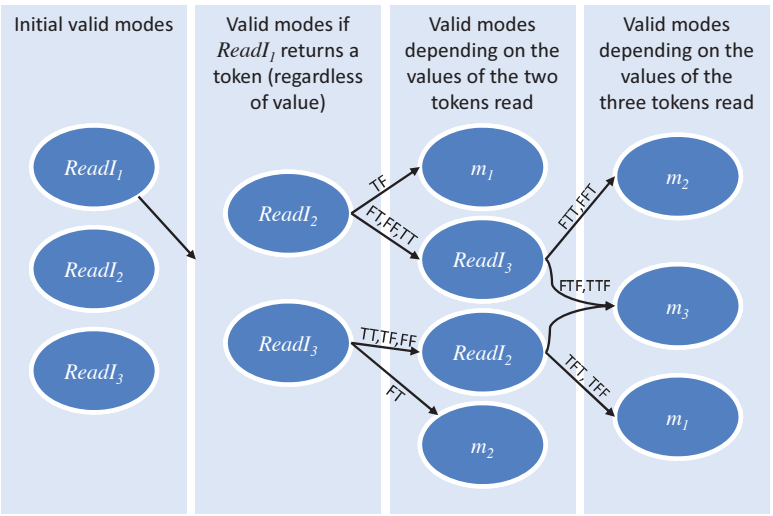

Fig. 3. Example traces through the EIDF implementation of the Gustav Function

The key to this implementation is using non-deterministic mechanisms present in EIDF to capture the unique deterministic behavior of this function. The Gustav function is another benchmark for expressibility that shows the power of EIDF. With only 6 modes (one for each input read, and one for each action),

we are able to describe and implement the Gustav function in a natural way amenable for implementation.

6 Design and Implementation

To construct an efficient realization of CFDF, extensions to the DIF package have been made carefully. The following section discusses these changes, along with the functionality needed to simulate CFDF applications.

6.1 Software Architecture

The DIF package has been restructured to extend it to functional DIF. We have introduced an library of actors described in the Java programming language for use in functional DIF applications. Actors are objects derived from a base class that provides each actor with mode and edge interfaces along with base methods for the enabling and invoking functions, called the enable method and the invoke method respectively. Modes can be created either by a user through an API or automatically, based on other information about the application (e.g., the sequence of phases in a cyclo-static dataflow representation). While a designer will redefine an actor's class methods to define the proper functionality, the enable method is always restricted to only checking the number of tokens on each input (as per the enabling function definition). The invoke method may read values from inputs, but it must consume them as tokens. In other words, when a mode is invoked on an actor, the actor consumes a fixed number of tokens that is associated with that mode, and no more values are read. In either case, we expect designers to effectively construct a case statement of all of the possible modes for a given actor, and fill in the functionality of each mode in a case.

6.2 Scheduler and Simulator

We have used the *generalized schedule tree* (GST) [21] representation to represent schedules generated by schedulers in functional DIF. The GST representation is a generalization of the (binary) schedule tree representation developed for R-schedules [22]. The GST representation can be used to represent dataflow graph schedules irrespective of the underlying dataflow model or scheduling strategy being used. GSTs are ordered trees with leaf nodes representing the actors of an associated dataflow graph. An internal node of the GST represents the loop count of a schedule loop (an iteration construct to be applied when executing the schedule) that is rooted at that internal node. The GST representation allows us to exploit topological information and algorithms for ordered trees in order to access and manipulate schedule elements. To functionally simulate an application, we need only generate a schedule for the application, and then traverse the associated GST to iteratively enable (and then execute, if appropriate) actors that correspond to the schedule tree leaf nodes. Note that if actors are not enabled, the GST traversal simply skips their invocation. Subsequent schedule

rounds (and thus subsequent traversals of the schedule tree) will generally revisit actors that were unable to execute in the current round.

We can always construct a *canonical schedule* for an application graph. This is the most trivial schedule that can be constructed from the application graph. The canonical schedule is a single appearance schedule (a schedule in which actors of the application graph appear once) which includes all actors in some order. In terms of the GST representation, a canonical schedule has a root node specifying the loop count of 1 with its child nodes forming leaves of the schedule tree. Each leaf node points to a unique actor in the application graph. The ordering of leaf nodes determines the order in which actors of the application graph are traversed. When the simulator traverses GST, each actor in the graph is fired, if it is enabled.

7 Design Example - Polynomial Evaluation

Polynomial evaluation is a commonly used primitive in various domains of signal processing, such as wireless communications and cryptography. Polynomial functions may change whenever senders transmit data to receivers. The kernel is the evaluation of a polynomial $P_i(x) = \sum_{k=0}^{n_i} c_k \times x^k$, where $c_1, c_2, \ldots, c_n$ are coefficients, x is the polynomial argument, and n_i is the degree of the polynomial. Since the coefficients may change at runtime, a programmable *polynomial evaluation accelerator* (PEA) is useful for accelerating the computation of multiple P_i's.

7.1 Programmable PEA

Since the degree and the coefficients may change at runtime (e.g., for different communications standards or different subsystem functions), a programmable polynomial evaluation accelerator (PEA) is useful for accelerating the computation of multiple P_i 's in a flexible way. To this end, we design a PEA with the following instructions: *reset*, *store polynomial*(STP), *evaluate polynomial*, and *evaluate block*. *Evaluate polynomial* is for a single evaluation, and *evaluate block* is for bulk evaluation of the same polynomial.

Since data consumption and production behavior for the PEA depends on the specific instruction, a PEA actor cannot follow the semantics of conventional dataflow models, such as SDF. However, if we define multiple modes of operation, we can capture the required dynamic behavior as a collection of CFDF modes. Following this principle, we have implemented the PEA as a single CFDF actor. In our functional description of the actor, we defined different modes according to the four PEA instructions. These modes are summarized in Table 3.

The normal mode (like the "decode" stage in a typical processor) reads an instruction and determines the next operating mode of the datapath. Of particular note here is the behavior of STP, in which a variable number of coefficients is read. Each individual mode is restricted to one particular consumption rate, so when the STP mode is invoked, it reads a single coefficient, stores it, and updates

Table 3. The behavior of the PEA modes

mode	consumes		produces	
	Control	Data	Result	Status
Normal	1	0	0	0
Reset	0	0	0	0
Store Poly	0	1	0	1
Evaluate Poly	0	1	1	1
Evaluate Block	0	1	1	1

an internal counter. If the counter is less than the total number of coefficients to be stored, invoke returns STP as the next mode, so it will continue reading until done. Note that persistent internal variables ("actor state variables"), such as a counter, can be represented in dataflow as self-loop edges (edges whose source and sink actors are identical), and thus, the use of internal variables does not violate the pure dataflow semantics of the enclosing DIF environment. In future versions, we plan to incorporate parameterized dataflow [1] semantics to implement STP as a single PEA mode with a dynamically parameterized mode consumption rate.

Using an implementation based on the same behavior implemented by the functional DIF actor, we were able to compactly specify the behavior of the PEA in functional DIF. Through simulation of our functional DIF implementation, we verified its correctness and confirmed it produced the same output of a Verilog implementation we wrote of the same actor and test bench. A comparison of simulation time for the PEA between functional DIF and Verilog is shown in Table 4. We simulated this Verilog description using Modelsim version 6.3 SE. We used two different test bench input files and measured the time spent in simulation. Functional DIF improved the simulation time by over a factor of four in this example. Because the simulation of the Functional DIF description occurs at a higher level than Modelsim, we expect the speedups could be even higher. However, it is worth noting that we have not spent any time performance tuning the DIF simulator. We believe that with more performance tuning these speedup number should improve dramatically.

Table 4. Simulation times of Verilog and Functional DIF for the PEA test bench with two different sets of instructions.

Instruction set	Verilog Simulation Time (ms)	Functional Simulation Time (ms)	DIF Speedup
Case 1	250	55	4.6x
Case 2	170	33	5.1x
Average	210	44	4.9x

7.2 Multi-PEA design

To illustrate the problem of heterogeneous complexity, we suppose that a DSP application designer might use two PEA actors customized for different length polynomials. For this application, we restrict the PEA's functionality to be a CSDF actor with two phases: reading the polynomial coefficients and then processing a block of x's to be evaluated, as shown in Table 5. The overall PEA system is shown in Figure 4. Two PEA actors are in the same application and made them selectable by bracketing them with a Switch and a Select block. To manage the two PEA actors properly, this design requires control to select the *PEA1* or *PEA2* branch. In this system, the CSDF PEA actors consume a different number of polynomial coefficient tokens, so the control tokens driving the switch and select on the datapath must be able to create batches of 19 and 22 tokens, respectively for each path. If the designer is restricted to only Switch and Select for BDF functionality, the balloon with *CONTROLLER* shows how this can be done.

Table 5. The behavior of the CSDF implementation of the restricted PEA used in the dual PEA application

Actor	mode	consumes Data	produces Result
PEA1	Store Poly	4	0
	Evaluate Block	15	15
PEA2	Store Poly	7	0
	Evaluate Block	15	15

This design can certainly be captured with model oriented approach, pulling the proper actors into super-nodes with different models. But like many designs, this application has a natural functional hierarchy in it with the refinement of *CONTROLLER* and with *PEA1* and *PEA2*. We believe that competing design concerns of functional and model hierarchy will ultimately be distracting for a designer. With this work, we focus designers on efficient application representation and not model related issues.

Immediate simulation of the dual PEA application is possible to verify correctness by using the canonical schedule. We simulated the application with a random control source and a stream of integer data. A nontrivial schedule tree can significantly improve upon the canonical performance. Given that the probability of a given PEA branch being selected is uniform, we can derive a single appearance schedule shown in Figure 5, where each leaf node is annotated with an actor and each interior node is annotated with a loop count. Leaf nodes are double ovals to indicate they are guarded by the enabling function, while a single oval is unguarded. Figure 6 shows a manually designed multiple appearance schedule (a schedule in which actors may appear more than once) that attempts to process polynomial coefficients first, before queuing up data to be evaluated,

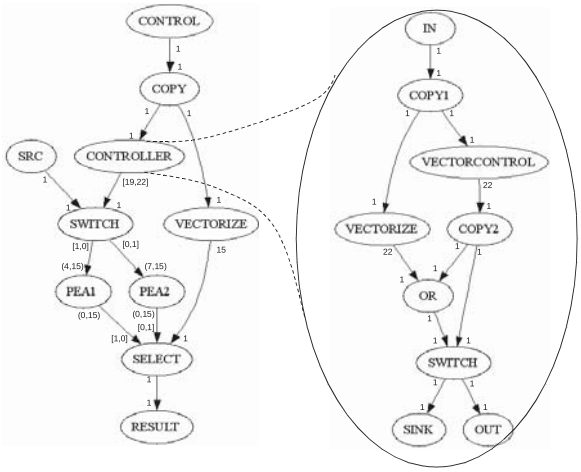

Fig. 4. A pictorial representation of the dual PEA application.

to reduce buffering. Note that the *SRC* and *CONTROL* actor are unguarded as they require no input tokens to successfully fire.

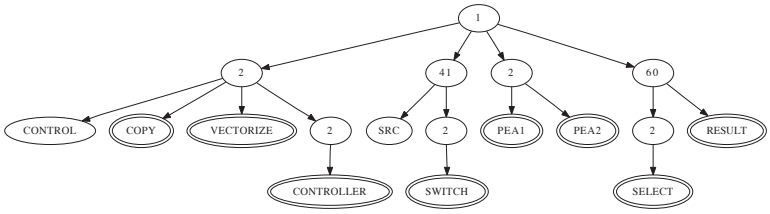

Fig. 5. Single appearance schedule for the dual PEA system

We also implemented an M channel uniform discrete Fourier transform (DFT) filter bank and a sample rate conversion application. We constructed the decimated uniform DFT filter bank using a mixed-model consisting of CSDF and SDF actors. The sample rate conversion application is based on concepts found in [23] and [12]. Results for these different implementations with different schedules are summarized by Table 6. We simulated 10,000 evaluations running on a 1.7GHz Pentium with 1GB of physical memory. We measured the time it took to complete enough iterations to complete all of the evaluations and maximum total queue size. The manually designed schedules performed notably better than the canonical schedule. Such insight can be invaluable when considering the final implementation of the controller logic.

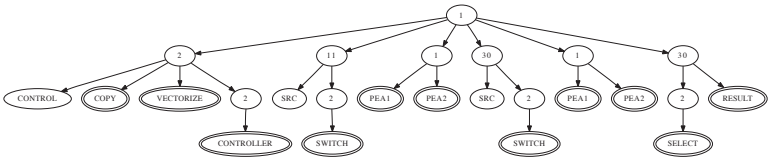

Fig. 6. Multiple appearance schedule for the dual PEA system

Table 6. Simulation times and max buffer sizes of the dual PEA design.

Application	Schedule	Simulation Time (s)	Max observed buffer size (tokens)
Dual PEA - BDF Strict	Canonical	6.88	2,327,733
	Single appearance	1.72	1,729
	Multiple appearance	1.59	1,722
Dual PEA - CFDF	Canonical	3.57	1,018,047
	Single appearance	0.95	1,791
	Multiple appearance	0.99	1,800
DFT Filter	Canonical	0.91	17
	Single appearance	1.02	24
Sample Rate Conv	Canonical	9.15	9,394
	Single appearance	1.43	2,408

8 Conclusions and Future Work

In this work, we have presented a new dataflow approach to enable the description of heterogeneous applications that utilize multiple forms of dataflow. This is based on a new dataflow formalism, a construction scheme to translate from existing dataflow models to it, and a simulation framework that allows designers to model and verify interactions between those models. With this approach integrated into TDP, we demonstrated it on the heterogeneous design of a dual polynomial evaluation accelerator. Such an approach allowed us to functionally simulate the design immediately and then to focus on experimenting on schedules and dataflow styles to improve performance.

This new dataflow model and modeling technique have many aspects for further work. One of the ongoing efforts focuses on developing generalized scheduling technique for CFDF graphs, in particular those representing dynamic applications. Such a technique would explore the fact that production and consumption rates of an actor are fixed for any given mode. This fact can be exploited to decompose a given dynamic CFDF graph into a set of static graphs that interact dynamically. Although the number of such static dataflow graphs can grow exponentially, application specific reachability analysis can be performed to include only a subset of all possible static graphs. The existing scheduling techniques for static dataflow models can then be leveraged for scheduling such static graphs. These schedules interact with each other dynamically. The GST framework discussed earlier provides an elegant way of representing such gen-

eralized schedules and simulating heterogeneous applications inside functional DIF. We believe the profiling results supplied by functional DIF could also provide valuable information for improving complex schedules automatically. We also plan to support parameterized dataflow modeling, which will permit more natural description of certain kinds of dynamic behavior, without departing from strong dataflow formalisms.

Acknowledgments

This research was sponsored in part by the U.S. National Science Foundation (Grant number 0720596), and the US Army Research Office (Contract number TCN07108, administered through Battelle-Scientific Services Program).

References

1. Lee, E.A., Messerschmitt, D.G.: Synchronous dataflow. Proceedings of the IEEE **75**(9) (September 1987) 1235–1245
2. Shen, C., Plishker, W., Bhattacharyya, S.S., Goldsman, N.: An energy-driven design methodology for distributing DSP applications across wireless sensor networks. In: Proceedings of the IEEE Real-Time Systems Symposium, Tucson, Arizona (December 2007) 214–223
3. Hemaraj, Y., Sen, M., Shekhar, R., Bhattacharyya, S.S.: Model-based mapping of image registration applications onto configurable hardware. In: Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers, Pacific Grove, California (October 2006) 1453–1457 Invited paper.
4. Plishker, W.: Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors. PhD thesis, University of California, Berkeley (January 2006)
5. Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional DIF for rapid prototyping. In: Proceedings of the International Symposium on Rapid System Prototyping, Monterey, California (June 2008) 17–23
6. Hsu, C., Corretjer, I., Ko., M., Plishker, W., Bhattacharyya, S.S.: Dataflow interchange format: Language reference for DIF language version 1.0, users guide for DIF package version 1.0. Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park (June 2007) Also Computer Science Technical Report CS-TR-4871.
7. Plishker, W., Sane, N., Kiemb, M., Bhattacharyya, S.S.: Heterogeneous design in functional DIF. In: Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation, Samos, Greece (July 2008) 157–166
8. Johnson, G.: LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control. McGraw-Hill School Education Group (1997)
9. The MathWorks Inc.: Using Simulink. Version 3 edn. (Jan 1999)
10. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static dataflow. IEEE Transactions on Signal Processing **44**(2) (February 1996) 397–408
11. Buck, J.T., Lee, E.A.: Scheduling dynamic dataflow graphs using the token flow model. In: In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing. (April 1993)

12. Hsu, C., Ko, M., Bhattacharyya, S.S.: Software synthesis from the dataflow interchange format. In: Proceedings of the International Workshop on Software and Compilers for Embedded Systems, Dallas, Texas (September 2005) 37–49
13. Hsu, C., Bhattacharyya, S.S.: Porting DSP applications across design tools using the dataflow interchange format. In: Proceedings of the International Workshop on Rapid System Prototyping, Montreal, Canada (June 2005) 40–46
14. Eker, J., Janneck, J., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S.R., Xiong, Y.: Taming heterogeneity - the Ptolemy approach. Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software **91**(1) (January 2003) 127–144
15. Haubelt, C., Falk, J., Keinert, J., Schlichter, T., Streubühr, M., Deyhle, A., Hadert, A., Teich, J.: A systemc-based design methodology for digital signal processing systems. EURASIP J. Embedded Syst. **2007**(1) (2007) 15–15
16. Eker, J., Janneck, J.: Caltrop—language report (draft). Technical memorandum, Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley California, Berkeley, CA (2002)
17. Kienhuis, B., Deprettere, E.F.: Modeling stream-based applications using the SBF model of computation. In: Proceedings of the IEEE Workshop on Signal Processing Systems. (September 2001) 385–394
18. Sriram, S., Bhattacharyya, S.S.: Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc. (2000)
19. Bhattacharya, B., Bhattacharyya, S.S.: Parameterized dataflow modeling for DSP systems. IEEE Transactions on Signal Processing **49**(10) (October 2001) 2408–2421
20. Berry, G.: Bottom-up computation of recursive programs. ITA **10**(1) (1976) 47–82
21. Ko, M., Zissulescu, C., Puthenpurayil, S., Bhattacharyya, S.S., Kienhuis, B., Deprettere, E.: Parameterized looped schedules for compact representation of execution sequences. In: Proceedings of the International Conference on Application Specific Systems, Architectures, and Processors, Steamboat Springs, Colorado (September 2006) 223–230
22. Murthy, P.K., Bhattacharyya, S.S.: Shared buffer implementations of signal processing systems using lifetime analysis techniques. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **20**(2) (February 2001) 177–198
23. Dalcolmo, J., Lauwereins, R., Ade, M.: Code generation of data dominated DSP applications for FPGA targets. In: Proceedings of the International Workshop on Rapid System Prototyping. (June 1998) 162–167