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 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 an example, 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.

Keywords: 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]).

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.

Leveraging our existing dataflow interchange format (DIF) package [4], we implement an extension to DIF based on a form of dataflow, called core function 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. 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.

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 [5][6][7]. Ideally, designers are able to find a match between their application and one of the well studied models, including cyclo-static dataflow (CSDF) [8], synchronous dataflow (SDF) [9], single-rate dataflow, homogeneous synchronous dataflow (HSDF), or a more complicated model such as boolean dataflow (BDF) [10].

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) [4], a standard language founded in dataflow semantics and tailored for DSP system design. It 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, DIF 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.

To utilize the DIF language, the DIF package has been built. Along with the ability to transform DIF descriptions into a manipulable internal representation, the DIF package contains graph utilities, optimization engines, algorithms that may prove useful properties of the application, and a C synthesis framework [11]. These facilities make the DIF package an effective environment for modeling dataflow applications, providing interoperability with other design environments, and developing new tools.

Beyond these features, DIF 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. The DIF package also has an infrastructure for porting applications from other dataflow tools to DIF. What is lacking in the existing DIF package 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 [12]. 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 [13]. Other languages specifically targeting actor descriptions such as CAL [14]. For complete functionality in Simulink [7], 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 [6] 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 [15]. 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 minimally-restricted specification of actor functionality, and support for efficient prototyping of static, quasi-static, and dynamic scheduling techniques. Each may 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

For a formalism able to support this level of heterogeneity, we derive a special case of enable-invoke dataflow [16] that we refer to as core functional dataflow (CFDF), which ensures that the application is deterministic. In this formalism, each actor has a set of *modes* in which it can execute. 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 *enabling function* for a is defined as:

$$\varepsilon_a : (T_a \times M_a) \to B, \tag{1}$$

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 $B = \{true, false\}$ 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 *invoking function* for an actor a is defined as:

$$\kappa_a : (I_a \times M_a) \rightarrow (O_a \times M_a), \tag{2}$$

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. Invoking an actor can in general change the mode of execution of the actor, so the invoking function also produces the next mode that is valid. This mode can then be subsequently checked by the enabling function, and if true for any mode, the actor may be invoked in that mode. If no mode is returned (i.e., an empty mode set is returned), the actor is forever disabled.

5 Translation to CFDF

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.

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 the switch actor modes in terms of tokens produced and consumed

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

6 Scheduling for a Heterogeneous Application

We use generalized schedule trees (GSTs) [17] to represent schedules generated by schedulers in functional DIF. The GST representation is a generalization of the (binary) schedule tree representation. 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 to be able to 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. To this end, we create a CSDF actor with two phases: reading the polynomial coefficients and then processing a block of x's to be evaluated.

To illustrate the problem of heterogeneous complexity, we suppose that a DSP application designer might use two PEA actors customized for different length polynomials. The overall PEA system is shown in Figure 1. 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.

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

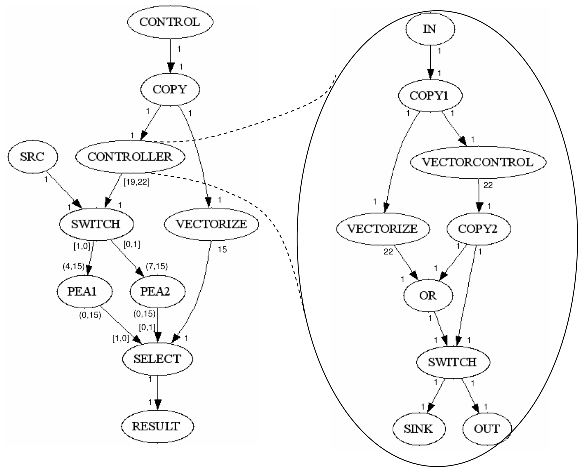

Fig. 1. A pictorial representation of the PEA application

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 2, 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. Figure 3 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, to reduce buffering.

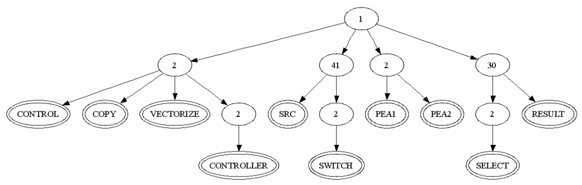

Fig. 2. Single appearance schedule for the dual PEA system

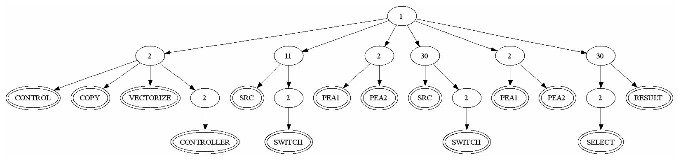

Fig. 3. Multiple appearance schedule for the dual PEA system

Table 2. Simulation times and max buffer sizes of the dual PEA design

Application style	Schedule	Simulation Time (s)	Max observed buffer size (tokens)
BDF Strict	Canonical	6.88	2,327,733
BDF Strict	Single appearance	1.72	1,729
BDF Strict	Multiple appearance	1.59	1,722
CFDF	Canonical	3.57	1,018,047
CFDF	Single appearance	0.95	1,791
CFDF	Multiple appearance	0.99	1,800

Results for these different styles of implementation with different schedules are summarized by Table 2. 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.

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 DIF package, 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.

We plan to build on this work in a number of ways. First, support for parameterized dataflow modeling will permit more natural description of certain kinds of dynamic behavior, without departing from strong dataflow formalisms. We are also interested in more general scheduling techniques that can automatically generate efficient schedules for such heterogeneous application. We believe the profiling results supplied by functional DIF could also provide valuable information for improving complex schedules automatically.

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.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. 36(1), 24–35 (1987)
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, pp. 214–223 (December 2007)
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, pp. 1453–1457 (October 2006) (invited paper)
4. Hsu, C., Corretjer, I., Ko., M., Plishker, W., Bhattacharyya, S.S.: Dataflow interchange format: Language reference for DIF language version 1.0, user's 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)
5. Flatscher, R.G.: Metamodeling in EIA/CDIF—meta-metamodel and metamodels. ACM Trans. Model. Comput. Simul. 12(4), 322–342 (2002)
6. Johnson, G.: LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control. McGraw-Hill School Education Group (1997)
7. The MathWorks Inc.: Using Simulink. Version 3 edn. (January 1999)
8. Bilsen, G., Engels, M., Lauwereins, R., Peperstraete, J.A.: Cyclo-static data flow. In: Proceedings of ICASSP, pp. 3255–3258 (May 1995)
9. Lee, E.A., Messerschmitt, D.G.: Synchronous dataflow. Proceedings of the IEEE 75(9), 1235–1245 (1987)
10. Buck, J.T.: Scheduling dynamic dataflow graphs with bounded memory using the token flow model. PhD thesis, Chair-Edward A. Lee (1993)
11. 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, pp. 37–49 (September 2005)
12. 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), 127–144 (2003)
13. 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), 15 (2007)
14. 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, CA (2002)
15. Kienhuis, B., Deprettere, E.F.: Modeling stream-based applications using the sbf model of computation. J. VLSI Signal Process. Syst. 34(3), 291–300 (2003)
16. Plishker, W., Sane, N., Kiemb, M., Anand, K., Bhattacharyya, S.S.: Functional dif for rapid prototyping. In: Proceedings of International Symposium on Rapid System Prototyping, Monterey, California, USA (June 2008)
17. 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, pp. 223–230 (September 2006)