In Proceedings of the International Symposium on Rapid System Prototyping, pages 17-23, Monterey, California, June 2008.

Functional DIF for Rapid Prototyping

William Plishker, Nimish Sane, Mary Kiemb, Kapil Anand, 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, kapil, ssb}@umd.edu

Abstract

Dataflow formalisms have provided designers of digital signal processing systems with optimizations and guarantees to arrive at quality prototypes quickly. As system complexity increases, designers are expressing more types of behavior in dataflow languages to retain these implementation benefits. While the semantic range of DSP-oriented dataflow models has expanded to cover quasi-static and dynamic applications, efficient functional simulation of such applications has not. Complexity in scheduling and modeling has impeded efforts towards functional simulation that matches the final implementation. We provide this functionality by introducing a new dataflow model of computation, called enable-invoke dataflow (EIDF), that supports flexible and efficient prototyping of dataflow-based application representations. EIDF permits the natural description of actors for dynamic and static dataflow models. We integrate EIDF into the dataflow interchange format (DIF) package and demonstrate the approach on the design of a polynomial evaluation accelerator targeting an FPGA implementation. Our experiments show that a design environment based on EIDF can achieve functionally-correct simulation compared to Verilog, allowing the application designer to arrive at a verified functional simulation faster, and therefore at a functional prototype much more quickly than traditional design practices.

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 [17]. As a result, dataflow languages are increasingly popular. Their diversity, portability, and intuitive appeal have extended them to many application areas (e.g., see [4,9,22]).

As system complexity 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 applications, efficient functional simulation of such applications 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 a correctly working functional prototype.

To quickly arrive at quality prototypes, designers must be able to describe their complex 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 the 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 appropriately captured, designers need to be able to evaluate static schedules (for high performance) alongside dynamic behavior without loosing semantic ground. With a properly-constructed schedule and a fully-described application, designers should be able to verify the functionality of a dataflow-based system. With such a feature set, designers should arrive at quality prototypes faster.

To address these designer needs, we introduce a new dataflow model called *enable-invoke dataflow* (EIDF). 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. Furthermore, building on our prior rapid prototyping work [10] in the dataflow interchange format (DIF) package [11], we propose and implement an extension to DIF based on a form of EIDF, enabling rapid prototyping of scheduling strategies as well as complete applications. This extension, called *functional DIF*, can prototype static, dynamic, and quasi-static schedules. To demonstrate our approach, we chose a representative ker-

nel from the software radio domain: a polynomial evaluation accelerator (PEA). While its behavior does not obviously fit into any of the popular forms of dataflow, we describe its functionality naturally with EIDF and show that it produces the same results as a prototype implementation on an FPGA.

This paper is organized into the following sections: Section 2 gives an overview of dataflow modeling and the tools that we have already developed. Section 3 touches on related dataflow work and Section 4 introduces our new dataflow model, EIDF. Section 5 discusses the restricted version of it that we implemented. Section 6 discusses the implementation of EIDF in our existing software framework. Section 7 demonstrates this implementation with a complex functional module, and Section 8 summarizes this work and discusses future directions.

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 [7, 13, 19]. Furthermore, Turing-complete DSP-oriented dataflow modeling approaches, such as Boolean dataflow (BDF) [3], are available to provide for full expressibility within the dataflow framework. Designers can find a match between their application and one of the well studied models, including cyclo-static dataflow (CSDF) [2], synchronous dataflow (SDF) [16], single-rate dataflow, homogeneous synchronous dataflow (HSDF), or a more complicated model such as multidimensional dataflow (MDSDF) [18] or BDF.

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* $src: E\rightarrow V$, maps edges to their source vertex, and a *sink function* $snk: E\rightarrow V$ 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) [11], a standard language founded in dataflow semantics and tailored for DSP system design. DIF is suitable as an interchange format for different dataflow-based DSP design tools because it provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification [10].

From the 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. Therefore, the dataflow semantics of a DSP application is unique in DIF regardless of any design tool used to originally enter the application specification. Moreover, DIF also provides syntax to specify design-tool-specific information, which is captured within the data structures associated with DIF intermediate representations.

2.3. The DIF Package

To utilize the semantics captured by describing applications in the DIF language, the DIF package was created. An overview is illustrated in Figure 1 (for a full explanation of it see [11]). Along with the ability to transform a DIF description into a manipulatable internal representation, the DIF package contains graph utilities, optimization engines, and algorithms that can prove useful properties of an application. These facilities make the DIF package an effective environment for modeling dataflow applications, providing interoperability with other design environments and developing new tools. To promote reuse, DIF provides common dataflow features so that developers and users of design tools can focus on the novel features and unique constraints associated with their design problems.

Beyond these features, DIF is also suitable as a design environment for implementing dataflow-based application representations. Developer productivity benefits from the tailored semantics and the dataflow tool suite. The internal

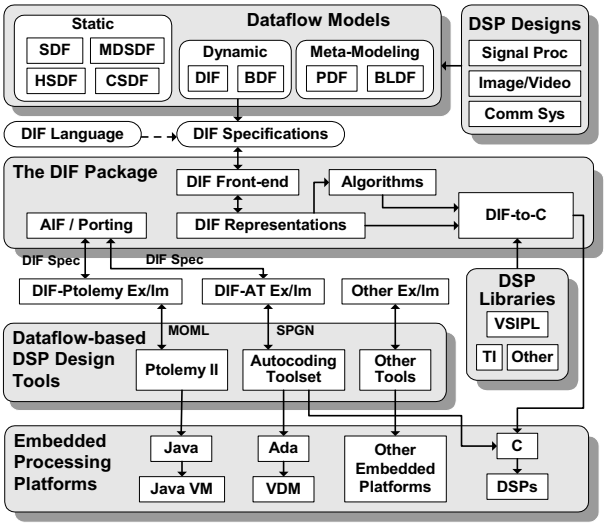

Fig. 1. DIF-based design flow.

representation can be turned into functional implementation with the DIF-to-C tool [12], which is an efficient and optimized code synthesis tool for SDF. What is lacking in the existing DIF package is the ability to simulate functional designs in the design environment. Such efforts would streamline the design process, allowing applications to be verified without going to implementation.

3. Related Work

A number of development environments utilize dataflow models to aid in the description and optimization of functional applications. Ptolemy II allows a diversity of dataflow-oriented and other models of computation [5]. 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 many models of computation. To describe an application with multiple models of computation, developers 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 actors only at their associated levels. 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 design.

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 [8]. Other languages specifically targeting actor descriptions such as CAL [6]. For complete functionality in Simulink [19], actors are described in the form of "S-functions." By describing them in a specific format, such that actors can be used in continuous, discrete, and hybrid systems. LABVIEW [13] 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 [14]. 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.

EIDF and functional DIF differ from these related efforts in dataflow-based design in their 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, EIDF allows a designer to describe actor functionality in an arbitrary set of fixed modes, instead of parcelling out actor behavior as side-effect free functions, a controller, and a transition function. EIDF is also more general than SBF as it permits multi-token reads, non-deterministic behavior, 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, and this a key motivation for the integrated actor- and scheduler-level prototyping considerations in functional DIF. We elaborate further on these features in Section 4.

4. Enable-Invoke Dataflow (EIDF)

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. 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:

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

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 $a \in A$ 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 at that time.

The *invoking function* for an actor a is defined as:

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

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 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 set of next modes that are valid from this element, captured here by

the power set of M_a in the function range. These modes can then be 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. Unlike the enabling function, the invoking function is considered to have consumed the tokens it reads (i.e. tokens are not available for a subsequent reads). No information regarding tokens (including token count) can be used in the invoke function unless the corresponding tokens are consumed.

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 [1, 24]).

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 [1], to yield parameterized EIDF (PEIDF). Such rigorous integration of dynamic parameterization into EIDF is a useful topic for future work.

5. Functional DIF

To utilize EIDF in an actual dataflow-based development environment, we focus on models that are deterministic as they are supported in the DIF package. The functional DIF (DIF with functional designs) package enables fast simulation and prototyping of scheduling strategies. It should be noted, that while actors in functional DIF may be implemented with functional languages (e.g. ML), we use the term "functional" to imply that simulation is functionally accurate, producing sequences of output values for a given sequence of input values.

5.1. Core Functional Dataflow

For a formalism customized to functional DIF, we derive a special case of the EIDF model developed in Section 4 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 (equation 2) can be modified as

$$\kappa_a^* : (I_a \times M_a) \rightarrow (O_a \times M_a) . \quad (3)$$

With this restricted form of invoking function, only one mode can meaningfully be interrogated by the enabling function, ensuring that the application is now deterministic.

5.2. 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 in Section 5.1, CFDF is a restricted form

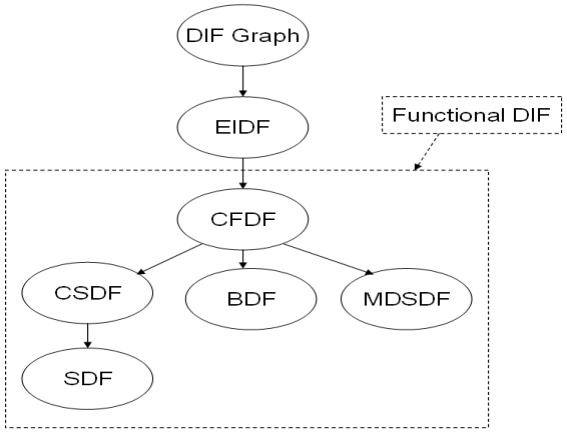

Fig. 2. Functional DIF semantic hierarchy.

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 for rapid prototyping of functional DIF.

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/Simulator

We have used the *generalized schedule tree* (GST) [15] 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 [20]. 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.

7. Design Example — Polynomial Evaluation

Polynomial evaluation is a commonly used primitive in various domains of signal processing, such as wireless communications. The following equation represents the evaluation of a polynomial P_i:

$$P_i(x) = \sum_{k=0}^{n_i} c_k \times x^k, \qquad (4)$$

where $c_1, c_2, \ldots, c_{n_i}$ are coefficients, x is the polynomial argument, and n_i is the degree of the polynomial. 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 (RST), store polynomial (STP), evaluate polynomial (EVP), and evaluate block (EVB). EVP is for a single evaluation, and EVB 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

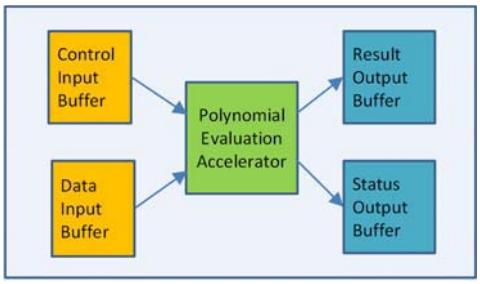

Fig. 3. Dataflow diagram of the PEA test bench.

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 1. The production and consumption columns describe the behavior.

Table 1: The modes of the PEA actor.

mode	consumes		produces	
	Control	Data	Result	Status
Normal	1	0	0	0
RST	0	0	0	0
STP	0	1	0	1
EVP	0	1	1	1
EVB	0	1	1	1

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 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.

A test bench for the PEA is shown in Figure 3. The Control Input Buffer (CIB) feeds instructions, while the Data Input Buffer (DIB) supplies coefficients and x to be evaluated. Results are put in the Result Output Buffer (ROB), and status into the Status Output Buffer (SOB).

Figure 4 shows pseudocode of the enable method for the

```
bool A.enable( CIB, DIB, mode ) {
   if( mode = Normal ) then
     if( there is a token in CIB ) then return true
   else if( mode = RST ) then
      return true;
   else if( mode = STP ) then
     if( there is 1 token in DIB ) then return true;
     else  return false;
     end if;
   else if( mode = EVP ) then
     if( there is 1 token in DIB ) then return true;
     else  return false;
     end if;
   else if( mode = EVB ) then
     if( there is 1 token in DIB ) then return true;
     else  return false;
     end if;
  end if;
}
```

Fig. 4. Pseudocode of the PEA enable method.

PEA actor. Since the execution condition differs according to the given mode, the enable method first checks the mode. The invoke method has a similar structure, but for each mode it consumes a fixed number of tokens, performs some computation, and returns the next mode to invoke.

Using an implementation based on the pseudocode outlined in Figure 4, 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 2. 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.

Table 2: 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 DIF Simulation Time (ms)	Speedup
Case 1	250	55	4.6x
Case 2	170	33	5.1x
Average	210	44	4.9x

Thus, when targeting an HDL implementation, prototyping in functional DIF is useful because it not only allows one to rapidly validate the overall functionality and high level dataflow architecture of a design, but also allows for a much faster simulation of complete system functionality. Once the functional DIF prototype has been completed and validated, the designer can proceed with greater confidence to tackling the lower level implementation details required

for the targeted HDL implementation. At the same time, the designer has a valuable reference implementation for functional validation of the HDL design as it evolves.

8. Conclusions and Future Work

In this work, we have presented a new dataflow formalism, called enable-invoke dataflow (EIDF), that facilitates the natural description of dataflow actors while retaining important properties of dataflow behavior. We integrated EIDF into the DIF package and demonstrated our approach on the design of a polynomial evaluation accelerator. Our experiments show that a design environment based on EIDF can achieve functionally-correct simulation compared to Verilog, allowing an application designer to arrive at a verified functional design faster, and therefore a functional prototype more quickly than traditional practices.

We plan to build on this work in a number of ways. First, support for parameterized dataflow modeling [1] will permit more natural description of certain kinds of dynamic behavior (such as the STP mode in the PEA application), without departing from strong dataflow formalisms. We will also explore automated support for constructing CFDF mode-sets for actors when sufficient information is available from the application description (e.g., from the specific form of dataflow that is being used or from specific values for actor or graph attributes).

Acknowledgements

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] B. Bhattacharya and S. S. Bhattacharyya. Quasi-static scheduling of reconfigurable dataflow graphs for DSP systems. *In Proceedings of the International Workshop on Rapid System Prototyping*, pages 84-89, Paris, France, June 2000.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *Proc. ICASSP*, pages 3255-3258, May 1995.

[3] J. T. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model*. Tech. Report UCB/ERL 93/69, Ph.D. Thesis, Dept. of EECS, University of California, Berkeley, CA 94720, 1993.

[4] C. Shen, W. Plishker, S. S. Bhattacharyya, and N. Goldsman. An energy-driven design methodology for distributing DSP applications across wireless sensor networks. *In Proceedings of the IEEE Real-Time Systems Symposium,* pages 214-223, Tucson, Arizona, December 2007.

[5] J. Eker, J. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming Heterogeneity - the Ptolemy Approach," *Proceedings of the IEEE*, v.91, No. 2, January 2003.

[6] J. Eker and J. Janneck, "CAL—Language report." Electronics Research Lab, Department of Electrical Engineering and Computer Sciences, 2002, Technical Memorandum, University of California at Berkeley California, Berkeley, CA 94720, USA}

[7] R. Flatscher, "Metamodeling in EIA/CDIF - Meta-Metamodel and Metamodels", *ACM Transactions on Modeling and Computer Simulation*, vol. 12, no.4, pp. 322-342, October 2002.

[8] C. Haubelt, J. Falk, J. Keinert, et al., "A SystemC-Based Design Methodology for Digital Signal Processing Systems," *EURASIP Journal on Embedded Systems*, vol. 2007, Article ID 47580, 2007.

[9] Y. Hemaraj, M. Sen, R. Shekhar, and S. S. Bhattacharyya. Model-based mapping of image registration applications onto configurable hardware. *In Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 1453-1457, Pacific Grove, California, October 2006.

[10] C. Hsu and S. S. Bhattacharyya. Porting DSP applications across design tools using the dataflow interchange format. In *Proceedings of the International Workshop on Rapid System Prototyping,* pages 40-46, Montreal, Canada, June 2005.

[11] C. Hsu, I. Corretjer, M. Ko., W. Plishker, and S. S. Bhattacharyya, "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.

[12] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. *In Proceedings of the International Workshop on Software and Compilers for Embedded Systems,* pages 37-49, Dallas, Texas, Sept. 2005.

[13] G. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control,* - McGraw-Hill School Education Group, 1997.

[14] B. Kienhuis and E. F. Deprettere. Modeling Stream-Based Applications Using the SBF Model of Computation. *Proceedings of the IEEE Workshop on Signal Processing Systems*, pages 385-394, September 2001.

[15] 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,* 55(6):3126-3138, June 2007

[16] E. A. Lee and D. G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235-1245, September 1987.

[17] E. A. Lee and D. G. Messerschmitt, "Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing," *IEEE Trans. on Computers*, January, 1987.

[18] E. A. Lee, "Representing and Exploiting Data Parallelism Using Multidimensional Dataflow Diagrams", *Proc. ICASSP*, 1993.

[19] Using Simulink,. The MathWorks Inc., Jan. 1999,Version 3.

[20] P. K. Murthy and S. S. Bhattacharyya. Shared buffer implementations of signal processing systems using lifetime analysis techniques. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* 20(2):177-198, Feb. 2001.

[21] Object Management Group, *Meta Object Facility (MOF) Specification*, v. 1.4, April 2002. In http://www.omg.org.

[22] W. Plishker. *Automated Mapping of Domain Specific Languages to Application Specific Multiprocessors*. PhD thesis, University of California, Berkeley, January, 2006.

[23] C. B. Robbins. *Using The MCCI Autocoding Toolset Tutorial*. Version 0.9a, Management, Communications & Control, Inc.

[24] S. Sriram and S. S. Bhattacharyya. Embedded Multiprocessors: Scheduling and Synchronization. Marcel Dekker, Inc., 2000.