In Proceedings of the International Conference on Acoustics, Speech, and Signal Processing, Dallas, Texas, March 2010. To appear.

SIMULATING DYNAMIC COMMUNICATION SYSTEMS USING THE CORE FUNCTIONAL DATAFLOW MODEL

Nimish Sane[1]*, Chia-Jui Hsu*[2]*, José Luis Pino*[2]*, and Shuvra S. Bhattacharyya*[1]

[1]Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742, USA
{nsane, ssb}@umd.edu

[2]Agilent Technologies, Inc.
Westlake Village, CA 91362, USA
{jerry_hsu, jpino}@agilent.com

ABSTRACT

The latest communication technologies invariably consist of modules with dynamic behavior. There exists a number of design tools for communication system design with their foundation in dataflow modeling semantics. These tools must not only support the functional specification of dynamic communication modules and subsystems but also provide accurate estimation of resource requirements for efficient simulation and implementation. We explore this trade-off — between flexible specification of dynamic behavior and accurate estimation of resource requirements — using a representative application employing an adaptive modulation scheme. We propose an approach for precise modeling of such applications based on a recently-introduced form of dynamic dataflow called core functional dataflow. From our proposed modeling approach, we show how parameterized looped schedules can be generated and analyzed to simulate applications with low run-time overhead as well as guaranteed bounded memory execution. We demonstrate our approach using the Advanced Design System from Agilent Technologies, Inc., which is a commercial tool for design and simulation of communication systems.

Index Terms— Digital signal processing, wireless communication, modeling and simulation, dataflow.

1. INTRODUCTION

Dataflow modeling is used extensively for designing digital signal processing (DSP) and communication applications. There are various existing design tools with their semantic foundations in dataflow modeling such as PEACE [1], SysteMoc [2], and Compaan/Laura [3]. Dataflow-oriented DSP design tools typically allow high-level application specification, software simulation, and possibly synthesis for hardware or software implementation. Dataflow modeling involves representing an application using a directed graph $G(V, E)$, where V is a set of vertices (nodes) and E is a set of edges. Any vertex $u \in V$ is called an *actor* and represents a specific computation block, while any directed edge $e(u, v) \in E$ is a first-in-first-out (FIFO) buffer that represents a communication link between the *source* $u \in V$ and the *sink* $v \in V$. The edge e can also have a non-negative integer *delay*, del(e), associated with it that represents the number of initial data tokens present in that buffer. Dataflow graphs operate based on data-driven execution in which an actor is executed (*fired*) only when it has sufficient amounts of data (numbers of "samples" or "data tokens") available on all of its inputs. During each firing, an actor consumes a certain number of tokens from its inputs and produces a certain number of tokens on its outputs. We refer to these numbers of tokens consumed and produced in each actor execution as the *consumption rate* and *production rate*, respectively. Usually production and consumption rate information is characterized in terms of individual input and output ports so that each port of an actor can in general have a different production or consumption rate characterization. Such characterizations can have constant values as in *synchronous dataflow* (*SDF*) [4] or more complex forms that are data-dependent or otherwise time-varying (e.g., see [5]). We refer to functional components for which the production or consumption rates may not remain constant during the application execution as *dynamic dataflow components*. A *schedule* is a sequence of actors in the dataflow graph G and represents the order in which actors are fired during execution of the dataflow graph.

There is generally a trade-off between the expressive power of the dataflow model being used and the compile-time (i.e., prior to execution or simulation) predictability that is available when analyzing specifications in that model. Although we would like to have as much expressive power as possible to best capture the dynamic nature of modern DSP and communication applications, this can lead to significant reductions in the ability to predict hardware and software resource requirements when targeting simulation or efficient implementation. Many of these applications are "mostly-static" hybrids in that they involve static dataflow components along with a relatively small proportion of dynamic components.

In this paper, we propose an approach to modeling and scheduling of such hybrid communication system applications in terms of a recently-introduced model of computation called *core functional dataflow* (*CFDF*) [5]. Section 3 explains in more detail the class of applications that we are considering in this paper, and introduces an *adaptive modulation scheme* (*AMS*), which we use as a case study. We also show how efficient *parameterized looped schedules* (*PLS*s) can be derived from the CFDF representations. The CFDF model provides sufficient expressive power to model any form of deterministic dynamic dataflow behavior as explained in Section 4. In Section 5, we develop an efficient scheduling approach for a restricted class of the CFDF applications. This restricted form is defined in a way that introduces a new trade-off point between expressive power and analysis potential that is useful for modeling of modern communication systems. Section 6 describes our experimental approach and presents the corresponding results.

2. RELATED WORK

The problem of modeling dynamic DSP applications has been studied previously and various forms of dataflow models with dynamic

constructs such as *Boolean dataflow* (*BDF*) have been suggested [6]. The problem of scheduling dynamic dataflow applications has also been studied, and important results have been established regarding bounded memory execution and compile-time scheduling (e.g., see [6, 7]). Most of these approaches employ scheduling schemes that suffer from significant run-time overhead, difficulties in code generation, and lack of compile-time predictability (e.g., for validating real-time signal processing performance). There exist scheduling schemes that identify specialized forms of dataflow from more general dataflow representations, and exploit such specialized *regions* to construct efficient quasi-static schedules (e.g., see [8]). These techniques, however, do not in general guarantee bounded memory execution for the entire input application.

A meta-modeling technique called *parameterized dataflow* (*PDF*) has been proposed to represent certain kinds of application dynamics [9]. This model can be used with any arbitrary dataflow graph format that has a well-defined notion of a *schedule iteration*. The model of computation that results from integrating PDF with SDF is called *parameterized synchronous dataflow* (*PSDF*). It supports limited forms of dynamic behavior and has more compile-time predictability than more general kinds of dynamic dataflow models such as BDF. A useful feature of PSDF is its capability of efficient quasi-static scheduling in terms of PLSs [9]. This class of schedules allows for flexible, compact specification of nested loop structures, where loop iteration counts can be either constant values or symbolic expressions in terms of dynamic parameters in the underlying dataflow graph. While PDF is useful for many kinds of signal processing applications, it imposes significant restrictions on how applications are modeled (e.g., in terms of hierarchies of cooperating *init*, *subinit*, and *body* graphs [9]), and in general, major changes in the user interface are required to provide direct support for PDF in a design tool.

In contrast to the approaches described above, this paper aims at providing PLS-based bounded memory scheduling while operating within a semantic framework that can be integrated more directly into existing design tools compared to the more hierarchical semantic structure of PDF representations. This semantic framework is that of the CFDF, which is described further in Section 4.

3. DYNAMIC COMMUNICATION APPLICATIONS

The approach that we develop in this paper is targeted toward a useful, restricted class of dataflow-based applications or subsystem modules, the graphical representation of which can be reduced to the form shown in Fig. 1(a). Here S_1, S_2, and S_3 denote regions consisting of SDF actors that can be SDF-clustered, while actors D_1 and D_2 have dynamic behavior. We say that a group of actors can be *SDF-clustered* if its component actors can be scheduled together (i.e., the group can be scheduled as a single unit in the overall schedule for the graph) without introducing deadlock [10]. The dataflow edges $e_1, e_2, \ldots, e_5$ denote FIFO buffers. The D on edge e_5 denotes the delay, del(e_5), associated with it. In our targeted class of applications, we assume that the production and consumption rates k, x, y, z, q are positive integer constants, while the consumption rate c_i and production rate p_i can vary over finite ranges of positive integer values with known upper bounds $c_{\max}$ and $p_{\max}$, respectively. The subscript i in the symbols p_i and c_i represents the dependence of this production and consumption rate pair on the actor execution index i — thus, p_i represents the number of tokens produced onto e_4 in the ith execution (*firing*) of D_2, and c_i represents the number of tokens consumed from e_1 during the ith firing of D_1.

We consider one specific dynamic communication application,

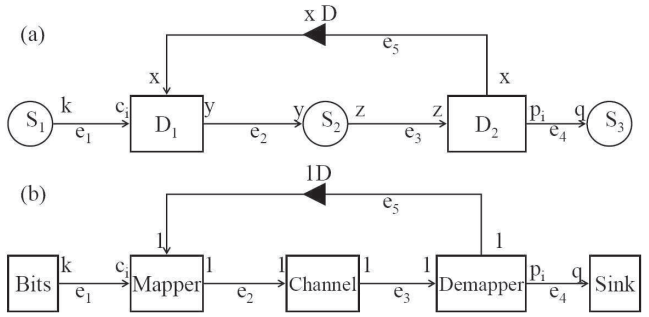

Fig. 1. Dataflow graph for (a) the generic class of applications under consideration and (b)simplified adaptive modulation scheme.

the AMS, which is an important part of modern wireless standards such as the *worldwide interoperability for microwave access* (*WiMAX*) [11] and *3rd generation partnership project — long term evolution* (*3GPP—LTE*) [12]) standards. Fig. 1(b) shows a simplified representation of the AMS. The *mapper* maps the bit(s) from the input bitstream to a symbol for transmission over the *channel*, while the *demapper* outputs one or more bits for each of the symbols received from the *channel*. The number of bits per symbol depends upon the modulation and demodulation schemes used by the *mapper* and *demapper*, respectively. The *mapper* receives feedback from the *demapper* indicating the result of channel estimation and accordingly selects one of the modulation schemes to be employed, which in turn determines the number of bits per symbol. Hence, the number of tokens consumed (produced) by the *mapper* (*demapper*) from the buffer e_1 (e_4) in general can vary from one invocation to the next. We have used the AMS as a representative application to demonstrate our approach. This approach, however, can be readily extended to any other application that exhibits the general dataflow structure illustrated in Fig. 1(a) (e.g., prediction error filters [13], systems for frequency domain block adaptive filtering [14]).

4. CORE FUNCTIONAL DATAFLOW

To provide a precise framework for modeling and scheduling the applications described in Section 3, we employ the CFDF model, which can be used to model deterministic dynamic dataflow behaviors (i.e., dynamic behaviors in which a given set of input streams always produces a unique set of output streams) [5]. It supports flexible and efficient prototyping of dataflow-based application representations and permits natural description of both dynamic and static dataflow actors [5]. CFDF semantics can be viewed as a "deterministic dataflow subset" of *enable-invoke dataflow* semantics, which require that actor specification be divided into separate *enable* and *invoke* functions [5] (described below). A CFDF actor a also has a set M_a of valid modes in which it can execute. When the actor a executes in a mode $m \in M_a$, it consumes (produces) a fixed number of tokens from its inputs (onto its outputs), but the number of tokens consumed and produced by the actor a can vary across different modes in M_a. The separation of *enable* and *invoke* capabilities helps in prototyping efficient scheduling techniques. The *enable* function is designed to be used as a "hook" for dynamic or quasi-static scheduling techniques to rapidly query actors at run-time, and check whether or not they are executable. It only checks for the availability of sufficient input data to allow an actor to fire in its current mode, and does not consume any tokens from the actor in-

Mode	Consumption rate		Production rate
	e_1	e_5	e_2
control	0	1	0
QPSK	2	0	1
16QAM	4	0	1
64QAM	6	0	1

Table 1. Valid modes for the *mapper* actor

puts. The current mode of an actor is always unique in CFDF, so this check of "data sufficiency" is unambiguous. The *invoke* function, on the other hand, consumes as many tokens from the inputs as specified by its mode of execution, and correspondingly produces the specified numbers of tokens onto the actor outputs. It can generally change the mode of the actor by returning a valid mode of execution in which the actor should be fired during its next invocation. Thus, actors proceed deterministically to a unique "next mode" of execution whenever they are enabled.

To apply the CFDF model for modeling the AMS, we focus our attention on the *dynamic mapper* and *demapper* actors. The remaining actors are SDF actors and can be modeled easily as CFDF actors with just one valid mode each. Table 1 shows the possible modes for a generic *dynamic mapper* actor with their respective production and consumptions rates. It has a mode corresponding to each of the possible modulation schemes being employed (here QPSK, 16QAM, and 64QAM), and an additional mode called *control*. In the control mode, the mapper actor reads a channel quality indicator token from the feedback edge e_5. This information is then used to determine the modulation scheme to be employed, and the *invoke* function returns (as the next mode value) the mode that corresponds to this scheme. The *demapper* actor can be modeled in a similar manner.

5. SCHEDULING

We consider the problem of scheduling the class of dynamic applications that is described in Section 3. We use the simple, but generally-applicable, *canonical scheduler* as our base scheduler and proceed to show how efficient PLSs having bounded memory execution can be generated.

5.1. Canonical Schedules

The canonical schedule (*CS*) for a CFDF graph is a simple schedule that consists of a *guarded execution* of each actor in the graph once per schedule iteration. In this context, a guarded execution of an actor means that an actor is fired (by calling its *invoke* function) only if its *enable* function returns `true`. We use the CS as a base schedule against which we compare the performance of our PLSs.

5.2. Parameterized Looped Schedules

As described in Section 2, we seek to generate efficient PLSs to reduce the run-time overhead associated with dynamic scheduling. Such quasi-static schedules are also useful from a code generation perspective as the only dynamic components of such schedules are the loop iteration counts. Our approach finds static regions in the application graph that can be clustered and completely scheduled at compile-time. It then proceeds to identify the dynamic components along with the corresponding static components, which must execute varying numbers of times in relation to the dynamic components.

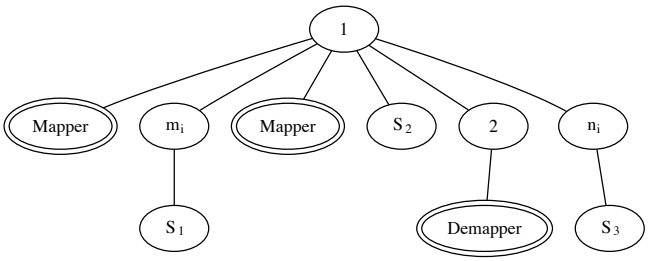

Fig. 2. Valid PLS for the application in Fig. 1(b)

We then merge the appropriately-iterated static and dynamic components into a single PLS. The following sequence of steps outlines our algorithm for PLS construction:

1. Identify SDF components in the dataflow graph and cluster them individually to obtain SDF-clustered regions S_1, S_2, and S_3. This step can be performed efficiently since the specification of a CFDF actor mode includes the associated production or consumption rate for each actor port.
2. Use established SDF scheduling techniques [10, 15] for scheduling the SDF regions identified in step 1, assuming that a valid consistent schedule exists for each of the SDF subgraphs [4].
3. Identify the pair D_1 and D_2 of actors with dynamic behavior, and determine which of the SDF sub-schedule loop (iteration) counts are dependent on D_1 and D_2.
4. Combine static sub-schedules into a PLS in which parameterized loop count expressions are set up at compile time, and symbolic parameters in these expressions are varied at run-time.

Fig. 2 illustrates a PLS for the dataflow graph in Fig. 1(b) that is derived using our approach to PLS construction. This schedule is shown using the *generalized schedule tree* (*GST*) representation [16]. An internal node of a GST denotes a loop count, while a leaf node represents an actor. The execution of a schedule involves traversing the GST in a depth-first manner, and during this traversal, the sub-schedule rooted at any internal node is executed as many times as specified by the loop count of that node. As can be seen from the GST in Fig. 2, CFDF actors use *guarded execution* (shown by two concentric ellipses). Other SDF actors are fired using *unguarded execution* in which the actor is fired without checking if it is enabled (enabling is guaranteed through a carefully constructed PLS). The values of m_i and n_i are determined dynamically by the simulator based on the current modes of the *mapper* and *demapper* actors, respectively. Since the mode of an actor is visible to the simulator (through a flexible mode-querying mechanism in our implementation of CFDF), it can be used to set loop counts based on dynamically-changing execution state of the actor.

For the class of applications targeted in this paper (see Section 3) and PLSs generated using the above algorithm, the number of tokens accumulated on the edge e_1 (e_4) after the ith iteration is related to m_i and c_i (n_i and p_i). These expressions can then be used to prove that the numbers of tokens accumulated on edges e_1 and e_4 are bounded by $k+c_{\max}-1$ and $q+p_{\max}-1$, respectively. Together with bounds that are derived based on the static dataflow properties of the other edges, this leads to a bound on total buffer memory requirement that can be computed at compile-time. Such bounds provide for more efficient execution or simulation (since dynamic memory allocation

(a)

Sink control condition	Average simulation time (sec)		Reduction (%)
	CS	PLS	
10000	2.468	1.412	42.79
20000	5.000	2.849	43.02
50000	12.458	7.109	42.94

(b)

Sink control condition	Total buffer requirement (number of tokens)		
	CS	PLS(Experiments)	PLS(Theory)
10000	28	21	21
20000	30	21	21
50000	40	21	21

Table 2. (a) Average simulation time and (b) total buffer requirement for different sink control conditions (numbers of tokens consumed by the sink) for CS and PLS.

is not required) as well as enhanced predictability and reliability. We omit the detailed proof due to space limitations.

6. RESULTS

We have implemented our approaches to CFDF modeling and PLS construction using the *Advanced Design System* (*ADS*) tool from the Agilent Technologies, Inc. [17]. We have employed the CFDF model for dynamic actors along with the existing SDF based actors in the Agilent ADS. We have implemented the AMS application shown in Fig. 1(b). The results of our experiments for different sink control conditions (the total number of tokens that must be consumed by the sink during the simulation) are shown in Table 2. As evident from the results, PLSs exhibit significant reductions in run-time overhead over CSs, which leads to improvements in average simulation time up to 43%. Our experiments not only confirm the theoretical buffer bounds for PLSs estimated using the results mentioned in Section 5.2 but also demonstrate significant reductions in the total buffer memory requirements over the CS, especially for larger values of sink control conditions (i.e., longer simulations).

7. CONCLUSION

We have employed CFDF semantics to model a class of signal flow topologies that is important for modern communication systems. Our approach identifies the underlying static components in the application, systematically integrates the well-established compile-time scheduling techniques for SDF graphs with more flexible CFDF semantics, and uses combined CFDF/SDF analysis to generate PLSs that have significantly reduced run-time overhead, guaranteed memory bounds, and reduced memory requirements. Our approach therefore provides robust simulation of dynamic communication applications without major limitations on compile-time predictability and efficient scheduling. Useful directions for further work include the exploration of more general CFDF topologies as targets for PLS construction, and the application of our methods to optimized hardware and software synthesis.

8. REFERENCES

[1] S. Kwon, H. Jung, and S. Ha, "H.264 decoder algorithm specification and simulation in simulink and PeaCE," in *Proceedings of the International SoC Design Conference*, October 2004, pp. 9–12.

[2] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubhr, A. Deyhle, A. Hadert, and J. Teich, "A SystemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, 2007, article ID 47580, 22 pages, doi 10.1155/2007/47580.

[3] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, "System design using Kahn process networks: the Compaan/Laura approach," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, February 2004.

[4] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Transactions on Computers*, February 1987.

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

[6] J. T. Buck, *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*, Ph.D. thesis, EECS Department, University of California, Berkeley, 1993.

[7] T. M. Parks, *Bounded Scheduling of Process Networks*, Ph.D. thesis, EECS Department, University of California, Berkeley, 1995.

[8] W. Plishker, N. Sane, and S. S. Bhattacharyya, "A generalized scheduling approach for dynamic dataflow applications," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, Nice, France, April 2009, pp. 111–116.

[9] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling of DSP systems," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, June 2000, pp. 1948–1951.

[10] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[11] J. G. Andrews, A. Ghosh, and R. Muhamed, *Fundamentals of WiMAX: understanding broadband wireless networking*, Prentice Hall, 2007.

[12] 3GPP TS 36.211 V8.7.0 (2009-05), *Physical channels and modulation*, 2009.

[13] S. Haykin, *Adaptive filter theory*, Prentice-Hall, Inc., 1996.

[14] J. J. Shynk, "Frequency-domain and multirate adaptive filtering," *IEEE Signal Processing Magazine*, vol. 9, no. 1, pp. 14–37, January 1992.

[15] C. Hsu, S. Ramasubbu, M. Ko, J. L. Pino, and S. S. Bhattacharyya, "Efficient simulation of critical synchronous dataflow graphs," in *Proceedings of the Design Automation Conference*, San Francisco, California, July 2006, pp. 893–898.

[16] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere, "Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation," *IEEE Transactions on Signal Processing*, vol. 55, no. 6, pp. 3126–3138, June 2007.

[17] J. L. Pino and K. Kalbasi, "Cosimulating synchronous DSP applications with analog RF circuits," in *IEEE Asilomar Conference Signals, Systems, and Computers*, Pacific Grove, California, November 1998, vol. 2, pp. 1710–1714.