OpenDF – A Dataflow Toolset for Reconfigurable Hardware and Multicore Systems

Shuvra S. Bhattacharyya
Dept. of ECE and UMIACS
University of Maryland, College Park, MD 20742
USA

Gordon Brebner, Jörn W. Janneck
Xilinx Research Labs
San Jose, CA 95123
USA

Johan Eker, Carl von Platen
Ericsson Research
Mobile Platforms
SE-221 83, Lund
Sweden

Marco Mattavelli
Microelectronic Systems Lab
EPFL
CH-1015 Lausanne
Switzerland

Mickaël Raulet
IETR/INSA Rennes
F-35043, Rennes
France

Abstract

This paper presents the OpenDF framework and recalls that dataflow programming was once invented to address the problem of parallel computing. We discuss the problems with an imperative style, von Neumann programs, and present what we believe are the advantages of using a dataflow programming model. The CAL actor language is briefly presented and its role in the ISO/MPEG standard is discussed. The Dataflow Interchange Format (DIF) and related tools can be used for analysis of actors and networks, demonstrating the advantages of a dataflow approach. Finally, an overview of a case study implementing an MPEG-4 decoder is given.

1 Introduction

Time after time, the uniprocessor system has managed to survive in spite of rumors of its imminent death. Over the last three decades hardware engineers have been able to achieve performance gains by increasing clock speed, and introducing cache memories and instruction level parallelism. However, current developments in the hardware industry clearly shows that this trend is over. The frequency in no longer increasing, but instead the number of cores on each CPU is. Software development for uniprocessor systems is completely dominated by imperative style programming models, such as C or Java. And while they provide a suitable abstraction level for uniprocessor systems, they fail to do the same in a multicore setting. In a time when new hardware meant higher clock frequencies, old programs almost always ran faster on more modern equipment. However, this is not true when programs written for single core system execute on multicore. And the bad news is that there is no easy way of modifying them. Tools such as OpenMP will help the transition, but likely fail to utilize the full potential of multicore systems.

Over the years considerable attention has been put to the data flow modeling, which is a programming paradigm proposed in the late 60s, as a means to address parallel programming. It is well researched area with a number of interesting results pertaining to parallel computing. Many modern forms of computation are very well suited for data flow description and implementation, examples are complex media coding [1], network processing [2], imaging and digital signal processing [3], as well as embedded control [4]. Together with the move toward parallelism, this represents a huge opportunity for data flow programming.

2 Why C etc. Fail

Before diving into dataflow matters, we will give a brief motivation why a paradigm shift is necessary. The control over low-level detail, which is considered a merit of C, tends to over-specify programs: not only the algorithms themselves are specified, but also how inherently parallel computations are sequenced, how inputs and outputs are passed between the algorithms and, at a higher level, how computations are mapped to threads, processors and application-specific hardware. It is not always possible to recover the original knowledge about the program by means of analysis and the opportunities for restructuring transformations are limited.

Code generation is constrained by the requirement of preserving the semantic effect of the original program. What constitutes the semantic effect of a program depends on the source language, but loosely speaking some observable properties of the program's execution are required to be invariant. Program analysis is employed to identify the set of admissible transformations; a code generator is required to be conservative in the sense that it can only perform a particular transformation when the analysis results can be used to prove that the effect of the program is preserved. *Dependence analysis* is one of the most challenging tasks of high-quality code generation (for instance see [5]). It determines a set of constraints on the order, in which the computations of a program may be performed. Efficient utilization of modern processor architectures heavily depends on dependence analysis, for instance:

- To determine efficient mappings of a program onto multiple processor cores (*parallelization*),
- to utilize so called SIMD or "multimedia" instructions that operate on multiple scalar values simultaneously (*vectorization*), and
- to utilize multiple functional units and avoid pipeline stalls (*instruction scheduling*).

Determining (a conservative approximation of) the dependence relation of a C program involves *pointer analysis*. Since the general problem is undecideable, a trade-off will always have to be made between the precision of the analysis and its resource requirements [6].

3 Dataflow Networks

A dataflow program is defined as a directed graph, where the nodes represent computational units and the arcs represent the flow of data. The lucidness of dataflow graphs can be deceptive. To be able to reason about the effect of the computations performed, the dataflow graph has to be put in the context of a computation model, which defines the semantics of the communication between the nodes. There exists a variety of such models, which makes different trade-offs between expressiveness and analyzability. Of particular interest are Kahn process networks [7], and synchronous dataflow networks [8]. The latter is more constrained and allows for more compile-time analysis for calculation of static schedules with bounded memory, leading to synthesized code that is particularly efficient. More general forms of dataflow programs are usually scheduled dynamically, which induces a run-time overhead.

It has been shown that dataflow models offer a representation that can effectively support the tasks of parallelization [8] and vectorization [9]—thus providing a practical means of supporting multiprocessor systems and utilizing vector instructions.

3.1 Actors

The fundamental entity of this model is an *actor* [10], also called dataflow actor with firing. Dataflow graphs, called *networks*, are created by means of connecting the *input* and *output ports* of the actors. Ports are also provided by networks, which means that networks can nested in a hierarchical fashion. Data is produced and consumed as *tokens*, which could correspond to samples or have a more complex structure. This model has the following properties:

- **Strong encapsulation.** Every actor completely encapsulates its own state together with the code that operates on it. No two actors ever share state, which means that an actor cannot directly read or modify another actor's state variables. The only way actors can interact is through streams, directed connections they use to communicate data tokens.
- **Explicit concurrency.** A system of actors connected by streams is explicitly concurrent, since every single actor operates independently from other actors in the system, subject to dependencies established by the streams mediating their interactions.
- **Asynchrony, untimedness.** The description of the actors as well as their interaction does not contain specific real-time constraints (although, of course, implementations may).

4 The CAL Actor Language

CAL [11] is a domain-specific language that provides useful abstractions for dataflow programming with actors. CAL has been used in a wide variety of applications and has been compiled to hardware and software implementations, and work on mixed HW/SW implementations is under way. Below we will give a brief introduction to some key elements of the language.

4.1 Basic Constructs

The basic structure of a CAL actor is shown in the `Add` actor below, which has two input ports `t1` and `t2`, and one output port `s`, all of type `T`. The actor contains one *action* that consumes one token on each input ports, and produces one token on the output port. An action may *fire* if the availability of tokens on the input ports matches the *port patterns*, which in this example corresponds to one token on both ports `t1` and `t2`.

```
actor Add() T t1, T t2 ⇒ T s :
  action [a], [b] ⇒ [sum]
  do
     sum := a + b;
  end
end
```

An actor may have any number of actions. The untyped `Select` actor below reads and forwards a token from either port `A` or `B`, depending on the evaluation of guard conditions. Note that each of the actions have empty bodies.

```
actor Select () S, A, B ⇒ Output:

    action S: [sel], A: [v] ⇒ [v]
    guard sel end

    action S: [sel], B: [v] ⇒ [v]
    guard not sel end
end
```

An action may be labeled and it is possible to constrain the legal firing sequence by expressions over labels. In the `PingPongMerge` actor, see below, a finite state machine *schedule* is used to force the action sequence to alternate between the two actions `A` and `B`. The schedule statement introduces two states `s1` and `s2`.

```
actor PingPongMerge () Input1, Input2 ⇒ Output:
  A: action Input1: [x] ⇒ [x] end
  B: action Input2: [x] ⇒ [x] end

  schedule fsm s1:
    s1 (A) --> s2;
    s2 (B) --> s1;
  end
end
```

The `Route` actor below forwards the token on the input port `A` to one of the three output ports. Upon instantiation it takes two parameters, the functions `P` and `Q`, which are used as predicates in the guard conditions. The selection of which action to fire is in this example not only determined by the availability of tokens and the guards conditions, by also depends on the `priority` statement.

```
actor Route (P, Q) A ⇒ X, Y, Z:

    toX: action [v] ⇒ X: [v]
         guard P(v) end

    toY: action [v] ⇒ Y: [v]
         guard Q(v) end

    toZ: action [v] ⇒ Z: [v] end

    priority
        toX > toY > toZ;
    end
end
```

For an in-depth description of the language, the reader is referred to the language report [11]. A large selection of example actors is available at the OpenDF repository, among them the MPEG-4 decoder discussed below.

4.2 Networks

A set of CAL actors are instantiated and connected to form a CAL application, i.e. a CAL network. Figure 1 shows a simple CAL network `Sum`, which consists of the previously defined `Add` actor and the delay actor shown below.

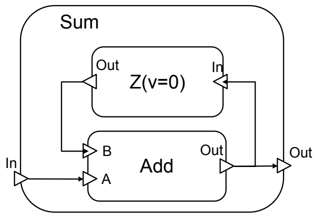

Figure 1. A simple CAL network.

```
actor Z (v) In ⇒ Out:

  A: action ⇒ [v] end
  B: action [x] ⇒ [x] end

  schedule fsm s0:
    s0 (A) --> s1;
    s1 (B) --> s1;
  end
end
```

The source that defined the network `Sum` is found below. Please, note that the network itself has input and output ports and that the instantiated entities may be either actors or other networks, which allows for a hierarchical design.

```
network Sum () In ⇒ Out:

entities
  add = Add();
  z = Z(v=0);

structure
  In --> add.A;
  z.Out --> add.B;

  add.Out --> z.In;

  add.Out -- > Out;
end
```

4.3 ISO-MPEG standardisation

The data-driven programming paradigm of CAL dataflow lends itself naturally to describing the processing of media streams that pervade the world of media coding. In addition, the strong encapsulation afforded by the actor model provides a solid foundation for the modular specification of media codecs.

MPEG has produced several video coding standards such as MPEG-1, MPEG-2, MPEG-4 Video, AVC and SVC. However, the past monolithic specification of such standards (usually in the form of C/C++ programs) lacks flexibility and does not allow to use the combination of coding algorithms from different standards enabling to achieve specific design or performance trade-offs and thus fill, case by case, the requirements of specific applications. Indeed, not all coding tools defined in a *profile@level* of a specific standard are required in all application scenarios. For a given application, codecs are either not exploited at their full potential or require unnecessarily complex implementations. However, a decoder conformant to a standard has to support all of them and may results in a non-efficient implementation.

So as to overcome the limitations intrinsic of specifying codecs algorithms by using monolithic imperative code, CAL language has been chosen by the ISO/IEC standardization organization in the new MPEG standard called Reconfigurable Video Coding (RVC) (ISO/IEC 23001-4 and 23002-4). RVC is a framework allowing users to define a multitude of different codecs, by combining together actors (called coding tools in RVC) from the MPEG standard library written in CAL, that contains video technology from all existing MPEG video past standards (i.e. MPEG-2, MPEG- 4, etc.). The reader can refer to [12] for more information about RVC. CAL is used to provide the reference software for all coding tools of the entire library. The essential elements of the RVC framework include:

- the standard Video Tool Library (VTL) which contains video coding tools, also named Functional Units (FU). CAL is used to describe the algorithmic behaviour of the FUs that end to be video coding algorithmic components self contained and communicating with the external world only by means of input and output ports.
- a language called Functional unit Network Language (FNL), an XML dialect, used to specify a decoder configuration made up of FUs taken from the VTL and the connections between the FUs.
- a MPEG-21 Bitstream Syntax Description Language (BSDL) schema which describes the syntax of the bitstream that a RVC decoder has to decode. A BSDL to CAL translator is under development as part of the OpenDF effort.

In summary the components and processes that lead to the specification and implementation of a new MPEG RVC decoder are based on the CAL dataflow model of computation and are:

- a Decoder Description (DD) written in FNL describing the architecture of the decoder, in terms of FUs and their connections.
- an Abstract Decoder Model (ADM), a behavioral (CAL) model of the decoder composed of the syntax parser specified by the BSDL schema, FUs from the VTL and their connections.
- the final decoder implementation that is either generated by substituting any proprietary implementation, conformant in terms of I/O behavior, of the standard RVC FUs, or obtained directly from the ADM by generating SW and/or HW implementations by means of appropriate synthesis tools.

Thus, based on CAL dataflow formalism, designers can build video coding algorithm with a set of self-contained modular elements coming from the MPEG RVC standard library (VTL). However, the new CAL based specification formalism, not only provide the flexibility required by the process itself of specifying a standard video codec, but also yields a specification of such standard that is the appropriate starting point for the implementation of the codec on the new generations of multicore platforms. In fact the RVC ADM is nothing else that a CAL datatflow specification that implicitly expose all concurrency and parallelism intrinsic to the model, features that classical generic specifications based on imperative languages have not provided.

5 Tools

CAL is supported by a portable interpreter infrastructure that can simulate a hierarchical network of actors. This interpreter was first used in the Moses[1] project. Moses features a graphical network editor, and allows the user to monitor actors execution (actor state and token values). The project being no longer maintained, it has been superseded by the Open Dataflow environment (OpenDF[2] for short).

OpenDF is also a compilation framework. Today there exists a backend for generation of HDL(VHDL/Verilog) [13], and another backend for that generates C for integration with the SystemC tool chain [14]. A third backend targeting ARM11 and embedded C is under development [15] as part of the EU project ACTORS[3]. It is also possible to simulate CAL models in the Ptolemy II[4] environment.

5.1 Analysis Support

A related tool is the dataflow interchange format (DIF), which is a textual language for specifying mixed-grain dataflow representations of signal processing applications, and TDP[5] (the DIF package), which is a software tool for analyzing DIF specifications. A major emphasis in DIF and TDP is support for working with and integrating different kinds of specialized dataflow models of computation and their associated analysis techniques. Such functionality is useful, for example, as a follow-on step to the automated detection of specialized dataflow regions in CAL networks. Once such regions are detected, they can be annotated with corresponding DIF keywords — e.g., CSDF

[1]http://www.tik.ee.ethz.ch/ moses/
[2]http://opendf.sourceforge.net
[3]http://www.actors-project.eu
[4]http://ptolemy.eecs.berkely.edu
[5]http://www.ece.umd.edu/DSPCAD/dif

(cyclo-static dataflow) and SDF (synchronous dataflow) — and then scheduled and integrated with appropriate TDP-based analysis methods. Such a linkage between CAL and TDP is under active development as a joint effort between the CAL and DIF projects.

A particular area of emphasis in TDP is support for developing efficient coarse-grain dataflow scheduling techniques. For example, the generalized schedule tree representation in TDP provides an efficient format for storing, manipulating, and viewing schedules [16], and the functional DIF dataflow model provides for flexible prototyping of static, dynamic, and quasi-static scheduling techniques [3]. Libraries of static scheduling techniques and buffer management models for SDF graphs, as well as an SDF-to-C translator are also available in TDP [17]. The set of dataflow models that are currently recognized and supported explicitly in the DIF language and TDP include Boolean dataflow [18], enable-invoke dataflow [3], CSDF [19], homogeneous synchronous dataflow [8, 20], multidimensional synchronous dataflow [21], parameterized synchronous dataflow [22], and SDF [8]. These alternative dataflow models have useful trade-offs in terms of expressive power, and support for efficient static or quasi-static scheduling, as well as efficient buffer management. The set of models that is supported in TDP, as well as the library of associated analysis techniques are expanding with successive versions of the TDP software.

The initial focus in integrating TDP with CAL is to automatically-detect regions of CAL networks that conform to SDF semantics, and can leverage the significant body of SDF-oriented analysis techniques in TDP. In the longer term, we plan to target a range of different dataflow models in our automated "region detection" phase of the design flow. This appears significantly more challenging as most other models are more complex in structure compared to SDF; however, it can greatly increase the flexibility with which different kinds of specialized, streaming-oriented dataflow analysis techniques can be leveraged when synthesizing hardware and software from CAL networks.

6 Why dataflow might actually work

Scalable parallelism. In parallel programming, the number of things that are happening at the same time can scale in two ways: It can increase with the size of the problem or with the size of the program. Scaling a regular algorithm over larger amounts of data is a relatively well-understood problem, while building programs such that their parts execute concurrently without much interference is one of the key problems in scaling the von Neumann model. The explicit concurrency of the actor model provides a straightforward parallel composition mechanism that tends to lead to more parallelism as applications grow in size, and scheduling techniques permit scaling concurrent descriptions onto platforms with varying degrees of parallelism.

- **Modularity, reuse.** The ability to create new abstractions by building reusable entities is a key element in every programming language. For instance, object-oriented programming has made huge contributions to the construction of von Neumann programs, and the strong encapsulation of actors along with their hierarchical composability offers an analog for parallel programs.
- **Scheduling.** In contrast to procedural programming languages, where control flow is made explict, the actor model emphasizes explicit specification of concurrency.
- **Portability.** Rallying around the pivotal and unifying von Neumann abstraction has resulted in a long and very successful collaboration between processor architects, compiler writers, and programmers. Yet, for many highly concurrent programs, portability has remained an elusive goal, often due to their sensitivity to timing. The untimedness and asynchrony of stream-based programming offers a solution to this problem.

  The portability of stream-based programs is evidenced by the fact that programs of considerable complexity and size can be compiled to competitive hardware [13] as well as software [14], which suggests that stream-based programming might even be a solution to the old problem of flexibly co-synthesizing different mixes of hardware/software implementations from a single source.
- **Adaptivity.** The success of a stream programming model will in part depend on its ability to configure dynamically and to virtualize, i.e. to map to collections of computing resources too small for the entire program at once. The transactional execution of actors generates points of *quiescence*, the moments between transactions, when the actor is in a defined and known state that can be safely transferred across computing resources.

7 The MPEG-4 Case Study

One interesting usage of the collection of CAL actors, which constitutes the MPEG RVC tools library, is as a vehicle for video coding experiments. Since it provides a source of relevant application of realistic sizes and complexity, the tools library also enables experiments in dataflow programming, the associated development process and development tools.

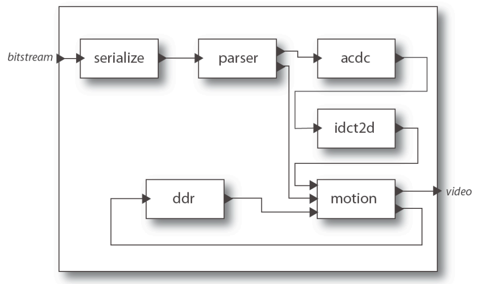

Figure 2. Top-level dataflow graph of the MPEG-4 decoder.

Some of the authors have performed a case study[13], in which the MPEG-4 Simple Profile decoder was specified in CAL and implemented on an FPGA using a CAL-to-RTL code generator. Figure 2 shows a top-level view of decoder. The main functional blocks include a bitstream parser, a reconstruction block, a 2D inverse cosine transform, a frame buffer and a motion compensator. These functional units are themselves hierarchical compositions of actor networks. The objective of the design was to support 30 frames of 1080p in the YUV420 format per second, which amounts to a production of 93.3 Mbyte of video output per second. The given target clock rate of 120 MHz implies 1.29 cycles of processing per output sample on average.

The results of the case study were encouraging in that the code generated from the CAL specification did not only outperformed the handwritten reference in VHDL, both in terms of throughput and silicon area, but also allowed for a significantly reduced development effort. Table 3 shows the comparison between CAL specification and the VHDL reference.

It should be emphasized that this counter-intuitive result cannot be attributed to the sophistication of the synthesis tool. On the contrary the tool does not perform a number of potential optimizations; particularly it does not consider optimizations involving more than one actor. Instead, the good results appear to be due to the development process. A notable difference was that the CAL specification went through significantly more design iterations than the VHDL reference —in spite of being performed in a quarter of the development time. Whereas a dominant part of the development of the VHDL reference was spent getting the system to work correctly, the effort of the CAL specification was focused on optimizing system performance to meet the design constraints.

The initial design cycle resulted in an implementation that was not only inferior to the VHDL reference, but one that also failed to meet the throughput and area constraints. Subsequent iterations explored several other points in the design space until arriving at a solution that satisfied the constraints. At least for the case study, the benefit of short design cycles seem to outweigh the inefficiencies that were induced by high-level synthesis and the reduced control over implementation details.

	Size	**Speed**	**Code size**	**Dev. time**
	slices, BRAM	kMB/S	kLOC	MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12
Improv. factor	1.2	1.6	3.75	4

Figure 3. Hardware synthesis results for an MPEG-4 Simple Profile decoder. The numbers are compared with a reference hand written design in VHDL.

In particular, the asynchrony of the programming model and its realization in hardware allowed for convenient experiments with design ideas. Local changes, involving only one or a few actors, do not break the rest of the system in spite of a significantly modified temporal behavior. In contrast, any design methodology that relies on precise specification of timing —such as RTL, where designers specify behavior cycle-by-cycle— would have resulted in changes that propagate through the design.

Figure 3 shows the quality of result produced by the RTL synthesis engine for a real-world application, in this case an MPEG-4 Simple Profile video decoder. Note that the code generated from the high-level dataflow description actually outperforms the VHDL design in terms of both throughput and silicon area for a FPGA implementation.

8 Summary

We believe that the move towards parallelism in computing and the growth of application areas that lend themselves to dataflow modeling present a huge opportunity for a dataflow programming model that could supplant or at least complement von Neumann computing in many fields.

We have discussed some properties that comes with using a dataflow model, such as explicit parallelism and decoupling of scheduling and communication. The open source simulation and compilation framework OpenDF was presented together with the CAL language and the DIF/TDP analysis tools. Finally, the work on the MPEG-4 decoder verifies the potential of the dataflow approach.

References

[1] J. Thomas-Kerr, J. W. Janneck, M. Mattavelli, I. Burnett, and C. Ritz, “Reconfigurable Media Coding:

Self-describing multimedia bitstreams," in *Proceedings IEEE Workshop on Signal Processing Systems—SiPS 2007*, October 2007, pp. 319–324.

[2] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The Click modular router," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, 1999.

[3] 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.

[4] S. S. Bhattacharyya and W. S. Levine, "Optimization of signal processing software for control system implementation," in *Proceedings of the IEEE Symposium on Computer-Aided Control Systems Design*, Munich, Germany, October 2006, pp. 1562–1567, invited paper.

[5] H. Zima and B. Chapman, *Supercompilers for parallel and vector computers*. New York, NY, USA: ACM, 1991.

[6] M. Hind, "Pointer analysis: haven't we solved this problem yet?" in *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. New York, NY, USA: ACM, 2001, pp. 54–61.

[7] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.

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

[9] S. Ritz, M. Pankert, V. Živojnović, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Intl. Conf. on Application-Specific Array Processors*. Prentice Hall, IEEE Computer Society, 1993, pp. 285–296.

[10] C. Hewitt, "Viewing control structures as patterns of passing messages," *Artif. Intell.*, vol. 8, no. 3, pp. 323–364, 1977.

[11] J. Eker and J. W. Janneck, "Cal language report," University of California at Berkeley, Tech. Rep. UCB/ERL M03/48, December 2003.

[12] C. Lucarz and J. J. Marco Mattavelli, Joseph Thomas-Kerr, "Reconfigurable media coding: A new specification model for multimedia coders," in *Proceedings of IEEE Workshop on Signal Processing Systems*, 2007, pp. 481–486.

[13] J. W. Janneck, I. D. Miller, D. B. Parlour, G. Roquier, M. Wipliez, and M. Raulet, "Synthesizing hardware from dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008.

[14] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour, "Automatic software synthesis of dataflow programs: an MPEG-4 simple profile decoder case study," in *Proceedings of the 2008 IEEE Workshop on Signal Processing Systems (SiPS)*, 2008.

[15] C. von Platen and J. Eker, "Efficient realization of a cal video decoder on a mobile terminal," in *Proceedings of IEEE Workshop on Signal Processing Systems*, 2008.

[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] 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*, Dallas, Texas, September 2005, pp. 37–49.

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

[19] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

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

[21] P. K. Murthy and E. A. Lee, "Multidimensional synchronous dataflow," *IEEE Transactions on Signal Processing*, vol. 50, no. 8, pp. 2064–2079, August 2002.

[22] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, October 2001.