In Proceedings of the Workshop on Resources Aware Sensor and Surveillance Networks in conjunction with the IEEE International Conference on Advanced Video and Signal Based Surveillance, Klagenfurt, Austria, August 2011.

Modeling and Optimization of Dynamic Signal Processing in Resource-Aware Sensor Networks

Shuvra S. Bhattacharyya, William Plishker, Nimish Sane, Chung-Ching Shen, Hsiang-Huang Wu
Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, USA
{ssb, plishker, nsane, ccshen, hhwu}@umd.edu

Abstract

Sensor node processing in resource-aware sensor networks is often critically dependent on dynamic signal processing functionality — i.e., signal processing functionality in which computational structure must be dynamically assessed and adapted based on time-varying environmental conditions, operating constraints or application requirements. In dynamic signal processing systems, it is important to provide flexibility for run-time adaptation of application behavior and execution characteristics, but in the domain of resource-aware sensor networks, such flexibility cannot come with significant costs in terms of power consumption overhead or reduced predictability. In this paper, we review a variety of complementary models of computation that are being developed as part of the dataflow interchange format (DIF) project to facilitate efficient and reliable implementation of dynamic signal processing systems. We demonstrate these methods in the context of resource-aware sensor networks.

1. Introduction

Dynamic signal processing systems, where significant changes in computational structure must be achieved while applications are running, are becoming increasingly important as computational platforms become more powerful, and feature-sets of signal-processing-powered applications become more sophisticated. A major objective in the *dataflow interchange format* (*DIF*) project is the development of efficient methods for design and implementation of signal processing systems from static and dynamic dataflow representations [8]. In this paper, we provide a discussion of different dataflow models of computation that are being developed in DIF project to help address the challenges of structured specification, simulation, and synthesis of dynamic signal processing systems. These models include the following.

- Enable-invoke dataflow (EIDF), which is aimed at improving the predictability of actor invocation and the efficiency with which dynamic scheduling techniques can be realized [20].
- Parameterized synchronous dataflow (PSDF), which allows arbitrary parameters of synchronous dataflow graphs to be adapted at run-time, and allows for construction of streamlined quasi-static schedules based on the framework of parameterized looped schedules [2, 11].
- The dataflow schedule graph (DSG), which provides a formal framework for representing and analyzing dataflow graph schedules, and accommodates a wide range of schedule classes, including static, quasi-static, and dynamic schedules, as well as both sequential and parallel schedule formats [24].

In this paper, we review the EIDF, DSG, and PSDF models of computation, and discuss their potential to improve the quality of embedded software for resource-aware sensor networks.

2. Background on dataflow graphs

Dataflow is widely used as a programming methodology for design and implementation of signal processing systems (e.g., see [3]). In the context of signal processing systems, dataflow provides a formal representation of high level signal flowgraph functionality, which can be leveraged to help verify key implementation properties, and optimize objectives such as performance, memory requirements, and energy consumption. In dataflow programming, an application is represented as a directed graph in which vertices (*actors*) represent computational modules, and edges represent first in first out (FIFO) buffers that connect pairs of communicating actors.

Rather than executing as continuous processes, dataflow actors execute as sequences of discrete computational units, which are referred to as *firings*. Firings can be executed through the control of a static schedule, a run-time process (dynamic scheduler), or a hybrid combination of static and dynamic scheduling subsystems. This concept of discrete firings is sometimes used to distinguish dataflow from the more general *Kahn process network* (*KPN*) model, although in informal discussions, the terms dataflow graph and KPN are often used interchangeably [9, 14].

As actors fire, they consume data values from their input FIFOs, and produce data values onto their output FIFOs. Conceptually, these data values are encapsulated in objects called *tokens*, although when primitive data type values are exchanged, they are typically moved to and from FIFOs in a "raw" fashion, without being embedded in any higher level storage structures.

Of particular interest for design of dynamic signal processing processing system are techniques for *quasi-static scheduling*, where a significant component of scheduling structure is fixed statically, and adaptations on top of the fixed scheduling structure are performed at run-time with relatively low overhead [12]. Quasi-static scheduling provides useful trade-offs among flexibility, efficiency, and predictability, and is increasingly used in model-based design processes for signal processing [3].

3. Enable-invoke dataflow

Dataflow models for signal processing are often characterized in terms of restrictions or assumptions on the rates (in terms of tokens per firing) at which actors produce and consume data from their input and output ports, respectively. Characteristics of these rates, called *token transfer rates*, typically have strong influence on scheduling techniques and other important forms of dataflow graph analysis.

Synchronous dataflow (*SDF*) refers to a restricted form of dataflow in which token transfer rates are constant, and statically known [13]. This form of token transfer predictability promotes the development of powerful techniques for analysis and optimization, and there is a large and growing body of literature on such SDF techniques (e.g., see [13, 8, 5, 16, 7, 25, 3]. However, the restriction to constant token transfer rates significantly limits the expressive power of SDF. This limitation is significant for many sensor network systems — in particular, systems that must rely on dynamic signal processing behavior, as described in Section 1.

Enable-invoke dataflow (EIDF) provides a highly expressive, dynamic dataflow model of computation in which actors are specified in terms of *modes*, which can be viewed intuitively as different "SDF configurations" of actor execution. In other words, individual EIDF modes of a given actor must have fixed token transfer rates, but the rates of different modes for the same actor can vary. Since the current mode can change after each actor firing, dynamic dataflow can be achieved through mode trajectories that have heterogeneous token transfer rates.

Fireability for an EIDF actor can be characterized in terms of a function called the *enabling function*. Given an EIDF actor A with mode set M, $\mathrm{in}(A)$ input ports, and $\mathrm{out}(A)$ output ports, the enabling function ϵ_A for A is a Boolean valued function

$$\epsilon_A : (T_A \times M_A) \to B, \qquad (1)$$

where $T_A = N^{\mathrm{in}(A)}$ represents the set of all possible combinations of buffer populations on the input FIFOs of A (here, $N = \{0, 1, \ldots\}$ is the set of non-negative integers), and $B = \{\mathrm{true}, \mathrm{false}\}$ represents the set of Boolean values. If A has no input ports, then the enabling function for A is uniformly true-valued: $\epsilon_A(m) = \mathrm{true}$ for all m.

Intuitively, $\epsilon_A(x_1, x_2, \ldots, x_{\mathrm{in}(A)}, m)$ is true-valued if and only if in mode m, the constant consumption rate for each ith input port of A is less than or equal to x_i. 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.

When an EIDF actor is executed in a given mode m, it consumes c_i tokens from each ith input port, where c_i is the constant consumption rate that is associated with port i and mode m; produces p_i tokens on each ith output port, where p_i is the constant production rate associated with i and m; and returns a "mode subset" $\nu \subseteq M$, which gives the set of possible *next modes* of the actor (i.e., the set of possible modes in which the next firing of the actor can execute).

Thus, when an EIDF actor fires, it must do so according to a specific (single) mode, and the actor must be enabled with respect to a given mode before it can be fired in that mode. The provision for multiple next mode possibilities ($\mid \nu \mid > 1$) allows for specification of non-deterministic systems where the actual selection of next modes is not part of the specification, but left rather to the implementation (i.e., to a scheduler that selects the actual next mode at any given time from the corresponding set ν of *possible* next modes).

When EIDF is restricted such that there is always exactly one possible next mode ($\nu = 1$), the resulting model of computation is deterministic in the sense that the inputs and initial state (e.g., parameter settings) of a specification uniquely determine the outputs. This deterministic special case of EIDF is called *core functional dataflow* (*CFDF*) [20], and is the basis for a major component of the capability in DIF for modeling dynamic signal processing applications. Although CFDF is significantly restricted compared to EIDF, it is highly expressive as a deterministic dataflow model of computation. Indeed, it has been shown that Boolean dataflow actors [4] can be transformed into functionally equivalent CFDF actors [20]. From such a transformation and the established Turing completeness of Boolean dataflow, it can be demonstrated that CFDF is also Turing complete.

In EIDF and CFDF, enabling and invoking capabilities of an actor must be specified separately. In other words, a function in the actor API (*method*) must be dedicated to implementing the enabling function (returning whether or not the actor is ready for its next firing based on the current state of the input FIFOs and a given mode argument), and a separate method (the *invoking method*) must be dedicated to executing one firing of the actor in a given mode under the assumption that all required input data is available. If the invoking method is called when there is not sufficient input data, the results are in general unpredictable. For each call to the invoking method, the designer or enclosing tool can ensure sufficient data through appropriate static or quasi-static scheduling techniques or by first checking the status of the input buffers using the enabling method. In practice — in particular, when bounded FIFO buffers are employed — it is useful to extend the enabling method to ensure that the output buffers have sufficient empty space (i.e., to accommodate the output data produced by a given actor mode). Such an extension is easy to incorporate when implementing the EIDF model.

In the implementation of dataflow tools, functionality related to the EIDF enabling and invoking methods are often interleaved. Even when they are not interleaved, the separation is conventionally not defined as a first-class citizen of the computational model, which makes it impossible for developers of scheduling techniques to reliably take advantage of it. As a common scenario, for example, an actor firing may have computations that are interleaved with blocking reads of data that provide successive inputs to those computations. In contrast, there is a full separation of enabling and invoking capabilities in EIDF. This separation helps to improve the predictability of an actor invocation (since availability of the required data can be guaranteed in advance by the enable method), and to develop efficient quasi-static scheduling techniques (through the standardized provisions for low-overhead fireability checking). Both of these features — predictability and support for quasi-static scheduling — are useful in addressing the real-time constraints and energy consumption minimization objectives that are characteristic in the domain of resource-aware sensor networks.

This separation also leads naturally to a concept of *guarded execution*. If A is an EIDF actor, then a guarded execution of A at time t has no effect if A is not enabled at time t, and has the effect of firing A once if A is enabled at time t. This concept of guarded execution provides a useful primitive for the design of CFDF scheduling techniques, and their representation in terms of intuitive *schedule tree* structures [19].

A useful application of CFDF modeling and the concept of guarded execution is the technique of *mode grouping*, which provides increased flexibility in identifying *static regions* — i.e., localized subsystems that exhibit constant token transfer rates — in dataflow graphs [19]. Static regions can be exploited in quasi-static scheduling by processing them with streamlined SDF scheduling techniques, and embedding the resulting SDF "region schedules" within the control of a global dynamic, inter-region scheduler [6]. By using CFDF modes as the basic primitive of static region construction, designers and tools generally have available a larger design space of static regions compared to conventional static region techniques, which are restricted to actor-based regions. This enhanced design space can lead to more powerful optimization by quasi-static schedulers.

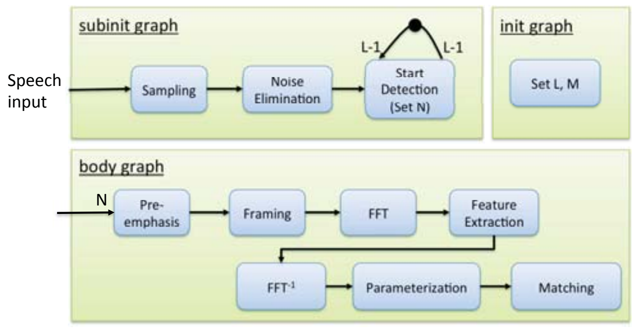

Figure 1. PSDF representation for the embedded speech recognition system of [18].

4. Parameterized synchronous dataflow

Parameterized synchronous dataflow (PSDF) can be viewed as the model of computation that results when the *parameterized dataflow* meta-model [2] is applied to integrate dynamic parameterization features into the synchronous dataflow model.

To illustrate DIF-based signal processing system design methods based on PSDF semantics, Figure 1 shows a PSDF representation for an embedded speech recognition system that was proposed in [18]. In this representation, sensing inputs and start detection schemes are modeled using the *init* and *subinit* graphs of PSDF, and the speech recognition processing algorithm is modeled using a *body* graph. In the PSDF framework, the body graph associated with a subsystem specification is used to represent the core signal processing functionality for the subsystem, while the optional init and subinit graphs are used to specify the functionality associated with any dynamic parameter adaptation in the subsystem (e.g., methods for updating parameter values associated with the body graph).

In the PSDF representation of Figure 1, the number of speech samples L that comprises a speech instance (i.e., a single instance of speech that is to be analyzed for the presence of a word) is set in the init graph. The value of L

relates to the sampling frequency that is employed, and the maximum time duration for a recognizable word. The init graph also determines the frame size M, which gives the FFT block size (“frame size”) prior to any zero padding.

In the subinit graph, voice data is sampled, and a noise elimination scheme can optionally (e.g., if energy availability and performance constraints permit) be implemented to reduce sampling noise. Then a start detection scheme is executed to detect the start of an utterance for a speech word. Once a possible utterance is detected, N consecutive signal samples are captured and stored in the memory of the associated sensor node for further processing. The value of N is set to be equal to the value of L, as determined by the init graph.

The processing steps for speech recognition are then applied to blocks of $N = L$ collected samples. Here, in order to ensure real-time processing, a circular window is maintained associated with the start detection scheme so that the past $L - 1$ samples are kept at each point in time.

In the PSDF body graph, the synchronous dataflow (SDF) model is applied to model the core speech recognition processing functionality for the application [13]. The core functionality includes algorithms for pre-emphasis, framing, FFT, feature extraction, inverse FFT, parameterization, and matching.

By embedding the speech processing functionality of Figure 1 in each node of a wireless sensor network, we have developed a distributed automatic speech recognition (DASR) system. This DASR system is designed to recognize isolated spoken words through a wireless sensor network that is based on a single-cluster, master-slave topology [22]. To optimize energy consumption and maximize overall network lifetime, it is useful to partition the body graph processing for each sensor (slave) node such that a subset of the processing is off-loaded to the master node. Such *energy-driven partitioning* of synchronous dataflow graphs can involve complex design spaces with wide variation in energy efficiency among alternative partitions [22]. Careful selection of a partition is thus important in terms of the system lifetime.

Using this PSDF-based application representation in conjunction with the partitioning scheme described above, we can achieve over 50% improvement in system lifetime. This improvement is achieved by modeling the energy consumption of sensor data processing and transceiver operation, and using these models in conjunction with the high level dataflow representation of Figure 1 to distribute the functionality across the sensor network for optimized energy consumption [22].

A variety of methods have been developed for modeling and analysis using PSDF and related modeling techniques. An exploration of PSDF and more general parameterized dataflow techniques for modeling of software defined radio systems is explored in [1] . Application of PSDF techniques to field programmable gate array implementation of the physical layer for 3GPP-Long Term Evolution (LTE) is investigated in [10]. The integration of concepts related to parameterized dataflow in language extensions for embedded streaming systems is explored in [15]. Techniques for analysis and verification of hierarchically reconfigurable dataflow graphs are explored in [17].

5. The Dataflow Schedule Graph

As described earlier, scheduling is an important task in the process of mapping a dataflow graph into an implementation. Scheduling generally involves the process of assigning actors to processors and ordering actor subsets that share common processors. By a “processor” in this context, we mean a hardware resource on which execution is time-multiplexed by actors that are assigned to it. Scheduling has a major effect on key implementation objectives, and many scheduling techniques are geared toward optimizing specific subsets of these objectives. Commonly-targeted objectives of optimized schedulers include exploiting parallelism, streamlining buffer management, and minimizing energy or power consumption. For discussions of various scheduling techniques in the context of signal processing system design, we refer the reader to [3].

Motivated by the fundamental role of scheduling in dataflow-based design, the *dataflow schedule graph* (*DSG*) representation has been developed as formal model for expressing a broad class of schedule structures, including static, dynamic, and quasi-static schedules, as well as both sequential and parallel schedules. Key objectives of the DSG representation include the following.

- To provide a common representation for representing, analyzing, and manipulating dataflow graph schedules.
- Flexible support for alternative application-level dataflow models, such as BDF, CFDF, PSDF, SDF, and many other forms of dataflow.
- Interoperability with other, more specialized schedule models.
- Formulation of scheduling logic in terms of dataflow based principles; that is, DSGs provide representations of dataflow graph schedules in terms of dataflow semantics.

The DSG representation, as introduced in [24], is formulated in terms of CFDF semantics, which is highly expressive and subsumes many other useful dataflow models as special cases [21]. However, the DSG representation can be extended or adapted to work with other modeling techniques; see [24] for further details.

DSG representations are intended to complement corresponding dataflow-based application models by providing formal representations of schedules for those models. More precisely, given a dataflow-based application model G and a given target architecture (collection of processors) A, a DSG $D(G)$ for G is another dataflow graph, which represents a schedule for G on A. The DSG $D(G)$ does not explicitly model data communication between actors, as this is already modeled in G. Instead, tokens that flow along edges of $D(G)$ serve to enable actors for execution. Such tokens can be viewed intuitively as actor-level "program counters" for the associated processors in A.

To enforce the assumption that processors execute only one actor at a time, the DSG model is formulated to ensure that the DSG subgraph $s_p(D)$ corresponding to each individual processor p in A contains at most one token at any given time. Formally, $s_p(D)$ is formed from the subset Z_p of actors in $D(G)$ that is associated with scheduling logic and actor execution on p, together with all of the edges in $D(G)$ whose source and sink vertices are in Z_p.

Intuitively, DSGs include three kinds of actors:

- *Reference actors* (*RAs*). An RA in $D(G)$ can be viewed as an execution (firing) wrapper for a specific application graph actor (called the *referenced actor* of the RA) in G. Intuitively, an RA corresponds to a CFDF guarded execution of its referenced actor. RAs themselves obey *homogeneous synchronous dataflow* (*HSDF*) semantics, which means that the token transfer rates on their ports are uniformly equal to 1.

- *Schedule control actors* (*SCAs*). SCAs provide for modeling of dynamic scheduling structures in DSGs. Unlike previous kinds of dynamic scheduling structures, SCAs are not restricted to pre-defined schemas, and provide a flexible and general framework for building dynamic scheduling constructs. SCAs adhere to *lumped HSDF* semantics, which means that on each firing the total number of tokens consumed across all input ports equals 1, and similarly, the number of tokens consumed across all output ports equals 1.

- *Interprocessor communication actors* (*ICAs*). ICAs are used to construct *concurrent DSGs*, which model parallel schedules. ICAs embed functionality for sending and receiving data across distinct processors, and performing associated synchronization functions. These actors are similar to the kinds of inter-processor communication actors used in the synchronization graph and interprocessor communication graph models (e.g., see [23]), but are not restricted to the context of HSDF schedules, as they are in those earlier models.

As a simple illustration of the capabilities in DSGs to model quasi-static schedules, Figure 2 illustrates a DSG

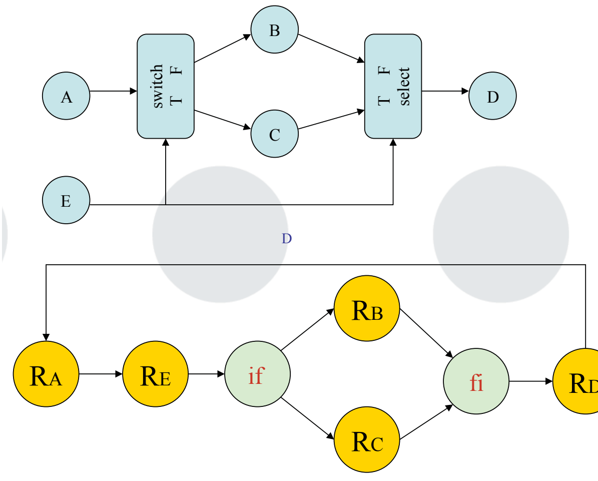

Figure 2. A dataflow schedule graph representation of a quasi-static schedule for a Boolean dataflow application graph.

representation (bottom of the figure) for a BDF application graph (top of the figure). The actors labeled `switch` and `select` in the application graph are fundamental dynamic dataflow actors in BDF; each actor labeled R_X in the DSG is a reference actor with the associated referenced actor being the actor X in the BDF graph; and the other two DSG actors are schedule control actors. For further details on this example, we refer the reader to [24].

The DSG representation is useful in the design and implementation of resource-aware sensor networks as it provides a common, formal framework through which a wide variety of scheduling techniques can be modeled. This makes the representation suitable as an intermediate representation for design tools, and also as a basis for software architectures that support designer-constructed schedules. For more details on the DSG representation, we refer the reader to [24].

6. Summary

In this paper, we have motivated the problem of dynamic signal processing in resource-aware sensor networks, and presented a set of complementary models of computation that are being developed in the dataflow interchange format (DIF) project to help address this problem. Specifically, we have covered the enable-invoke dataflow model for modeling of dynamic dataflow applications; parameterized synchronous dataflow model for modeling of reconfigurable dataflow behavior; and dataflow schedule graph for modeling of static, dynamic, and quasi-static schedule structures. We have also drawn connections to the large body of related work from which these methods have evolved, and have been motivated. The EIDF, PSDF and DSG models help to provide structured, formally-rooted methods for addressing the challenges of implementing dynamic signal processing reliably and efficiently in resource-aware sensor networks, and other important forms of signal processing systems.

7. Acknowledgments

This work was sponsored in part by the Laboratory for Physical Sciences, Laboratory for Telecommunication Sciences, U.S. Air Force Research Laboratory, and U.S. National Science Foundation.

References

[1] H. Berg, C. Brunelli, and U. Lucking. Analyzing models of computation for software defined radio applications. In *Proceedings of the International Symposium on System-on-Chip*, 2008.

[2] B. Bhattacharya and S. S. Bhattacharyya. Parameterized dataflow modeling for DSP systems. *IEEE Transactions on Signal Processing*, 49(10):2408–2421, October 2001.

[3] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, 2010.

[4] J. T. Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control systems. In *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, pages 508–513, October 1994.

[5] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, A. J. M. Moonen, M. J. G. Bekooij, B. D. Theelen, and M. R. Mousavi. Throughput analysis of synchronous data flow graphs. In *Proceedings of the International Conference on Application of Concurrency to System Design*, June 2006.

[6] R. Gu, J. Janneck, M. Raulet, and S. S. Bhattacharyya. Exploiting statically schedulable regions in dataflow programs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 565–568, Taipei, Taiwan, April 2009.

[7] N. Guan, Z. Gu, W. Yi, and G. Yu. Improving scalability of model-checking for minimizing buffer requirements of synchronous dataflow graphs. In *Proceedings of the Asia South Pacific Design Automation Conference*, pages 715–720, January 2009.

[8] 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, September 2005.

[9] G. Kahn. The semantics of a simple language for parallel programming. In *Proceedings of the IFIP Congress*, 1974.

[10] H. Kee, I. Wong, Y. Rao, and S. S. Bhattacharyya. FPGA-based design and implementation of the 3GPP-LTE physical layer using parameterized synchronous dataflow techniques. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 1510–1513, Dallas, Texas, March 2010.

[11] 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.

[12] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real time DSP. In *Proceedings of the Global Telecommunications Conference*, November 1989.

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

[14] E. A. Lee and T. M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–799, May 1995.

[15] Y. Lin, Y. Choi, S. Mahlke, T. Mudge, and C. Chakrabarti. A parameterized dataflow language extension for embedded streaming systems. In *Proceedings of the International Symposium on Systems, Architectures, Modeling and Simulation*, pages 10–17, July 2008.

[16] R. Lublinerman and S. Tripakis. Translating data flow to synchronous block diagrams. In *Proceedings of the IEEE Workshop on Embedded Systems for Real-Time Multimedia*, 2008.

[17] S. Neuendorffer and E. Lee. Hierarchical reconfiguration of dataflow models. In *Proceedings of the International Conference on Formal Methods and Models for Codesign*, June 2004.

[18] S. Phadke, R. Limaye, S. Verma, and K. Subramanian. On design and implementation of an embedded automatic speech recognition system. In *Proceedings of the International Conference on VLSI Design*, pages 27–132, 2004.

[19] W. Plishker, N. Sane, and S. S. Bhattacharyya. Mode grouping for more effective generalized scheduling of dynamic dataflow applications. In *Proceedings of the Design Automation Conference*, pages 923–926, San Francisco, July 2009.

[20] 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*, pages 17–23, Monterey, California, June 2008.

[21] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. In *Proceedings of the International Workshop on Systems, Architectures, Modeling, and Simulation*, pages 157–166, Samos, Greece, July 2008.

[22] C. Shen, W. L. Plishker, D. Ko, S. S. Bhattacharyya, and N. Goldsman. Energy-driven distribution of signal processing applications across wireless sensor networks. *ACM Transactions on Sensor Networks*, 6(3), June 2010. Article No. 24, 32 pages.

[23] S. Sriram and S. S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, second edition, 2009.

[24] H. Wu, C. Shen, N. Sane, W. Plishker, and S. S. Bhattacharyya. A model-based schedule representation for heterogeneous mapping of dataflow graphs. In *Proceedings of the International Heterogeneity in Computing Workshop*, pages 66–77, Anchorage, Alaska, May 2011.

[25] J. Zhu, I. Sander, and A. Jantsch. Buffer minimization of real-time streaming applications scheduling on hybrid CPU/FPGA architectures. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2009.