In Proceedings of the IEEE International Conference
on Multimedia and Expo, Barcelona, Spain, July 2011.

A DESIGN TOOL FOR EFFICIENT MAPPING OF MULTIMEDIA APPLICATIONS ONTO HETEROGENEOUS PLATFORMS

Chung-Ching Shen, Hsiang-Huang Wu, Nimish Sane, William Plishker, and Shuvra S. Bhattacharyya

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

ABSTRACT

Development of multimedia systems on heterogeneous platforms is a challenging task with existing design tools due to a lack of rigorous integration between high level abstract modeling, and low level synthesis and analysis. In this paper, we present a new dataflow-based design tool, called the *targeted dataflow interchange format* (*TDIF*), for design, analysis, and implementation of embedded software for multimedia systems. Our approach provides novel capabilities, based on the principles of task-level dataflow analysis, for exploring and optimizing interactions across application behavior; operational context; heterogeneous platforms, including high performance embedded processing architectures; and implementation constraints.

Index Terms— Embedded signal processing, software synthesis, design tools, dataflow graphs.

1. INTRODUCTION

Nowadays, a variety of design platforms, such as Texas Instruments' Multimedia Video Processor, Broadcom's Mobile Multimedia Processors, or Nvidia's GoForce Multimedia Processor, are available for implementing a wide range of multimedia applications. However, the application and exploitation of these heterogeneous platforms for multimedia system design remains largely ad hoc, and the retargetability of design tools across these platforms has not been adequately addressed, resulting in a lack of rigorous integration between high level modeling, and low level synthesis and analysis.

Multimedia applications can often be described in terms of signal processing block diagrams. Model-based design methods based on *dataflow* models of computation have become increasingly popular to provide formal semantics for such block diagrams because of their natural correspondence to signal flow graphs and system level DSP flows. Consequently, dataflow graphs are widely used to model applications in many multimedia domains (e.g., see [1]).

This research was sponsored in part by the US Air Force Research Laboratory, and the Laboratory for Telecommunication Sciences.

In dataflow models of computation, DSP applications are modeled as directed graphs, where vertices (*actors*) represent computational modules for executing (*firing*) functional tasks, and edges represent first-in-first-out (FIFO) channels for storing data values (*tokens*), and imposing data dependencies between actors. Whenever an actor fires, it produces and consumes tokens from its input and output edges, respectively.

There is a wide variety of development tools that utilize dataflow models to aid in the design and implementation of DSP applications (e.g., see [2, 3, 4, 5, 6, 7, 8]). Using these tools, application designers can develop the functionality of dataflow actors, and capabilities are provided for automated system simulation or synthesis. However, static dataflow models are largely used in such tools, which limits their utility in modern multimedia applications, where dynamic dataflow communication across functional subsystems is increasingly employed (e.g., variable data rates arising due to dynamically determined quality of service constraints). Furthermore, existing dataflow tools are largely platform or language specific, and do not address retargetability as a primary objective.

In this paper, we present a new dataflow-based design tool, called the *targeted dataflow interchange format* (*TDIF*), for design and analysis of embedded software for multimedia systems. TDIF extends the capabilities of DIF [5] with dynamic dataflow software synthesis, cross-platform actor design support, and dataflow-integrated features for instrumenting and tuning implementations. The dataflow-based approach used in this work is unique by leveraging the power of dynamic dataflow models and providing integration of automation of code generation for programming interfaces and low level customizations for implementations targeted to heterogeneous platforms.

TDIF provides a flexible environment without compromising the types of optimizations possible by offering a breadth of formal models for the application designer to choose from. This application description is then tied as closely as possible to the application domain, not the target, making it highly portable while still structured enough to be optimized for. Furthermore, individual actors can still

be tuned using low level techniques for the target platforms, and such tuning is facilitated by novel support that is provided for instrumenting dataflow representations and schedules.

2. BACKGROUND

2.1. Core Functional Dataflow

TDIF is based on a general dataflow model of computation called *core functional dataflow* (*CFDF*), which can be viewed as the deterministic sub-class of *enable-invoke dataflow* [9]. CFDF is a dynamic dataflow model that can express both static and data-dependent dataflow rates, as well as conditional behaviors. In CFDF, actors are specified as sets of *modes*, where each mode has a fixed production and consumption rate associated with each output and input port, respectively. During execution, each actor selects one mode from its set of modes as the *current mode*, which can be maintained as part of its state.

In CFDF, the separation of enable (firability checking) and invoke (firing) functionalities is defined as a first class characteristic of the model. Each actor has an associated *enable* function, which can be called at any time between firings, and returns a Boolean value indicating whether or not there is sufficient data available on the actor input ports to fire the actor in its current mode. Since such an isolated enable check is available, the *invoke* function of an actor assumes that sufficient data is present, and reads its input data without blocking reads. When an actor is invoked, it executes its current mode, produces and consumes data, and updates its current mode. Since different modes of an actor can have different production and consumption rates, dynamic dataflow can be modeled flexibly in CFDF.

2.2. The Dataflow Interchange Format

The Dataflow Interchange Format (DIF) framework provides a standard approach for specifying mixed-grain dataflow-based semantics for signal processing system design [5]. The DIF Language (TDL), which is part of the DIF framework, provides a unified textual language for expressing different kinds of dataflow semantics, including graph topologies, hierarchical design structure, dataflow-related design properties, and actor-specific information. TDL is therefore suitable for both programming and interchange (transfer of dataflow graphs across design tools). By using TDL, multimedia signal processing systems can be represented as dataflow graphs at a high level of abstraction.

The DIF package (TDP) is a software tool that accompanies TDL, and provides a variety of intermediate representations, analysis techniques, and graph transformations that are useful for working with dataflow graphs. With the support of module libraries for the actors referenced in a dataflow graph, an efficient software implementation for the graph can be synthesized automatically using the DIF-to-C tool [5]. Although DIF-to-C supports only static dataflow applications — in particular, those that are based on synchronous dataflow (SDF) semantics [10] — the tool is capable of exploring implementation trade-offs that are exposed effectively through DIF-based dataflow representations.

3. THE TARGETED DIF DESIGN TOOL

In this paper, we build on the capabilities of the DIF framework, and develop a new multimedia application development tool called Targeted DIF (TDIF). TDIF consists of new plug-ins to the DIF environment that focus on efficient mapping of CFDF-based multimedia application representations onto embedded platforms. By building on the CFDF model of computation, TDIF can flexibly accommodate both static and dynamically structured multimedia applications.

The TDIF environment currently supports C- and GPU-based implementations (i.e., for CPU and GPU platforms). The GPU-based capabilities of TDIF are currently oriented towards NVIDIA GPUs, based on the CUDA programming framework [11]. Since CUDA is a C-like programming language (CUDA can be viewed a variant of C with NVIDIA extensions and certain restrictions), a C- or CUDA-based actor can be implemented as an abstract data type (ADT) to enable efficient and convenient reuse of the actor across arbitrary applications. In typical C implementations, ADT components include header files to represent definitions that are exported to application developers and implementation files that contain implementation-specific definitions.

An illustration of the TDIF environment and associated design flow is shown in Fig. 1. By following the presented methodology, the designer can focus on design, implementation and optimization for dataflow actors and experiment with alternative scheduling strategies and instrumentation techniques for the targeted platforms based on programming interfaces that are automatically generated from the TDIF tool. These automatically-generated interfaces provide structured design templates for the designer to follow in order to generate dataflow-based actors that are formally integrated into the overall synthesis tool. In Fig. 1, the dashed line indicates design considerations that need to be taken into account jointly to achieve maximum benefit from TDIF-based system design.

The TDIF tool is based on four software packages — the *TDIF compiler*, *TDIFSyn* software synthesis package, *TDIF run-time library*, and *Software synthesis engine*. The interactions among these packages are illustrated in Fig. 1.

3.1. Application Programming Interfaces

As part of the TDIF environment, a new dataflow actor design language, called the *TDIF language*, is provided. The TDIF language gives a high level specification format for writing *dataflow actors* that can be efficiently and reliably retargeted across different platforms. In contrast, TDL in

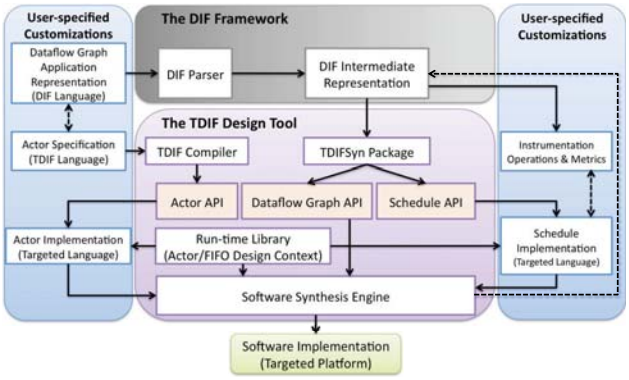


Fig. 1. TDIF-based design flow.

the DIF framework is used to describe high level specifications of *dataflow graphs* for the target application. The TDIF language is a light-weight language that consists of five keywords: `module`, `input`, `output`, `param`, and `mode`.

The keyword `module` is used to define an actor with a given name and type, where the type specifies the language used to implement the actor. The keywords `input` and `output` are used to define input and output ports of an actor along with the names and token types associated with the ports. The keyword `param` is used to define parameters of an actor with names and the associated parameter types. The keyword `mode` is used to define the modes of a CFDF actor.

In the TDIF language, a given actor specification should contain (at the beginning) a single `module` statement; each of the other kinds of statements can be repeated as many times as needed for the given type of structure being declared (e.g., two `input` statements and one `output` statement for a two-input, single-output actor). As discussed previously, C and CUDA are presently supported as target languages. As additional target languages are added to TDIF, the TDIF language will be extended by simply adding additional type options to the `module` statement.

The TDIF compiler, which is developed based on the Bison compiler construction framework [12], parses the TDIF specification of an actor and generates corresponding application programming interfaces (APIs) for CFDF-based, dataflow implementation of the actor in the targeted language. For C and CUDA, these APIs are generated in the form of header files for the actor programmer to base his or her implementations on. The APIs provide standard prototypes for interface functions, including the `invoke` function, which implements the functionality of the actor, and two data rate functions that return the production rate and consumption rate, respectively, associated with a given port and a given mode. The generated API features also include relevant constant definitions associated with the dataflow actor, including the numbers of input ports, output ports, modes, and parameters.

The TDIFSyn package is a software package that takes a DIF intermediate representation as input from the DIF framework (e.g., a representation that has been constructed from a TDL file), and generates a top-level C language implementation file and an associated API for scheduling. Here, by scheduling, we mean the assignment of dataflow actors to processors and the execution ordering of actors that share the same processor. Extensive prior work exists on scheduling dataflow graphs for various purposes (e.g., see [1]). However, systematic techniques are lacking for transferring the results of scheduling techniques into practical implementations. TDIFSyn helps to bridge this gap by providing target-language-specific APIs through which scheduling results can interact with the dataflow graph and its individual components.

The automatically generated top-level C file initializes the *operational contexts* of actors and FIFOs (communication channels between actors), which will be described further in Section 3.2; configures actor parameters; lays out the graph topology by instantiating connections between actor ports and their incident FIFOs; and calls a user-defined scheduler that is implemented based on the generated scheduling API.

3.2. Operational Contexts

In the TDIF environment, relevant state information of actors is encapsulated by instances of a retargetable data structure that is called the *operational context*. More specifically, the operational context of an actor contains the *execution context* (*EC*), which encapsulates actor parameters and state variables, and the *topological context* (*TC*) or *dataflow context*, which encapsulates the set of incident ports, thereby defining how the actor is connected as part of the enclosing dataflow topology. Both the EC and TC are integrated within the run-time library of the TDIF environment. Note that the presence of actor state can be modeled in dataflow graphs through a self-loop edge (an edge whose source and sink are connected to the same actor), and the use of state can make actor programming more convenient and scalable compared to purely functional actor programming (e.g., see [13, 2]). Thus, state-based actor programming is supported in TDIF.

Like actors in TDIF, each FIFO is also equipped with an associated operational context. The FIFO operational context includes information about the data type (token type) associated with the FIFO. For a given FIFO instance, there is a fixed token size (number of bytes per token). Tokens can have arbitrary types — e.g., they can be integers, floating point values (`float` or `double`), characters, or pointers (to any kind of data). The FIFO operational context (FOC) is not "aware" of its associated data type, only of the fixed token size. This organization allows for flexibility in storing different kinds of data values, and efficiency in storing and accessing the data values. A number of FOC utility functions are provided to query FIFO status, including the capacity of the FIFO, number of tokens that are currently in the FIFO, and associated token size (fixed number of bytes per token).

4. INSTRUMENTED SCHEDULE TREES

In the implementation of dataflow graphs, scheduling plays an important role. Scheduling and more broadly, the interactions among scheduling, inter-actor communication, and actor execution, typically have major impact on key metrics, including performance and memory usage [1]. Through the use of the instrumentation methodology provided in the TDIF environment, designers can experiment with and tune different scheduling techniques in order to assess their trade-offs, and steer implementations towards effective solutions.

Our approach to instrumentation in TDIF is designed to support the following key requirements: (a) no change in functionality (instrumentation directives should not change application functionality); (b) operations for adding and removing instrumentation points should be performed by designers in a way that is external to actors (i.e., does not interfere with or require modification of actor code); and (c) instrumentation operations should be *modular* so that they can be mixed, matched, and migrated with ease and flexibility.

Instrumentation support in TDIF builds on the *generalized schedule tree* (GST) representation, which provides a standard graphical format for representing a broad class of dataflow graph schedules [14]. In a GST, each leaf node refers to an actor invocation, and each internal node n represents an expression that is interpreted as an iteration count I_n for the associated sub-tree (i.e., execution of the sub-tree rooted at n is repeated I_n times).

In its *schedule tuning mode*, TDIF allows designers to augment the GST representation with functional modules, encapsulated as *instrumentation nodes*, which are dedicated to instrumentation tasks. Like iteration nodes, instrumentation nodes are incorporated as internal nodes. We refer to GSTs that are augmented with instrumentation nodes as *instrumented GSTs* (*IGSTs*). The instrumentation tasks associated with an instrumentation node are in general applied to the corresponding IGST sub-tree.

An IGST allows software synthesis for a schedule together with instrumentation functionality that is integrated in a precise and flexible format throughout the schedule. Upon execution, software that is synthesized from an IGST produces profiling data (e.g., related to memory usage, performance or power consumption) along with the output data that is generated by the source application.

An instrumentation node in general has two associated functions, pre and $post$, which represent instrumentation-related computations (e.g., system calls, accesses to specialized memory locations, etc.) that are to be carried out just before and after the associated IGST sub-tree executes.

Depending on the desired instrumentation functionality, one or both of the functions pre and $post$ can be used. If both are used (e.g., for performance measurement), such an instrumentation node can be viewed as providing *interval instrumentation*, whereas if only one is used (e.g., to record memory usage), it can be viewed as *point instrumentation*.

Instrumentation nodes therefore provide a formal, dataflow-integrated approach for specifying instrumentation functionality in a manner that flexibly interacts with but is cleanly separated from the code (schedule and actor code) that it interacts with. Such orthogonalization across scheduling, actor, and instrumentation functionality is a key strength of TDIF, which adds to the modularity and productivity features offered by the environment.

5. EXPERIMENTS

In this section, we present experiments with two application examples that are developed using the TDIF environment. Both applications are modeled as dynamic CFDF dataflow graphs, where each actor has at least one *process* mode for performing its main processing task. In each application, the main processing pipeline can be statically scheduled. However, conditional dataflow sub-tasks in each application (e.g., for loading different coefficients, and handling end-of-file behavior) generally prevent the use of a global static schedule. The applications are experimented with on a 3GHz PC with an Intel CPU, 4GB RAM, and an NVIDIA GTX260 GPU.

The first application is a simple image processing application centered around Gaussian filtering. Two dimensional Gaussian filtering is a kernel in image processing that is common as a preprocessing step, and can be used to denoise an image or prepare for multiresolution processing. The Gaussian filter is convolved with the input image by centering a matrix on each pixel and multiplying the value of each entry in the matrix with the appropriate pixel and then summing the results to produce the value of each new pixel. This operation is repeated until the entire image has been created.

Fig. 2(a) shows the CFDF dataflow graph for the Gaussian filtering application. The application reads bitmap files in tile chunks, inverts the values of the pixels of each tile, runs Gaussian filtering on the inverted tile, and then writes the result to an output bitmap file. Since in this design, the tiles vary somewhat between edges, Gaussian filtering applied to tiles must consider a limited neighborhood around each tile (called a *halo*) for correct results. Thus, tiles produced by `bmp_file_reader` overlap, while the halo is discarded after Gaussian filtering, and non-overlapping tiles are the input to `bmp_file_writer`.

The experiment settings are illustrated in Fig. 2. In the experiment, five matrices of Gaussian filter coefficients are stored by the `gaussian_filter` actor in its `init` mode to allow for different standard deviations. Users can configure which matrix will be used as well as the halo value for evaluating filtering effects at run time. The `invert` and `gaussian_filter` actors are implemented in both C and CUDA, and the `bmp_file_reader` and `bmp_file_writer` actors are implemented in C.

We apply IGSTs, which are based on the *CFDF canoni-*

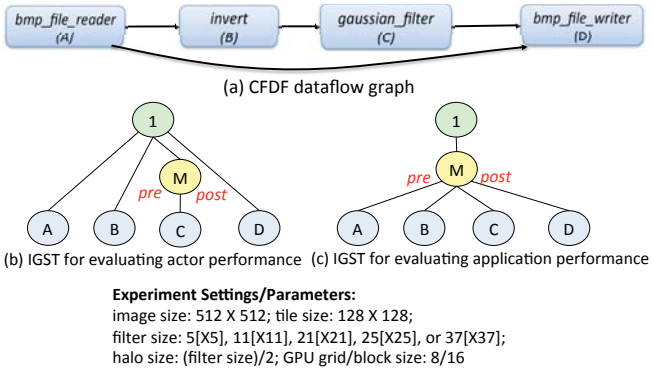


Fig. 2. Gaussian filtering application.

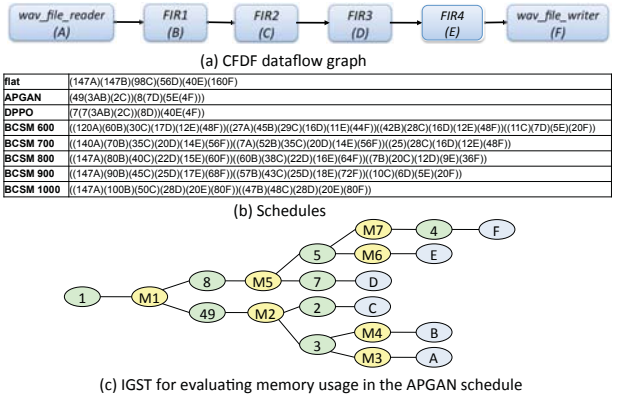


Fig. 3. 44.1 kHz to 48 kHz sampling rate conversion.

cal schedule [9], a standard type of schedule for CFDF graphs that can be constructed quickly and is suitable for rapid prototyping and bottleneck identification. Using the canonical schedule along with selected instrumentation operations, we derive two IGST variants, as shown in Fig. 2(b) and Fig. 2(c), to evaluate `gaussian_filter` actor and application performance, respectively. In these figures, M represents an instrumentation node used for interval instrumentation. The measurement results are reported in Table 1.

Table 1. Performance of the `gaussian_filter` actor (GF) and the Gaussian filtering application (App) for C and CUDA implementations.

Filter size	5X5	11X11	21X21	25X25	37X37
GF. C (ms)	50	280	1080	1540	3310
GF. CUDA (ms)	4.228	4.874	10.257	12.759	21.72
GF. **Speedup**	**11.83**	**57.45**	**105.29**	**120.70**	**152.39**
App. C (ms)	70	295	1100	1550	3340
App. CUDA (ms)	70	80	140	115	130
App. **Speedup**	**1**	**3.69**	**7.86**	**13.48**	**25.69**

As shown in Table 1, the CUDA implementations have superior performance compared to the corresponding C implementations for these experiments. However, the application-level speedups, while still significant, are consistently less than the corresponding actor-level speedups. This is due to factors such as context switch overhead and communication cost for memory movement, which are associated with overall schedule coordination in the application implementations.

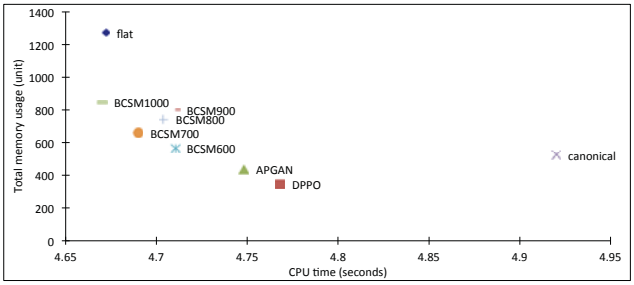
The second application used in our experiments is an audio processing application for 44.1 kHz to 48 kHz sampling rate conversion (e.g., between compact disc and digital audio tape formats). Fig. 3(a) shows the CFDF dataflow graph used for this application. The application reads wav files that contain data sampled at 44.1 kHz, runs a series of multirate filters of finite impulse response to convert the sample rate, and then writes the result to an output wav file that contains 48 kHz data. This application is implemented in C.

In the experiment, we apply IGSTs for memory instrumentation to evaluate schedules derived using a variety of scheduling techniques — in particular, the techniques of *flat scheduling (flat)*, *acyclic pairwise grouping of adjacent nodes (APGAN)*, *dynamic programming post optimization (DPPO)*, and *canonical scheduling (canonical)*. Details on the first three scheduling techniques can be found in [15] and the canonical scheduling is described in [9]. The APGAN and DPPO scheduling techniques, which are handle the main processing pipeline of this CD-DAT application, can be applied statically, while the high level conditional behavior of the application is processed before and after the main pipeline.

In addition, we apply a simple but effective *buffer-constrained, context-switch minimization* (*BCSM*) heuristic that operates as follows. Given a constraint on total available buffer size (e.g., 600 memory units for all FIFOs), and a distribution of the available buffer size equally across all of the application graph FIFOs (e.g. $600/5 = 120$ units per FIFO), a greedy approach is applied to minimize the rate of inter-actor context switching subject to the available buffer capacities — in particular, we start with the source actor, and execute each actor as many times as possible (subject to available input data and the output buffer size) before moving on to the next actor, and repeat the process until all actors have been scheduled the corresponding numbers of times dictated by the SDF repetitions vector [10].

Fig. 3(b) shows all of the derived schedules that we experimented with for the sample rate conversion application, including the results of BCSM with total buffer size constraints of 600, 700, 800, 900, and 1000. These schedules are expressed in terms of *looped schedule* notation [15], where each parenthesized term represents a loop whose iteration count is given by the first (integer) sub-term in the term, and whose loop body is given by the remaining sub-terms. For example, the looped schedule $(3A(2B))$ contains two nested parenthesized terms ("nested schedule loops"), and corresponds to the execution sequence $ABBABBABB$.

Due to space limitations, we only show the IGST for evaluating memory usage in the APGAN schedule (Fig. 3(c)).



	flat	DPPO	APGAN	canonical	BCSM 600	BCSM 700	BCSM 800	BCSM 900	BCSM 1000
CPU time	4.672	4.768	4.748	4.92	4.71	4.69	4.704	4.71	4.67
Total memory usage	1273	347	438	526	565	662	742	798	847
Total context switch	16206	1161430	1164131	2384400	64824	48618	48618	48618	32412

Fig. 4. Instrumentation results for the CD-DAT application.

Here, nodes *M1* through *M7* represent point instrumentation operations that keep track of the maximum population of each buffer that is being monitored.

Fig. 4 reports the instrumentation results for total memory usage versus CPU time for the different schedules implemented for the CD-DAT application. From the results, we can derive Pareto points that provide information based on which the designer can decide which scheduling strategy should be chosen for a desired performance/memory-cost trade-off. The ability to derive such Pareto points using a systematic, retargetable methodology based on high level dataflow representations is a valuable feature provided by the TDIF environment.

6. CONCLUSION

In this paper, we have introduced the *Targeted DIF* (*TDIF*) environment as a novel software tool for design and implementation of multimedia signal processing systems. TDIF is based on high level dataflow graphs, and provides a unique integration of dynamic dataflow modeling; retargetable actor construction; software synthesis; and instrumentation-based schedule evaluation and tuning. We have presented two application case studies to demonstrate the utility of the TDIF environment. Useful directions for future work include the development of hardware description language extensions to the TDIF language and synthesis engine.

7. REFERENCES

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

[2] J. Eker and J. W. Janneck, "CAL language report, language version 1.0 — document edition 1," Tech. Rep. UCB/ERL M03/48, Electronics Research Laboratory, University of California at Berkeley, December 2003.

[3] G. W. Johnson, *LabVIEW Graphical Programming: Practical Applications in Instrumentation and Control*, McGraw-Hill, 1997.

[4] J. L. Pino and K. Kalbasi, "Cosimulating synchronous DSP applications with analog RF circuits," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[5] 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*, September 2005, pp. 37–49.

[6] 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.

[7] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *Proceedings of the International Conference on Compiler Construction*, 2002.

[8] L. F. Teixeira, L. G. Martins, M. Lagrange, and G. Tzanetakis, "MarsyasX: multimedia dataflow processing with implicit patching," in *Proceedings of the ACM International Conference on Multimedia*, 2008.

[9] 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*, June 2008, pp. 17–23.

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

[11] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[12] C. Donnelly and R. Stallman, *Bison – The Yacc-compatible Parser Generator*, August 2010.

[13] E. A. Lee, "Recurrences, iteration, and conditionals in statically scheduled block diagram languages," in *Proceedings of the International Workshop on VLSI Signal Processing*, 1988.

[14] 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.

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