Design and Synthesis for Multimedia Systems using the Targeted Dataflow Interchange Format

Chung-Ching Shen, *Member, IEEE,* Shenpei Wu, Nimish Sane, *Member, IEEE,*
Hsiang-Huang Wu, William Plishker, *Member, IEEE,*
and Shuvra S. Bhattacharyya, *Fellow, IEEE*

IEEE Transactions on Multimedia, 14(3):630-640, June 2012

Abstract

Development of multimedia systems that can be targeted to different platforms is challenging due to the need for rigorous integration between high level abstract modeling, and low level synthesis and optimization. In this paper, a new dataflow-based design tool called the targeted dataflow interchange format (TDIF) is introduced for retargetable design, analysis, and implementation of embedded software for multimedia systems.

Our approach provides novel capabilities, based on principles of task-level dataflow analysis, for exploring and optimizing interactions across design components; object-oriented data structures for encapsulating contextual information for components; a novel model for representing parameterized schedules that are derived from repetitive graph structures; and automated code generation for programming interfaces and low level customizations that are geared toward high performance embedded processing architectures. We demonstrate our design tool for cross-platform application design, parameterized schedule representation, and associated dataflow graph code generation using a case study centered around an image registration application.

C. Shen, H. Wu, W. Plishker, and S. S. Bhattacharyya are with the Department of Electrical and Computer Engineering, University of Maryland, College Park, MD 20742, USA (e-mail: ccshen@umd.edu; hhwu@umd.edu; plishker@umd.edu; ssb@umd.edu).

S. Wu is with SAIC, Rockville, MD 20852, USA. (e-mail: shenpei.wu@gmail.com).

N. Sane is with New Jersey Institute of Technology, Newark, NJ 07102, USA (e-mail: nimish.sane@njit.edu).

Index Terms

Embedded signal processing, software synthesis, design tools, scheduling, dataflow graphs.

I. Introduction

A variety of design platforms, such as the Texas Instruments Multimedia Video Processor [1], Broadcom Mobile Multimedia Processors [2], and NVIDIA Tegra Multimedia Processor [3], 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 [4]). Furthermore, the behavior of many multimedia applications can be characterized by patterns of stream processing computations and modeled efficiently using dataflow.

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 consumes and produces tokens from and to its input and output edges, respectively.

When implementing a dataflow-based multimedia application model on a target platform, scheduling plays an important role (e.g., see [4]). Here, by *scheduling*, we refer to the process of determining which processing resource each actor executes on, and the ordering of execution among actors that share the same resource. By affecting key metrics that include performance, memory usage, and power consumption, scheduling often has significant impact on implementation quality.

The underlying graph representations for multimedia systems, especially for large-scale applications, often consist of smaller sub-structures that repeat multiple times. *Topological patterns* have been shown to enable more concise representation and direct analysis of such substructures in the context of high level DSP specification languages and design tools [5]. Furthermore, by allowing designers to explicitly identify such repeating structures, use of topological patterns provides an efficient alternative to automated detection of such patterns, which entails costly searching in terms of graph-isomorphism and related forms of computation. A topological pattern is inherently parameterized and provides a natural interface for parameterized scheduling, which enables efficient derivation of adaptive schedule structures that adjust symbolically in terms of design time or run-time variations.

In [6], we introduced a new dataflow-based design tool, called the *targeted dataflow interchange format* (*TDIF*), for design and implementation of embedded software for multimedia systems. TDIF is a companion design tool of the *Dataflow Interchange Format* (*DIF*) framework [7]. TDIF extends the capabilities of DIF with dynamic dataflow software synthesis, cross-platform actor design support, and dataflow-integrated features for instrumenting and tuning implementations.

In this paper, we go beyond our introduction to TDIF in [6], and introduce a novel schedule model called the *scalable schedule tree* for parameterized scheduling based on topological patterns. We have integrated this model into the DIF framework and TDIF environment by 1) providing a new language syntax for specifying topological patterns, and 2) developing a software plug-in for constructing internal representations and generating code for implementations targeted to different platforms based on our new schedule model. This integration provides new capabilities in the TDIF environment for application design, experimentation with parameterized schedules that are derived from scalable dataflow graph models, and implementation of multimedia signal processing systems that employ repetitive graph structures. Through a case study centered around an image registration application, we have validated our new methods and tools, and demonstrated their utility in the design and implementation of multimedia systems.

II. Related Work

In early design stages for describing multimedia applications as signal processing block diagrams, system blocks are treated as "black boxes," and designers focus on defining application specifications and features at a high level of abstraction. After a target platform is chosen, system blocks are manually transcoded, and the resulting implementations are tuned to match the platform. Such a design process, from an initial application description to a final implementation, often consists of several complex design steps that are linked by different design languages and tools, and ad-hoc transcoding processes. While targeting heterogeneous design platforms, such a process tends to be more error-prone and time-consuming due to the need for efficient coordination across different processor types. Therefore, a cross-platform design environment is needed that provides capabilities for the designer to experiment with key design phases — ranging from early design exploration to final implementation tuning — on different platforms.

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 [8], [9], [10], [7], [11], [12], [13]). 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.

For design and implementation for multimedia applications using dataflow, scheduling is a critical aspect of implementing dataflow graphs (e.g., see [4]). Parameterized schedules have been studied before (e.g., see [14], [15]), and previously, production and consumption rates were key dataflow graph aspects that were used to generate parameterized schedules. In topological patterns, even if production and consumption rates are fixed, the schedule is still scalable in terms of the numbers of actors and edges. Such scalability, when formulated in term of topological patterns, leads to new opportunities and constraints for developing parameterized scheduling techniques.

Early work on parameterized scheduling for dataflow graphs was done in the context of parameterized dataflow representations. *Parameterized dataflow* is a meta-modeling technique that can be applied to any underlying "base" dataflow model, such as SDF [16], CSDF [17], FRDF [18], and BDF [19], for dynamically reconfiguring the behavior of dataflow actors, edges, subsystems, and graphs through dynamic reconfiguration of parameter values [14]. Quasi-static scheduling techniques were developed for parameterized synchronous dataflow (PSDF) specifications [14], which is the integration of the parameterized dataflow meta-model with SDF as the base model. However, in this work, parameterized scheduling for scalable topologies was not addressed — the underlying sets of actors and edges were assumed to be fixed.

The *reactive process networks* (*RPN*) model of computation supports the construction of analysis and synthesis tools for dynamic streaming multimedia applications that include both event-based and dataflow-based computations [20]. RPN provides an integration framework with run-time reconfiguration for event and stream processing which is flexible to handle run-time scheduling decisions and may also be used to represent non-deterministic stream processing behaviors.

Using the *parameterized Kahn process network* (*PKPN*) model, designers can analyze the behavior of a parameterized system at runtime based on self-timed scheduling without introducing non-deterministic behaviors [21]. PKPN also automates the design process through integration with the Compaan/Laura tool [22].

The operational semantics of the RPN and PKPN models can be viewed as extensions of the Kahn process network (KPN) modeling framework [23], where processes execute concurrently, applying blocking reads to assess availability of data on their inputs, and control is incorporated into processes in a distributed fashion without use of a global scheduler. While these models lead to flexible and efficient execution of KPN-related models, they, like the parameterized dataflow framework, do not address the scheduling of scalable topologies.

In [6], the Targeted Dataflow Interchange Format (TDIF) is introduced for design, analysis, and implementation of embedded software for multimedia systems. In TDIF, designers construct schedules based on programming interfaces that are automatically generated from the TDIF tool. These programming interfaces provide a consistent, formal dataflow abstraction layer between designer-constructed schedules and the actors that are executed by the schedules.

In this paper, we extend the developments of [6] in a number of ways. First, we introduce a novel schedule model called the scalable schedule tree (SST) for representing scalable schedules based on topological patterns, and we integrate specification and code generation support for SSTs to enhance the capabilities of the TDIF environment. Rather than having designers specify schedules in terms of arbitrary target-language code that connects to TDIF-generated actor interfaces (as in [6]), we raise the level of abstraction for schedule specification by allowing SST-based specification of schedules. SSTs are specified programmatically using graph construction APIs associated with the SST formal model, which we have developed. Through code generation techniques that we have integrated into the TDIF environment, complete code that implements the scheduler for a given application can be automatically generated from an SST. Furthermore, in this paper, we use a more complex case study centered around an image registration application to demonstrate our methods and the overall utility of the TDIF environment.

III. Background

In this section, we summarize background on dataflow graph modeling, schedule representation, and design tool development that we build on in this paper.

A. 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* [24]. 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 (fireability 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.

B. 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 [7]. 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 supports 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 [7]. Although DIF-to-C supports only static dataflow applications — in particular, those that are based on synchronous dataflow (SDF) semantics [16] — the tool is capable of exploring various kinds of implementation trade-offs that are exposed effectively through DIF-based dataflow representations (e.g., see [7]).

C. Topological Patterns

For large-scale models of multimedia signal processing applications, the underlying dataflow graph representations often consist of smaller substructures that repeat multiple times. A method for scalable representation of dataflow graphs using *topological patterns* was introduced in [5]. Topological patterns, such as the *ring*, *butterfly*, and *chain* patterns, are pervasive in signal processing applications, including multi-dimensional signal processing systems, where processing of large scale dataflow structures is common.

Topological patterns enable concise representation and direct analysis of substructures in the context of high level DSP specification languages and design tools. Modeling based on topological patterns also provides a scalable approach to specifying regular functional structures that is formally integrated with the framework of dataflow. This integration allows not only for specification of functional patterns, but also for their analysis and optimization as part of the larger framework of dataflow.

For more details on modeling and design based on topological patterns, we refer the reader to [5].

D. Generalized Schedule Trees

The *generalized schedule tree* (*GST*) is a compact, tree-structured graphical format that can represent a variety of dataflow graph schedules [15]. In GSTs, each leaf node refers to an actor invocation, and each internal node n (called a *loop node*) is configured with an iteration count I_n for the associated sub-tree, where execution of the sub-tree rooted at n is repeated I_n times.

The GST has been demonstrated to represent looped schedules for dataflow graphs effectively in the context of static, non-scalable schedules (e.g., see [15]). In this paper, we go significantly beyond the capabilities of GSTs by formulating and implementing a novel schedule tree model for representing scalable schedules (i.e., schedules that symbolically accommodate variations in the numbers of actors and edges in the associated dataflow graphs). We refer to this new form of schedule tree as the *scalable schedule tree* (*SST*) model. Our implementation of the SST representation is integrated with the TDIF environment for generating platform-specific code from DIF models.

IV. Scalable Schedule Trees

In this section, we build on the GST representation, and develop a new formal method to formulate and represent a class of parameterized schedules. This targeted class of schedules is useful for implementing dataflow graph models that employ topological patterns, as we demonstrate in subsequent sections of this paper. Our new model for schedule representation is significantly more powerful than the original GST formulation, and as a target for scheduling techniques, this new model enables the development of correspondingly more powerful schedulers.

A *scalable schedule tree* (*SST*) has all of the features of a GST (see Section III), and additionally provides the following new features.

1. Parameterization. An SST has an associated parameter set K. Nodes within the schedule tree can be parameterized in terms of this parameter set (we will describe this in more detail below). The semantics of how SST parameters (i.e., values associated with elements of K) change is not specified in the SST model; rather, it is determined by the model of computation that is used for application specification (e.g., SDF with static graph parameters [25], parameterized dataflow [14], or scenario aware dataflow [26]), in conjunction with the scheduling strategy that is used to derive the schedule tree. This decoupling from parameter change semantics allows the SST model to be applied to a variety of different kinds of dataflow application models and design environments.

2. Guarded execution. An SST leaf node, which encapsulates a firing of an individual enable-invoke dataflow actor, has an optional *guarded* attribute, which indicates that firing of the corresponding actor should be preceded by a run-time fireability (*enabling*) check. Such an enabling check determines whether or not sufficient input data is available for the actor to fire. If sufficient input data is not available, the firing is aborted — i.e., the corresponding actor is effectively "skipped" during the current visitation of the leaf node. The guarded attribute of SSTs is motivated by the enable-invoke dataflow model of computation, where guarded executions play a fundamental role [27].

3. Dynamic iteration counts. Loop nodes can be dynamically parameterized in terms of SST parameters, which provides capabilities for data- or mode-dependent iteration in schedules. An SST loop node L can be viewed as a parameterizable form of the constant-iteration-count loop nodes in GSTs. An SST loop node L has an associated *iteration count evaluation function* $c_L : K \rightarrow Z+$, where $Z+$ represents the set of non-negative integers. An implementation of c_L takes as arguments zero or more of the parameters in K, and returns a non-negative integer (zero parameters are used if the iteration count is constant). Visitation of L begins by calling c_L to determine the iteration count, and then executing the subtree of L successively a number of times equal to this count. Note that c_L is evaluated by the enclosing schedule — as a schedule executes, it invokes c_L to determine the iteration count for the associated SST subtree execution.

4. Arrayed children. In addition to leaf nodes and SST loop nodes, there is a third kind of internal node called an *arrayed children node* (*ACN*). ACNs are perhaps the most distinctive aspect of SSTs, and the most closely related to topological patterns. These are discussed in more detail in the following subsection.

A. Arrayed Children Nodes

An ACN z has an associated parameter set P_z. Each $p \in P_z$ in turn has an associated evaluation function $f_p : K \rightarrow \nu_p$, where ν_p is the set of admissible values (parameter domain) of p, and again, K is the parameter set of the associated schedule tree.

An ACN z has an associated array $\mathrm{children}_z$, which represents an ordered list of candidate children nodes during any execution of the SST subtree rooted at z. For simplicity, we assume that $\mathrm{children}_z$ is a one-dimensional array, but the associated formulations can easily be extended to handle multi-dimensional arrays of candidate children. The array $\mathrm{children}_z$ has a positive integer size (denoted size_z), which gives the number of elements in the array. It is assumed that the array is indexed starting at 0.

Each element in $\mathrm{children}_z$ represents a schedule tree leaf node (i.e., an encapsulation of an actor in the enclosing dataflow graph), an SST loop node, or another SST — i.e., a “nested” SST. An ACN z also has three functions associated with it, which we denote as cinit_z, cstep_z, and climit_z, that determine how $\mathrm{children}_z$ is traversed during a given execution of the enclosing subtree. These functions take as arguments pre-specified subsets of the parameters of z, and return, respectively, a non-negative, positive, and non-negative integer. One or more of these functions can be constant-valued — dependence on parameter settings is not essential but rather a feature that is provided for enhanced flexibility.

B. SST Traversal Process

When an ACN z is visited during traversal (execution) of the enclosing schedule tree, the following sequence of steps, called the *SST traversal process*, is carried out.

(1) The parameter settings for z are updated by applying the evaluation function f_p for each parameter $p \in P_z$.

(2) The values of cinit_z, cstep_z, and climit_z are evaluated in terms of the updated parameter settings. These values are stored in temporary variables, which we denote as `I`, `s`, and `L`, respectively.

(3) The computation outlined by the pseudocode shown in Algorithm 1 is carried out, where `A` represents the array $\mathrm{children}_z$; `count` represents the iteration count evaluation function of the associated SST loop node; and `K` represents the set of parameters for the enclosing SST.

A generalization of SSTs can be envisioned in which arrays of candidate children are replaced by lists, and the visitation process for a generalized ACN g starts by applying a function g, which takes parameter settings for P_g as arguments, and returns a list of children in the order that they should be visited.

Fig. 1 shows a synthetic example of a nested SST, where the scheduling result S shows the sequence of actor executions that results from traversing the given SST.

Algorithm 1 Outline of the SST traversal process.

```
for (i = I; i <= L; i += s) {
   if A[i] is a leaf node {
       execute the actor encapsulated by A[i]
   } else if A[i] is an SST loop node {
       Z = count(K)
       execute the loop node subtree Z times
   } else { // A[i] is a nested SST
       recursively apply the SST traversal
               process to A[i]
   }
}
```

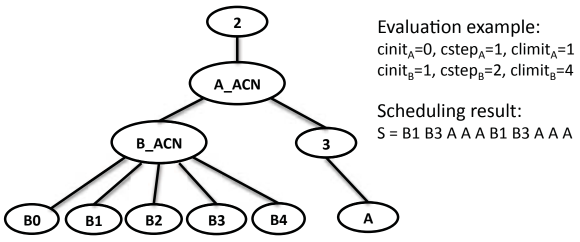

2
A_ACN
B_ACN
3
B0
B1
B2
B3
B4
A
Evaluation example:
cinit_A=0, cstep_A=1, climit_A=1
cinit_B=1, cstep_B=2, climit_B=4
Scheduling result:
S = B1 B3 A A A B1 B3 A A A

Fig. 1. An example of an SST.

C. Relationship to Scalable Dataflow

The form of scalability provided by SSTs, which can be viewed as *topological scalability*, is orthogonal to that provided by the *scalable dataflow* concept introduced by Ritz, Pankert, and Meyr [28]. The two techniques can be applied independently or jointly. In scalable dataflow, the objective is to execute block-processing versions of actors. Each scalable dataflow actor is programmed in terms of a *vectorization degree* N, which represents the number of firings of the actor that are executed together. This allows such an actor to process data in blocks of N units, and furthermore, to carry out internal computations in such a block-processed way, which can provide significantly increased throughput and data locality, possibly at the expense of latency and buffer memory requirements [29], [30].

While Ritz presents scalable dataflow in the context of SDF, referring to the model as *scalable SDF* or *SSDF*, the underlying form of scalability is more general and can be applied to arbitrary application programming interfaces (APIs) or software synthesis frameworks for signal processing dataflow graphs.

This form of vectorization-oriented scalability can be applied flexibly within SSTs. For example, an

SST loop node L can be connected as an element of $\mathrm{children}_\alpha$, where α is an ACN, and L contains as its single child the actor A that is to be vectorized. The loop count associated with L can then be passed dynamically to a vectorized implementation of A to execute A in a block-processing fashion.

V. The Targeted DIF Environment

The TDIF design tool, introduced in [6], builds on the capabilities of the DIF framework [7], and introduces new features for cross-platform actor design; static and dynamic scheduler implementation; and code generation for targeted platforms. TDIF consists of new plug-ins to the DIF environment that focus on efficient mapping of CFDF-based multimedia application representations onto embedded platforms. The core of the TDIF environment contains a new dataflow actor design language called the *TDIF language* for describing high level specifications of dataflow actors that can be retargeted across different platforms; object-oriented data structures for encapsulating contextual information associated with CFDF-based dataflow components; and application programming interfaces (APIs) that allow designers to design and experiment with these components along with scheduling strategies and run-time instrumentation techniques. By building on the CFDF model of computation, TDIF can flexibly accommodate both static and dynamically structured multimedia applications.

The TDIF tool is based on four software packages — the *TDIF compiler*, *TDIFSyn* software synthesis package, *TDIF run-time library*, and *Software synthesis engine*, and it currently supports C- and GPU-based implementations (i.e., implementations for CPU and GPU platforms). The GPU-based capabilities of TDIF are currently oriented towards NVIDIA GPUs, based on the CUDA programming framework [31]. 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.

A. SST Plug-In

A new software plug-in, called the *topological pattern plug-in*, to the DIF framework has been implemented to extend the DIF language (TDL) with support for topological patterns. The topological pattern plug-in also allows designers to construct SSTs for schedules associated with dataflow graphs that are specified in TDL, and that employ arbitrary numbers of topological pattern instantiations. The topological pattern plug-in integrates the SST formulations developed in Section IV as a new internal representation format and associated set of graph (schedule) transformations within the DIF framework. Topological patterns that are currently supported by TDL and defined as *pattern keywords* in the language include `chain`, `ring`, `merge`, `broadcast`, `parallel`, and `butterfly`. An example of a code segment in which topological patterns are specified in TDL will be shown in Section VI.

The topological pattern plug-in allows designers to construct SSTs, systematically maintain SSTs and SST subsystems (subtrees) as reusable design components, and link SSTs to TDIF code generation capabilities so that control code that implements a selected SST can be synthesized automatically to coordinate application execution. SSTs also provide formal representations that can be generated automatically at the back-end of automated schedule construction techniques, and provide a natural interface through which such techniques can be integrated into TDIF as scheduling plug-ins. Development of such automated scheduling plug-ins is a useful direction for further work.

In the topological pattern plug-in, an SST node can be instantiated in the form of either a leaf node or an internal node. An internal node can be configured with an iteration count or specified as an ACN. A leaf node instance is associated with an actor from the original (application-level) dataflow graph. An ACN instance in general contains pointers to the cinit, cstep, and climit functions, which are defined in Section IV. Through these pointers, the functions can be evaluated any time during traversal of the enclosing tree to obtain up-to-date values. Using a Java-based API for SST construction, designers can programmatically build up SSTs, and link the constructed SSTs with other phases of the overall TDIF design flow. This programmatic approach also facilitates iterative, experimentation-driven refinement and optimization of SSTs, as well as maintaining libraries of alternative SSTs that can be drawn upon to match different sets of implementation constraints. The abstract (language-independent) formulation of the SST also facilitates retargeting of our SST construction API to other languages — e.g., so that the methodology can be readily integrated into other design environments.

B. Integration in the TDIF Environment

In the TDIF environment [6], we have integrated specification and code generation support for SSTs. Through this integration, we have raised the level of abstraction for schedule specification by allowing SST-based specification of schedules, where leaf nodes in the schedule trees are connected to the TDIF-generated actor interfaces. As described in the previous section, an SST is specified programmatically using tree construction APIs associated with the SST internal representation.

Code generation in TDIF for an SST is carried out by applying depth first search to traverse the schedule tree, and invoking a specialized code generation module in each visitation step depending on the kind of node that is visited (leaf node, SST loop node, or ACN). During the traversal process for code generation, if a visited node x is an SST leaf node, then the guarded attribute of x (see Section IV) is first checked. If the guarded attribute is not set, then the associated actor is assumed to have sufficient input data whenever x is visited (e.g., through static scheduling analysis), so no run-time check for data availability is performed (i.e., code is not generated to perform such data availability checking). This allows results of static analysis to help streamline the generated code, while also providing flexibility for run-time checking when static analysis is not performed or does not guarantee fireability.

During the traversal process for code generation, if a visited node x is an SST ACN, then code is generated to evaluate cinit_x, cstep_x, and climit_x (through appropriate function pointers, macros or constants set up for these attributes), and generate code for a loop that iterates across the subtrees associated with selected children from $\mathrm{children}_x$. Code for the subtrees associated with $\mathrm{children}_x$ is then generated by recursively traversing these subtrees in a depth-first fashion.

When a loop node λ is visited during code generation traversal, code is generated to evaluate the associated iteration count evaluation function, and code is then generated for a loop structure that is controlled by this iteration count. To generate the body of this loop structure, the SST traversal process is recursively applied to each child node of λ.

The code generated from an SST, which implements the scheduler for the given application, can be linked together with a top-level C file that is automatically generated from the TDIFSyn software synthesis package, and actor code from the associated actor library to construct an executable that implements the application. For examples of scheduler code segments that are generated from SSTs, we refer the reader to [32].

Fig. 2 illustrates the design flow of TDIF, which incorporates specification and code generation support for SSTs and parameterized scheduling, as we have described. By following the methodology underlying this design flow, the designer can focus on design, implementation and optimization for dataflow actors and experiment with alternative scheduling strategies for specific 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 process.

VI. Case Study: Image Registration Application

To demonstrate the capabilities of the TDIF environment along with the new topological pattern plug-in, we use an image registration application based on the *Scale-Invariant Feature Transform* (*SIFT*)

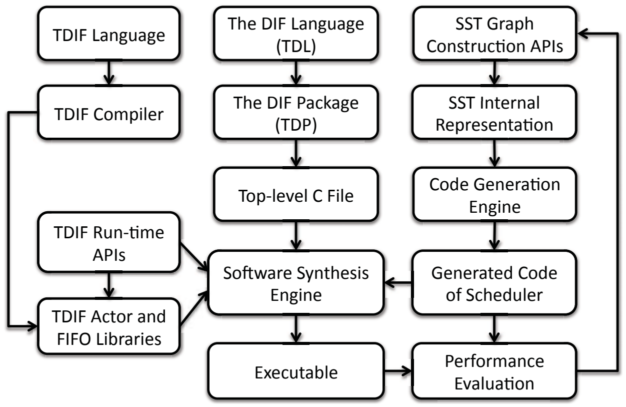


Fig. 2. TDIF-based design flow.

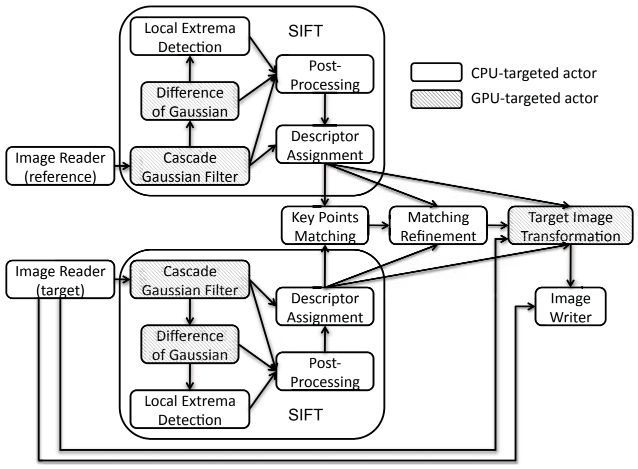


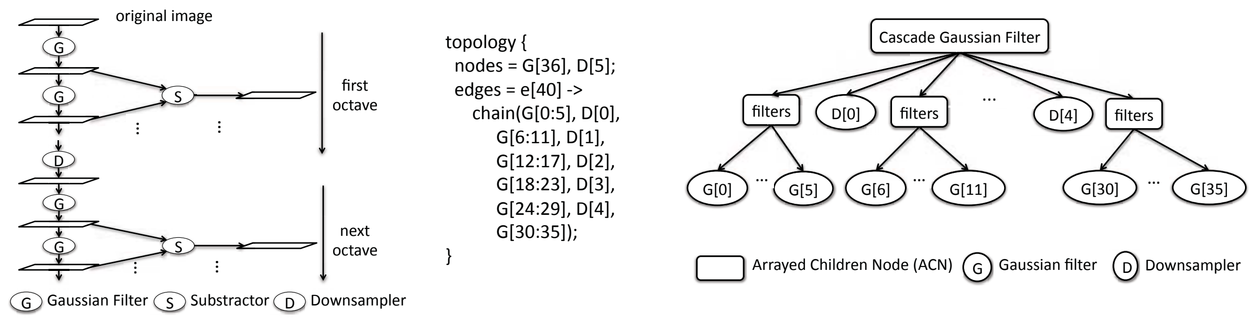
Fig. 3. Design flow for the targeted image registration application.

algorithm as a case study [33]. SIFT is a well-known algorithm in computer vision for feature detection and matching of images.

A. Application Overview

In the image registration application, two or more images of the same scene can be overlaid through the process of geometric alignment [34]. The SIFT algorithm provides a method to extract distinctive scale- and rotation-invariant features from images. SIFT can be used to perform feature matching between images that are taken from different views of the same scene. Fig. 3 shows a dataflow graph representation of this application.

In SIFT, as shown in Fig. 3, the `Cascade Gaussian Filter` actor implements a *cascade Gaus-*



```
topology {
  nodes = G[36], D[5];
  edges = e[40] ->
     chain(G[0:5], D[0],
          G[6:11], D[1],
          G[12:17], D[2],
          G[18:23], D[3],
          G[24:29], D[4],
          G[30:35]);
}
```

(a) Design process (b) TDL code (*o*=6 and *s*=6) (c) SST representation

Fig. 4. The cascade Gaussian filtering process in SIFT.

sian filtering subsystem that produces a series of Gaussian filtered images. Neighboring images that are filtered by the `Cascade Gaussian Filter` actor (e.g., see Fig. 4(a)) are subtracted by the `Difference of Gaussian` actor to produce a series of difference of Gaussian images. Then the `Local Extrema Detection` actor selects the maxima and minima of difference of Gaussian images as key point candidates. Each key point is selected only if it is larger or smaller than all of its 26 neighbors (8 neighboring pixels in the enclosing image and 18 neighboring pixels of the adjacent two images).

The `Post Processing` actor eliminates key points that are localized near the boundary of the image or localized along line segments or curves across which there are large gradients in pixel intensity. Orientation is assigned to each key point as well. Finally, image gradient information near the key points is extracted and stored as key point descriptors by the `Descriptor Assignment` actor.

When performing feature matching between two images, the `Key Points Matching` actor matches a key point i in an image A to a key point j in another image B only if the Euclidean distance between i's descriptor and j's descriptor multiplied by a user defined threshold is not greater than the Euclidean distance of i's descriptor to all other key point descriptors [33].

Since key points matching may generate false matches between the reference image and the target image, a refinement step is needed in order to eliminate these false matches. For such matching refinement computation, we applied the *random sample consensus* (*RANSAC*) algorithm [35]. RANSAC is an iterative method to estimate the parameters of a model from a set of observed inlier data that is contaminated by a set of outlier data. In our case, inliers are correct matches and outliers are false matches.

The `Target Image Transformation` actor takes the outputs produced by the SIFT computation, the refined matching result, and the target image, and produces the resulting registered image. Here, a rigid transformation process, which includes steps of translation, rotation, and scaling, is used to determine the corresponding positions (in the target image) of each pixel in the registered image. These positions are coordinates with fractions. We use bilinear interpolation to determine each pixel value in the registered image by taking weighted average values of four surrounding pixels in the target image to reduce visual distortion.

B. Applying the Scalable Schedule Tree

The cascade Gaussian filtering subsystem in SIFT is a relevant case study for experimenting with topological patterns and SSTs because it can be modeled naturally in terms of parameterized topologies. Cascade Gaussian filtering can be modeled as a dataflow graph consisting of actors that perform Gaussian filtering and downsampling computations. These operations can be divided into a set of o groups, such that each group involves s filtering steps. Both o and s are parameters that can be configured by the designer (e.g., to explore trade-offs between processing complexity and image processing accuracy).

In the cascade Gaussian filtering process illustrated in Fig. 4(a), the original image is convolved with the first filter. The filtered image is saved and then convolved with the next filter, and so on. After one group of filtering operations is carried out, s different blurred Gaussian images are labeled as a separate octave. The next step is to downsample the last image of the previous octave by a factor of two. This process, as shown in Fig. 4(a), repeats until o octaves of images are produced.

The topological pattern underlying this subsystem with $o = 6$ and $s = 6$ is a chain (linear arrangement of actors) that can be specified using the TDL code shown in Fig. 4(b). Here, an array of 40 edges is instantiated by connecting 41 specified nodes (six groups of six nodes each that are interleaved with five individual nodes) in a chain.

Note that the binding of nodes to specific functions is done in a separate part of the TDL specification that is dedicated to assigning *actor attributes*. This part of the specification is not shown for conciseness (for details, we refer the reader to [7]).

In this example of cascade Gaussian filtering, since both o and s are parameters that can be configured, one can naturally derive a nested SST as shown in Fig. 4(c). Such a representation provides a formal, target-language-independent model of schedule structure that can be applied to coordinate execution for this subsystem in a manner that is parameterized across two dimensions.

In the case that $o = 6$ and $s = 6$ (as shown in Fig. 4(c)), the `Cascade Gaussian Filter` ACN has 11 children nodes, which include 6 nested ACNs, each labeled as `filter`, and 5 `downsampler` actors encapsulated as leaf nodes, which are labeled as `D[0]`, `D[1]`, . . . , `D[4]`. Each of these leaf nodes represents an encapsulation of a `downsampler` actor in the cascade Gaussian filtering application. Each internal node labeled `filter` is an ACN that contains 6 children nodes, where each of these children nodes represents an encapsulation of a `Gaussian filtering` actor in the application.

C. *Evaluation in Terms of Coding Efficiency*

Our design framework for specifying topological patterns enables concise and scalable representation of multimedia applications. To help quantify this kind of benefit, we apply an evaluation metric called the *lines of code* (*LOC*), which is the number of lines of code required for an application. Unless otherwise specified, the LOC cost refers to code that the designer needs to manually provide (e.g., in contrast to code that is automatically generated or reused from some other part of an implementation). The LOC metric has been widely used in various methods, such as the Constructive Cost Model [36], SEER for Software [37], and the Putnam model [38], for estimating software development effort. We apply this metric on various applications (listed in Table I), including the cascade Gaussian filtering application, that are specified with and without use of topological patterns. Note that use of the LOC metric is facilitated by employing lines that have reasonably consistent complexity — we have tried to follow such an approach in our comparisons. A more accurate metric along these lines would be to compare the numbers of lexical tokens. Exploration of such a more detailed metric is an interesting direction for further study.

We first compare LOC evaluation results by using TDL with and without the support of topological patterns. Table I shows a comparison result in terms of LOC for TDL specifications with and without the support of topological patterns for different applications. In this comparison, we compare the specifications of topology components in terms of nodes and edges in TDL, where, for consistency, each node and edge declaration occupies a separate line of code.

We also assess the LOC benefit for the cascade Gaussian filtering application that is obtained from code generation in the TDIF environment. More specifically, we compare the LOC cost of an implementation that uses code generation and the LOC cost of the generated code (i.e., the LOC cost of the generated implementation). This gives a comparison of the complexity of the complete implementation generated using TDIF compared to the complexity of the code that the designer has to write and maintain as source code.

As discussed in Section V, the TDIF tool contains a code generator to translate SSTs into C code that implements the corresponding schedules [32]. The TDIF environment also provides tools to translate concise specifications of actor interface information (input, output, state, etc.) into APIs for implementing the actors according to standardized dataflow implementation structures in TDIF [6]. Additionally, the

TABLE I

LOC COMPARISONS FOR TDL SPECIFICATIONS WITH AND WITHOUT THE SUPPORT OF TOPOLOGICAL PATTERNS (TPS).

Application	without TPs	with TPs
Cascade Gaussian filtering (CGF)	81	3
Image registration that contains CGF as a subsystem	205	18
JPEG encoder	37	9
FFT (size $N = 8$)	32	2

TDIF environment provides translation from DIF specifications into top-level C language implementations that construct and execute the specified dataflow graphs.

Table II summarizes the LOC costs for different implementation components of the `Cascade Gaussian Filter` subsystem when code generation is used. These are the costs for the designer-written code that can be viewed as input to the TDIF toolset. These costs are listed as functions of the numbers of dataflow graph actors n and edges e in the scalable application model, and the total LOC costs c in the designer-written component of the actor implementations.

On the other hand, Table III shows the LOC costs of the complete generated implementation — i.e., the generated code together with the designer-written TDIF input code that is used directly (without translation) in the implementation.

In the `Cascade Gaussian Filter` subsystem, the underlying topological pattern is a chain, and the number of edges is of the same order as the number of nodes. Thus, comparing the LOC listings in the two tables, we see that as the number of nodes n in the application is increased, the ratio of the designer-written LOC cost to the complete implementation LOC cost decreases. For example, with $n = 41$, $e = 40$, and $c = 2,960$ in the discussed `Cascade Gaussian Filter` subsystem, the designer-written code only takes about 41% of the complete generated implementation using the TDIF environment. This helps to quantify the utility of the TDIF tool in terms of LOC costs as a function of graph complexity. This comparison incorporates the use of topological patterns, which helps to reduce the LOC cost for the top-level DIF specification.

D. Cross-Platform Experimentation

TDIF includes capabilities for targeting CUDA-enabled graphics processing units (GPUs) in addition to pure C code ("CPU targeted") implementations [6]. As part of this application case study, we experi-

TABLE II

LOC COSTS FOR DESIGNER-WRITTEN CODE IN THE TDIF ENVIRONMENT.

Top-level DIF specification	$5n + e + 6$
TDIF specification	$5n$
Building SST	16
Actor development	c
Total	$10n + e + 22 + c$

TABLE III

LOC COSTS FOR THE IMPLEMENTATION GENERATED BY THE TDIF ENVIRONMENT.

Top-level C file	$9n + 6$
Function declaration	$56n$
Scheduling APIs	$22n$
Scheduling file header	$2n + 5$
Scheduling	$41n$
Actor development	c
Total	$130n + 11 + c$

mented with the CUDA-targeted synthesis capability of TDIF for implementing different actors for the image registration application. As shown in Fig. 3, parts of the application are a good match for GPU execution, and thus, the synthesized GPU implementation exhibits significant performance improvement. This aspect of our case study validates the utility of topological patterns and the developed tool chain in enhancing application specification and scalability in the context of cross-platform experimentation to explore trade-offs on alternative targets. Linkage to such experimentation capabilities is important for multimedia-oriented tools since there is a wide variety of relevant platforms available for multimedia system implementation.

In these experiments, input to the application is a 1200×900 gray-scale bitmap image, and the implementations are executed on a 3GHz PC with an Intel CPU that is equipped with 4GB RAM, and co-located with an NVIDIA GTX260 GPU. This GPU has a 576MHz graphics clock and a 1.242GHz processor clock. The GPU block size and grid size are set to 256 and 4219, respectively. This latter value

TABLE IV

PERFORMANCE COMPARISON FOR CPU-TARGETED AND GPU-TARGETED IMPLEMENTATIONS.

	CPU(sec)	GPU(sec)	Speedup
`Cascade Gaussian Filter`	11.896	0.416	28.60
`Difference of Gaussian`	0.584	0.012	48.67
`Target Image Transformation`	0.614	0.017	36.12
Overall image registration application	55.575	30.523	1.82

(the grid size γ) is computed as the ceiling of the image size divided by the block size — i.e.,

$$\gamma = ceil(\frac{1200 \times 900}{256}) = 4219.$$

Table IV shows a performance comparison between CPU-targeted and GPU-targeted implementations for the GPU-targetable actors, as well as for the overall image registration application, using the TDIF environment. We note that for the CPU-targeted implementation of the overall image registration application, all of the actors in the application are implemented in C (i.e., without GPU acceleration). On the other hand, for the GPU-targeted implementation, the GPU-targetable actors such as `Cascade Gaussian Filter`, `Difference of Gaussian`, and `Target Image Transformation` are implemented in CUDA and the remaining are implemented in C. Furthermore, implementations of the cascade Gaussian filtering subsystem are generated by TDIF based on SSTs that exploit topological pattern structures in the application specifications.

The results are obtained according to the average execution time for 10 runs in each of the two cases. The results show that GPU acceleration provides significant benefit in this application, and validates the retargetability of our use of topological patterns and SSTs in TDIF. However, the application-level speedup is less than the corresponding actor-level speedups. This is due to factors such as context switch overhead and communication cost for memory movement between CPU- and GPU-targeted subsystems. These factors are associated with overall schedule coordination in the application implementations. Use of the TDIF environment allows us to obtain such a comparison with relatively high coding efficiency, and a correspondingly high degree of automation, as demonstrated in Section VI-C. This is due to the high level of abstraction and accompanying formal modeling capabilities provided by TDIF and the associated TDL programming features. Use of topological patterns helps to enhance the coding efficiency and raise the level of abstraction further by representing applications in terms of scalable, higher level constructs that are complementary to conventional forms of hierarchy in graphical design specifications.

VII. 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. We have presented a novel scalable schedule tree (SST) model for representing parameterized schedule structures based on topological patterns. We have also presented a new plug-in to the DIF framework for specifying SSTs that execute dataflow-based application models containing topological patterns, and we have demonstrated the integration of this plug-in with TDIF to provide code generation from SSTs to platform-specific implementations. Through a case study centered around an image registration application, we have validated our new methods and tools, demonstrated their utility in cross-platform design, and evaluated their coding and performance efficiency. Useful directions for further work include exploring SSTs that incorporate more complex forms of adaptivity, and supporting code generation from TDIF to additional classes of platforms, such as FPGAs and multicore digital signal processors.

VIII. Acknowledgement

This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-11-1-0049, Laboratory for Telecommunication Sciences, and US National Science Foundation. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this article are those of the authors and do not reflect the views of AFRL, LTS, or NSF.

References

[1] Texas Instruments, *Military Multimedia Video Processor (MVP) 320C8X Data Sheet: SMJ320C80*, June 2002.

[2] Broadcom, *High-definition 720P Mobile Multimedia Processor: BCM2727*, October 2007.

[3] NVIDIA Corporation, *Whitepaper: the Benefits of Multiple CPU Cores in Mobile Devices*, 2010.

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

[5] N. Sane, H. Kee, G. Seetharaman, and S. S. Bhattacharyya, "Scalable representation of dataflow graph structures using topological patterns," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, San Francisco Bay Area, USA, October 2010, pp. 13–18.

[6] C. Shen, H. Wu, N. Sane, W. Plishker, and S. S. Bhattacharyya, "A design tool for efficient mapping of multimedia applications onto heterogeneous platforms," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011, 6 pages in online proceedings.

[7] 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.

[8] 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.

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

[10] 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, pp. 1710–1714.

[11] 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.

[12] W. Thies, M. Karczmarek, and S. Amarasinghe, “StreamIt: A language for streaming applications,” in *Proceedings of the International Conference on Compiler Construction*, 2002, pp. 179–196.

[13] 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, pp. 873–876.

[14] 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.

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

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

[17] 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.

[18] H. Oh and S. Ha, “Fractional rate dataflow model for efficient code synthesis,” *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 37, pp. 41–51, May 2004.

[19] 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*, October 1994, pp. 508–513.

[20] M. Geilen and T. Basten, “Reactive process networks,” in *Proceedings of the International Workshop on Embedded Software*, September 2004, pp. 137–146.

[21] H. Nikolov, T. Stefanov, and E. Deprettere, “Modeling and FPGA implementation of applications using parameterized process networks with non-static parameters,” in *Proceedings of the IEEE Symposium on FPGAs for Custom Computing Machines*, 2005, pp. 255–263.

[22] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere, “System design using Kahn process networks: the Compaan/Laura approach,” in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, February 2004, pp. 340–345.

[23] G. Kahn, “The semantics of a simple language for parallel programming,” in *Proceedings of the IFIP Congress*, 1974, pp. 471–475.

[24] 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.

[25] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharyya, "Gabriel: A design environment for DSP," *IEEE Transactions on Acoustics, Speech, and Signal Processing*, vol. 37, no. 11, pp. 1751–1762, November 1989.

[26] B. D. Theelen, M. C. W. Geilen, T. Basten, J. P. M. Voeten, S. V. Gheorghita, and S. Stuijk, "A scenario-aware data flow model for combined long-run average and worst-case performance analysis," in *Proceedings of the International Conference on Formal Methods and Models for Codesign*, July 2006, pp. 185–194.

[27] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya, "Heterogeneous design in functional DIF," in *Transactions on High-Performance Embedded Architectures and Compilers IV*, Per Stenström, Ed., vol. 6760 of *Lecture Notes in Computer Science*, pp. 391–408. Springer Berlin / Heidelberg, 2011.

[28] S. Ritz, M. Pankert, and H. Meyr, "High level software synthesis for signal processing systems," in *Proceedings of the International Conference on Application Specific Array Processors*, August 1992, pp. 679–693.

[29] S. Ritz, M. Pankert, and H. Meyr, "Optimum vectorization of scalable synchronous dataflow graphs," in *Proceedings of the International Conference on Application Specific Array Processors*, October 1993, pp. 285–296.

[30] M. Ko, C. Shen, and S. S. Bhattacharyya, "Memory-constrained block processing for DSP software optimization," *Journal of Signal Processing Systems*, vol. 50, no. 2, pp. 163–177, February 2008.

[31] *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide, Version 1.0*, June 2007.

[32] S. Wu, "Representation and scheduling of scalable dataflow graph topologies," M.S. thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, 2011.

[33] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[34] B. Zitova and J. Flusser, "Image registration methods: a survey," *Image and Vision Computing*, vol. 21, pp. 977–1000, 2003.

[35] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, June 1981.

[36] B. Boehm, B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby, and B. Steece, "Cocomo ii model definition manual," Tech. Rep., University of Southern California, 2000.

[37] L. Fischman, K. McRitchie, and D. D. Galorath, "Inside seer-sem," *Journal of Defense Software Engineering*, pp. 26–28, April 2005.

[38] L. H. Putnam and W. Myers, *Five Core Metrics : The Intelligence Behind Successful Software Management*, Dorset House Publishing, 2003, New York, New York, USA.

[39] I. Cho, C. Shen, S. Potbhare, S. S. Bhattacharyya, and N. Goldsman, "Design methods for wireless sensor network building energy monitoring systems," in *Proceedings of the IEEE International Workshop on Practical Issues in Building Sensor Network Applications*, Bonn, Germany, October 2011, pp. 974–981.