ABSTRACT

Title of Thesis: Synthesis of Embedded Software using Dataflow Schedule Graphs

Abhay Raina, Master of Science, 2017

Thesis directed by: Professor Shuvra Bhattacharyya
Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies

In the design and implementation of digital signal processing (DSP) systems, dataflow is recognized as a natural model for specifying applications, and dataflow enables useful model-based methodologies for analysis, synthesis, and optimization of implementations. A wide range of embedded signal processing applications can be designed efficiently using the high level abstractions that are provided by dataflow programming models. In addition to their use in parallelizing computations for faster execution, dataflow graphs have additional advantages that stem from their modularity and formal foundation. An important problem in the development of dataflow-based design tools is the automated synthesis of software from dataflow representations.

In this thesis, we develop new software synthesis techniques for dataflow based design and implementation of signal processing systems. An important task in software synthesis from dataflow graphs is that of *scheduling*. Scheduling refers to the assignment of actors to processing resources and the ordering of actors that share

the same resource. Scheduling typically involves very complex design spaces, and has a significant impact on most relevant implementation metrics, including latency, throughput, energy consumption, and memory requirements. In this thesis, we integrate a model-based representation, called the *dataflow schedule graph* (*DSG*), into the software synthesis process. The DSG approach allows designers to model a schedule for a dataflow graph as a separate dataflow graph, thereby providing a formal, abstract (platform- and language-independent) representation for the schedule.

While we demonstrate this DSG-integrated software synthesis capability by translating DSGs into OpenCL implementations, the use of a model-based schedule representation makes the approach readily retargetable to other implementation languages. We also investigate a number of optimization techniques to improve the efficiency of software that is synthesized from DSGs. Through experimental evaluation of the generated software, we demonstrate the correctness and efficiency of our new techniques for dataflow-based software synthesis and optimization.

Synthesis of Embedded Software using Dataflow Schedule Graphs

by

Abhay Raina

Thesis submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Master of Science
2017

Chair/Advisor: Dr. Shuvra S. Bhattacharyya
Dr. Rance Cleaveland
Dr. Gang Qu

Acknowledgments

I would sincerely like to thank Prof. Shuvra S. Bhattacharyya for giving me a chance to work with him and offering immense support, guidance, and encouragement for the past one year. He has always been very considerate and has patiently answered all my doubts and questions no matter how trivial they were. He helped me in overcoming the roadblocks that stood there in the way of the of completion of my thesis. His effective organizational skills greatly fostered the development speed of this project and at the same time, has helped me to gain invaluable skills to organize my work. It has been an enriching experience working with him.

I also want to thank my committee members, Professor Qu & Professor Cleaveland for providing insightful and valuable feedback. I am really grateful to my colleagues Yanzhou Liu, Allen Lin, Kyunghun Lee and Jiahao Wu. They helped me in understanding more about different parts of the project at various stages. Yanzhou and Allen were especially instrumental in helping me to get started with LIDE-OCL and DIF-GPU respectively. Last but not least I give my heartfelt thanks to my worthy parents for their immense motivational support.

Table of Contents

List of Figures

Chapter 1: Introduction

In the design and implementation of digital signal processing (DSP) systems, dataflow is recognized as a natural model for specifying applications, and dataflow enables useful model-based methodologies for analysis, synthesis, and optimization of implementations. A wide range of embedded signal processing applications can be designed efficiently using the high level abstractions that are provided by dataflow programming models (e.g., see [6]).

In dataflow-based modeling and design of DSP systems, applications are represented as directed graphs in which vertices correspond to signal processing hardware/software modules, such as digital filters and fast Fourier transforms (FFTs), and each edge represents the flow of data from the output of one vertex to the input of another. Vertices in DSP-oriented dataflow graphs are referred to as *actors*. Edges in dataflow graphs can be viewed as first in, first out (FIFO) channels that buffer data as it passes between pairs of communicating actors. Dataflow representations are useful in exposing parallelism in programs, which has motivated their extensive study in the context of parallel computation (e.g., see [13]).

In addition to the directed graph structure of dataflow representations, another distinctive feature in dataflow is that actor execution is decomposed naturally into

discrete units of execution, which are called *firings* of the associated actor [20]. Each firing of an actor A consumes a well defined amount of data values (referred to as *tokens*) from its input ports and produces a well defined number of tokens on its output ports. These numbers of tokens produced and consumed are referred to as the *production and consumption rates* that are associated with the given firings and actor ports. Information about production and consumption rates is often of great relevance in deriving efficient implementations from dataflow graphs [6].

In addition to their use in parallelizing computations for faster execution, dataflow graphs have additional advantages that stem from their modularity and formal foundation. For example, dataflow models can be applied to optimize memory requirements, enhance portability, and improve energy efficiency [6].

Figure 1.1 shows a simple example of a dataflow graph. This graph consists of five actors, numbered $1, 2, \ldots, 5$, and six FIFO channels that are modeled by the six edges in the graph. For example, the edge directed from Actor 5 to Actor 1 indicates that data produced as output by Actor 5 is consumed as input by Actor 1.

An important problem in the development of dataflow-based design tools is the automated synthesis of software from dataflow representations. Various software synthesis environments for dataflow environments have been presented in the literature (e.g., see [23, 24, 29, 30, 32]), and dataflow-based software synthesis continues to be an active area of research in the embedded systems, electronic design automation, and signal processing communities. In this thesis, we develop new software synthesis techniques for dataflow based design and implementation of signal

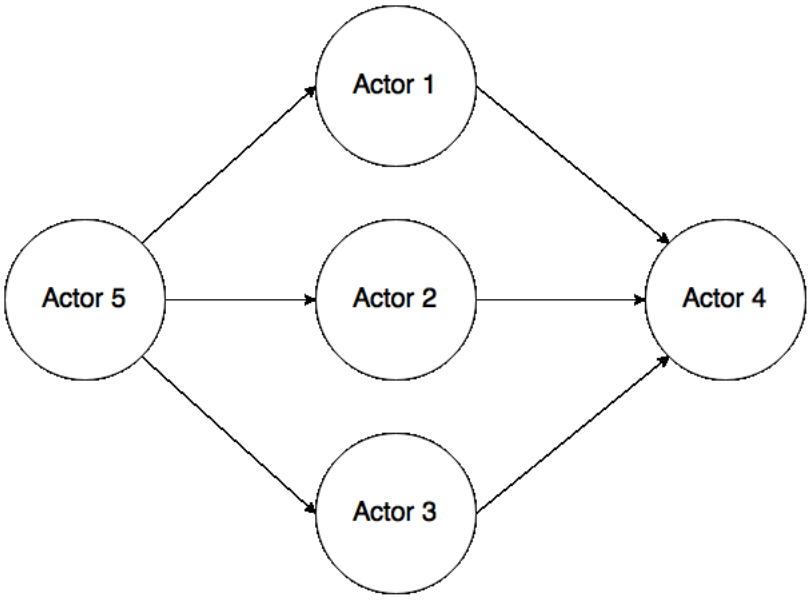

Figure 1.1: A simple example of a dataflow graph.

processing systems.

An important task in software synthesis from dataflow graphs is that of *scheduling*. Scheduling refers to the assignment of actors to processing resources and the ordering of actors that share the same resource. Scheduling typically involves very complex design spaces, and has significant impact on most relevant implementation metrics, including latency, throughput, energy consumption, and memory requirements. Our work in this thesis builds on a model-based representation, called the *dataflow schedule graph* (*DSG*), which has been introduced in prior work to model schedules for dataflow graphs [31]. The DSG approach allows designers to model a schedule for a dataflow graph as a separate dataflow graph, thereby providing a formal, abstract (platform- and language-independent) representation for the schedule.

A distinguishing aspect of our approach to software synthesis, compared to

most related software synthesis techniques, is that we leave the design of the schedule up to the designer rather than generating the schedule automatically as part of the synthesis process. The schedule design is modeled by the designer using the DSG model described above.

Requiring the designer to specify the schedule has the disadvantage of requiring more effort by the designer, but offers the advantage of providing more flexibility to designers who wish to have more control over the implementation process or who wish to target architectures or design constraints that are not well supported by available, fully-automated software synthesis processes. Thus, while at first the concept of a designer-specified schedule may seem to be purely a limitation, it in general actually represents a trade-off. This thesis represents an effort to investigate the side of this trade-off that favors giving more control and flexibility to designers.

More specifically, in this thesis we develop methods to synthesize embedded software for implementing schedules from abstract DSG representations of the schedules. While we demonstrate this software synthesis capability by translating DSGs into OpenCL implementations, the use of a model-based schedule representation makes the approach readily retargetable to other implementation languages. We also investigate a number of optimization techniques to improve the efficiency of software synthesized from DSGs. We experiment with our proposed new software synthesis techniques by implementing them in the dataflow interchange format (DIF) environment, which is a software tool that enables experimentation with new kinds of dataflow-based techniques for modeling, analysis, and optimization [12]. We demonstrate the correctness and efficiency of our software synthesis techniques through

experimental evaluation of the generated software from a number of application examples, including an example of a jitter measurement application for wireless communications.

Figure 1.2 illustrates a specific design flow to which our proposed new software techniques can be applied. In this approach, the user specifies a dataflow model of an application (*application graph*) and DSG model of a schedule for the application graph. These two graphs are specified using the DIF Language, which is a language for specifying abstract signal processing dataflow graphs. An implementation of the DIF Language is included as part of the DIF environment described above. Our software synthesis tool receives the application graph and DSG as input, and generates a software implementation of the DSG together with the application in the target language (OpenCL in our case). The software synthesis process assumes that implementations of the application graph actors in the target language are available. These actor implementations are instantiated in the generated code. Configuration of and communication between these actors is fully coordinated in the generated code along with execution of the designer-specified schedule.

A DSG can be a *sequential DSG* or a *concurrent DSG* [31]. A concurrent DSG is composed of multiple sequential DSGs, where communication between different sequential DSGs is carried through special DSG actors that are devoted to inter-processor or inter-thread communication. In this thesis, we are concerned primarily with sequential DSGs, and henceforth, when we write "DSG", we implicitly mean "sequential DSG", unless otherwise stated. We envision that the methods and tools developed in this thesis will be useful in the context of concurrent DSGs — for

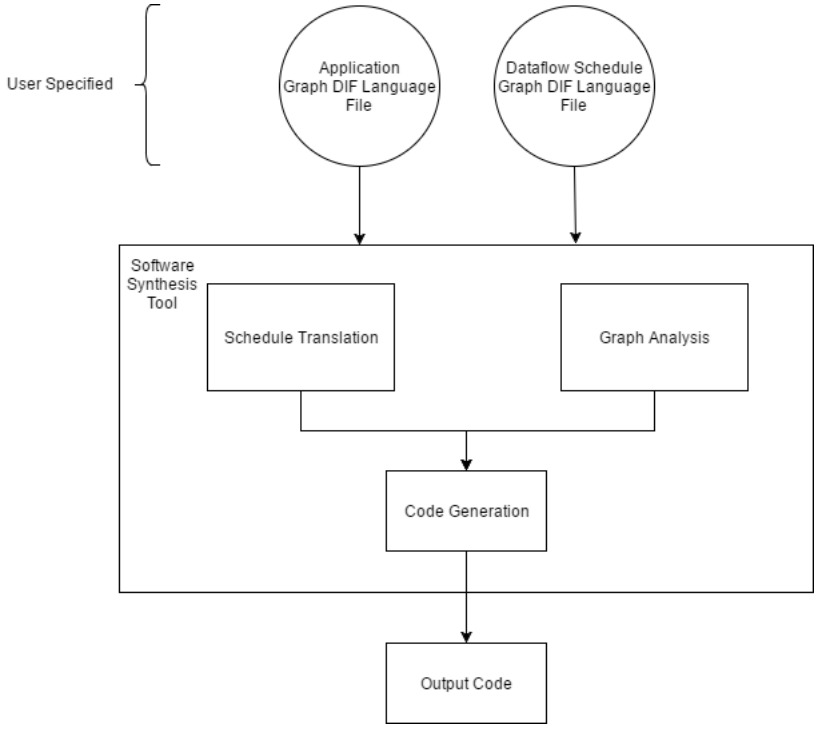

Figure 1.2: An illustration of a design flow to which our proposed new software synthesis techniques can be applied.

example, to synthesize and optimize code that implements the individual sequential DSGs that make up a concurrent DSG. Extension of the developments in this thesis to support concurrent DSGs is a useful topic for future work.

Chapter 2: Background

In this section, we cover background on dataflow modeling and software synthesis that is relevant for this thesis.

2.1 Synchronous Dataflow

As we discussed in Chapter 1, the dataflow graphs that we are concerned with in this thesis consist of vertices, called actors, which represent discrete computations, and edges, which correspond to FIFO communication channels between actors.

Synchronous dataflow (*SDF*) is a special case of dataflow where the production and consumption rates of the actors are specified and known in advance. Fig. 2.1 shows an example of an SDF graph. The numbers above the edges represent the numbers of tokens that will be consumed from and produced onto each edge when its sink and source actors fire, respectively. For example, *Actor 4* in Fig. 2.1 cannot fire until 4 tokens are available on each of the input edges *e1* , *e2* and *e3*.

A properly-constructed SDF graph can be run indefinitely (e.g., by encapsulating software for the graph within an infinite loop). Moreover, such indefinite or unbounded execution can be performed with finite memory requirements that can be predicted at compile time [19]. This capability for indefinite execution under

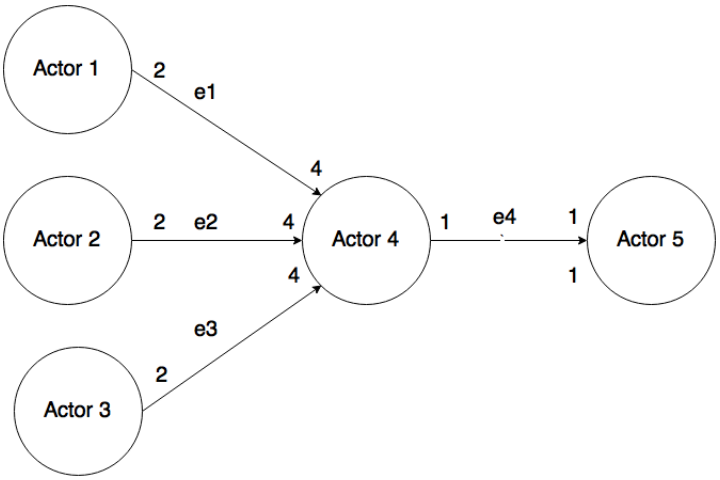

Figure 2.1: An example of a synchronous dataflow graph.

bounded memory is of great utility in embedded signal processing.

2.2 Core Functional Dataflow

From the previous section, we discussed how specifying the production and consumption rates a priori, as in the SDF model, can be advantageous. However, some applications involve dynamics in the rates at which data is produced and consumed from actors. SDF is not adequate for working with these *dynamic dataflow* applications. Various forms of dynamic dataflow have been introduced to address this limitation (e.g., see [6]). These dynamic dataflow models of computation provide more generality compared to SDF in expressing application behavior.

Core functional dataflow (CFDF) is a specific dynamic dataflow model of computation that we apply in this thesis[25]. In CFDF, each actor has an associated set of *modes*. An actor can have any positive integer number of modes. Each mode

is associated with constant production and consumption rates ("dataflow rates") on the actor ports. However, the production and consumption rates can vary across different modes, and the mode of an actor can change from one firing to the next.

Each CFDF actor has a *current mode*, which can be viewed as part of its internal state. When a CFDF actor is fired, it executes based on its current mode, and produces and consumes data to/from its ports based on the constant dataflow rates associated with this mode. As part of each firing, a CFDF actor also determines its *next mode*, which in turn determines how much input data and output buffer space (on the actor output edges) must be available to launch the next firing of the actor. When the actor is fired again, this next mode becomes the actor's current mode, and its production and consumption rates are governed by that mode.

2.3 Dataflow Schedule Graphs

As motivated in Chapter 1, dataflow models are used extensively in the design and implementation of DSP applications, and scheduling is an important aspect in the process of mapping dataflow models into efficient implementations. Some scheduling techniques are oriented toward simplicity of scheduler implementation or fast generation of schedules, and do not incorporate optimization of the constructed schedules. Other schedulers, which we refer to as *optimizing schedulers*, are designed to optimize relevant metrics in the targeted software implementations. These metrics include latency, throughput, code size, buffer memory requirements, and energy consumption.

A wide variety of scheduling techniques has been developed, with most of these being optimizing scheduler techniques. Many kinds of optimizing schedulers exist, and these target different subsets of metrics (e.g., scheduling for throughput maximization or for joint minimization of code size and buffer memory requirements).

A great deal of heterogeneity in dataflow-based design flows is brought about by this large and growing variety of scheduling techniques, along with the diversity in their targeted metrics, and the diversity in the hardware platforms on which the derived schedules are to be executed. To help manage this heterogeneity, the concept of a *model-based representation for dataflow schedules* is useful. This concept has been investigated in recent prior work, such as that by Wu et al. [31], and more recently, by Zebelein [33]. While conventional use of dataflow graphs in DSP system modeling is for the modeling of application behavior, model-based scheduling representations provide for abstract modeling of schedules.

In this thesis, we apply a specific form of model-based schedule representation called the dataflow schedule graph (DSG) [31]. DSGs are used to model schedules for CFDF-based application representations. Like CFDF graphs, DSGs are based on dataflow semantics. When a DSG G_s is used to model the schedule for a CFDF-based application representation G_a, we distinguish the two graphs by referring to G_a as the *application graph* that is associated with the *schedule graph* (DSG) G_s. DSGs apply to the highly expressive CFDF model of computation. Thus, they are significantly more flexible compared to prior model-based, dataflow schedule representations, such as the synchronization graph representation [4], which were restricted to SDF- or cyclo-static dataflow (CSDF) [7] application graphs.

DSGs are constructed using two different types of actors, which are referred to as *reference actors* (*RAs*) and *schedule control actors* (*SCAs*). Each RA r is associated with a specific application graph actor, which is denoted by $ref(r)$ and referred to as the *referenced actor* of r. The RA r can be viewed intuitively as a wrapper around the *guarded firing* of $ref(r)$ in its enclosing application graph. Here, by a guarded firing of an application graph actor, we mean the execution of the actor *if it is enabled*. If the actor is not enabled, then a guarded firing is effectively like a no-operation.

An RA has two associated sub-functions, called the *pre* and *post* functions of the RA, that provide placeholders for optional preprocessing and postprocessing that can be performed prior to and after, respectively, the guarded firing of $ref(r)$ that is associated with each firing of r.

Intuitively, SCAs are used to perform actions to control the order in which subsets of RAs are fired. SCAs provide an extensible set of constructs to control the flow of RA firings, together with the guarded firings of their encapsulated referenced actors. An example of an SCA is the "loop" SCA, which is illustrated in Fig. 2.2. The loop SCA can be used to execute individual RAs or chains of connected RAs repeatedly until some pre-specified or data-dependent termination condition is met. A pair of related SCAs that provide different control functionality is the "if" SCA together with the "fi" SCA (sse Fig. 2.3). These SCAs can be used to conditionally execute portions of a DSG. Another pair of related SCAs is the "snd" SCA and "rec" SCA, which are used for interprocessor communication and synchronization when multiple DSGs are executed concurrently.

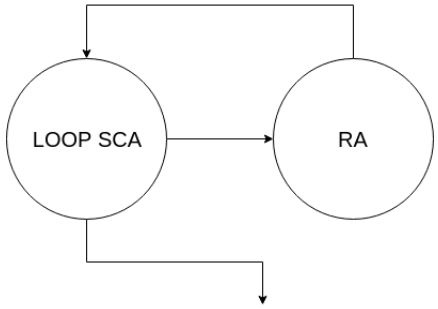

Figure 2.2: An illustration of the loop SCA.

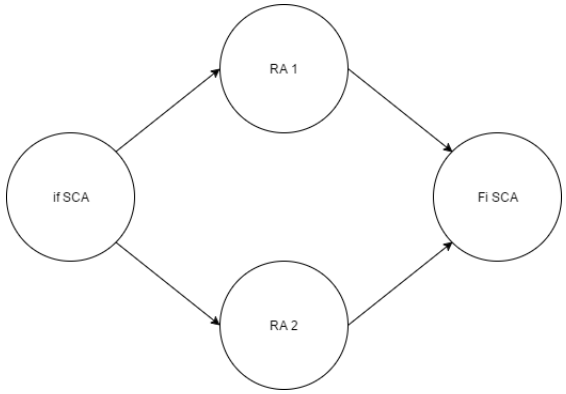

Figure 2.3: An illustration of if and fi SCAs.

An important property of DSGs is the property that given a DSG G, *at most one token* can be present in the entire DSG at any given time during execution. This property is ensured by their construction — that is, by specific rules that RAs, SCAs, and their connections must conform to. In other words, at any given time during execution of a DSG, either (a) there are no tokens on any of the FIFOs in the graph or (b) there is exactly one nonempty FIFO, and this FIFO contains exactly one token. In general, case (a) may occur *during* the firing of a DSG actor — that is, after its input has been consumed and before any corresponding output has been produced.

For more details on DSG concepts, including RAs and different types of SCAs, we refer the reader to [31].

Chapter 3: Related Work

Various prior research efforts are related to the problem of modeling of dataflow schedules. For example, Lee an Ha presented four classes of alternative scheduling strategies for real-time DSP systems [18]. These classes include, in decreasing order of flexibility, fully dynamic, static assignment, self-timed, and fully static scheduling. Ko et al. proposed a representation called the generalized schedule tree (GST) to represent a class of schedules called looped schedules [17]. The synchronization graph is a schedule representation for modeling self-timed, multiprocessor schedules for homogeneous synchronous dataflow graphs [4]. The model facilitates optimization of interprocessor communication and synchronization for this class of schedules.

As discussed in Chapter 1, our work in this thesis builds on the dataflow schedule graph (DSG) representation introduced in [31]. The DSG went beyond previously-developed schedule models in its handling of dynamic scheduling and dynamic dataflow application behavior within a unified, dataflow-based schedule representation. DSGs are capable of representing a large class of static, dynamic, and quasi-static schedules. DSGs are capable of representing both single and multiprocessor schedules. This thesis introduces novel software synthesis capabilities that help to automate the use of DSGs within design flows for model-based signal

processing.

Zebelin et al. present a unified model in which dataflow graphs and their schedules are represented together within the same model-based representation [33]. In this model, control flow associated with schedules is represented through composite actors that control execution of their encapsulated subsystems. This unified, hierarchical modeling approach is significantly different from the distinct application graph and schedule graph models that are used in the DSG model. This separation is useful to promote orthogonality in the design process. In particular, the DSG approach allows schedules and application graphs and their associated hierarchies to be designed, represented, and manipulated with strong separation of concerns. For general background on the utility of orthogonalization in system-level design, we refer the reader to [15]. Orthogonalization is an important consideration in the design of the DSPCAD Framework [21], which is a software tool that we have used to prototype the software synthesis techniques introduced in this thesis.

For prototyping of and experimentation with DSG actor implementations, we use the lightweight dataflow environment (LIDE), which is a component within the DSPCAD Framework [21, 27]. We present new extensions to LIDE that are useful in providing efficient support for RAs and SCAs. As described in Section 2.3, RAs and SCAs are the two main types of actors in DSGs. We apply the dataflow interchange format (DIF) package, which is another component of the DSPCAD Framework, to develop data structures and analysis techniques for working with DSGs, and to implement software synthesis techniques for mapping DSGs into embedded software realizations [12]. We also extend the DIF Language with features for programming

DSG-based schedules.

As part of our software synthesis framework for DSGs, we develop a clustering approach for optimizing the efficiency of the generated code. Here, by *clustering* a dataflow graph, we mean the grouping together of a subset of nodes into a "super node" that is treated as a single unit for some kind of subsequent analysis or optimization. A dataflow algorithm that applies clustering typically applies the process repeatedly to construct a series of super nodes. Clustering is a general and widely used technique in dataflow tools. Examples of other kinds of dataflow-based clustering techniques can be found in [3, 8, 9, 16].

Chapter 4: DIF-OCL

GPU-based heterogeneous computing platforms are used widely in embedded signal processing systems. Substantial performance boosts can be achieved from these platforms through their support for data- and task-level parallelism. In this chapter, we introduce DIF-OCL, which is a software tool for automated derivation of OpenCL implementations targeted to heterogeneous CPU-GPU platforms. DIF-GPU takes as input model-based specifications of signal processing dataflow graphs and schedules, and translates these into OpenCL implementations that can be deployed on heterogeneous platforms.

The proposed new design flow using DIF-OCL applies the LIDE-OCL environment, which integrates LIDE with OpenCL programming, and the DIF Language and DIF Package, which were introduced in [12].

LIDE, the LIghtweight Dataflow Environment, is a flexible, lightweight design environment that allows designers to experiment with dataflow-based approaches for design and implementation of DSP systems[21, 27]. LIDE-OCL is a plug-in of LIDE for the OpenCL language. LIDE-OCL provides application programming interfaces (APIs) and libraries for construction and interconnection of dataflow actors using the OpenCL language.

DIF-OCL provides techniques for model-based programming and software synthesis that are complementary to LIDE-OCL. In particular, DIF-OCL automates the derivation of OpenCL implementations from dataflow graphs that are constructed using LIDE-OCL actors.

A proposed DIF-OCL-based workflow for dataflow-based development of CPU-GPU implementations is illustrated in Fig. 4.1. The DIF-OCL framework takes as input an application graph specified in the DIF Language. Each actor in the application graph is an instantiation of an actor from a library of pre-defined LIDE-OCL actors. These actors can in general be a mix of actors that are taken from the built-in actors within LIDE-OCL, and user-defined LIDE-OCL actors that are constructed using the APIs and utilities provided in LIDE-OCL.

DIF-OCL also takes as input a specification of a schedule for the given application graph. This schedule is specified using the DIF Language in terms of DSG semantics. The DIF parser is used to construct intermediate representations, based on data structures within the DIF Package, for the specified application graph and schedule graph. These representations are jointly processed in DIF-OCL to produce as output an OpenCL implementation of the given application graph together with the given schedule. Run-time support for executing the generated schedule efficiently in terms of DSG semantics is also generated as part of the software synthesis process. The generated OpenCL code can be compiled onto specific target platforms using the platform-based compilers associated with those platforms, as illustrated in Fig. 4.1.

An example of the DIF Language representation for a DSG is shown in Fig. 4.2

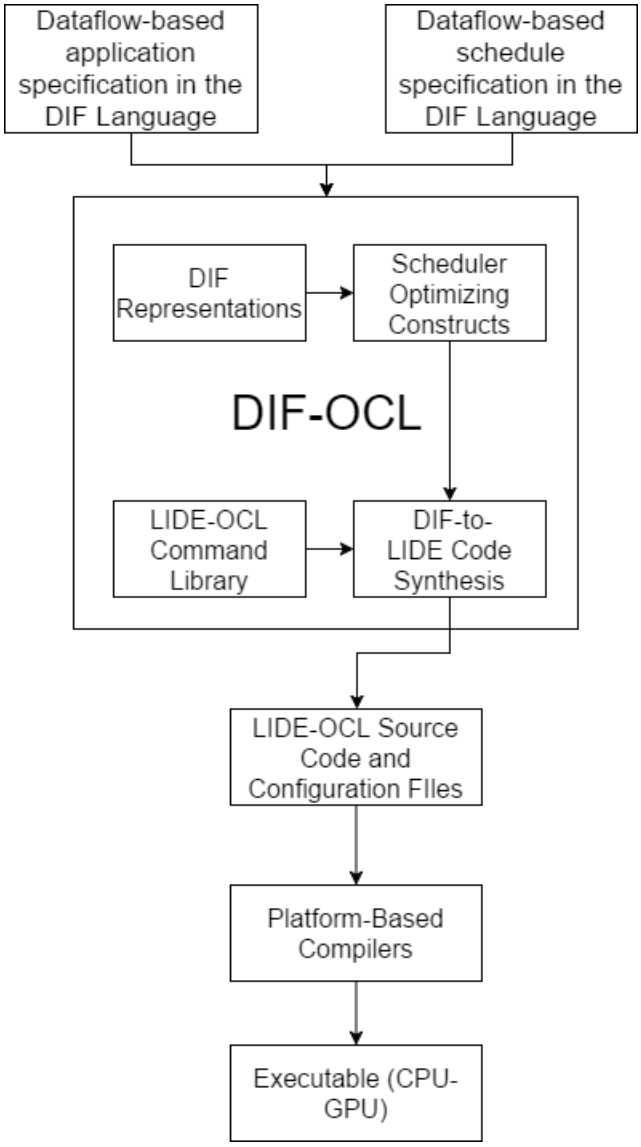

Figure 4.1: Workflow based on DIF-OCL.

```
sdf dsg_sch1 {
  topology {
    nodes = Rx,Ry,Sxy,Rm,Sip,Rip,Rout,startLoop;
    edges = e1 (startLoop,Rm),
            e2 (Rm, Sxy),
            e3 (Sxy, Rx),
            e4 (Rx,Ry),
            e5 (Ry, Sxy),
            e6 (Sxy, Sip),
            e7 (Sip, Rip),
            e8 (Rip, Sip),
            e9 (Sip,Rout),
            e10 (Rout, startLoop);
  }
```

Figure 4.2: An illustration of DIF Language code for a DSG: graph topology definition.

and Fig. 4.3. This example illustrates the first-version syntax for specifying various useful DSG-related features in the DIF Language. As this thesis represents a preliminary version of DSG software synthesis and the DIF-OCL tool, this syntax is currently evolving and being refined.

As mentioned above, certain DSG-specific details must be included in DIF Language specifications of DSGs. These details include the referenced actor associated with each RA; iteration counts associated with loop-related SCAs; and the names of the pre and post functions that are associated with the RAs. Note that the DIF Language specification of an RA does not necessarily need to have a pre function or post function. Specification of these functions is therefore optional in the DIF file for an RA. In the absence of a pre or post specification in the DIF file, it is assumed that the associated RA does not have such a function associated with it or equivalently, that the function exists, but simply does nothing (a "no-operation" function).

```
actor startNode {
  name = "sca_sch_loop";
  iterNum = "1";
  port_0 : INPUT = e10;
  port_1 : OUTPUT = e1;
}
actor Rm {
  name = "ref_actor";
  ref = "m";
  post_ref="readM";
  port_0 : INPUT = e1;
  port_1 : OUTPUT = e2;
}
actor Sxy {
  name = "sca_dynamic_loop";
  port_0 : INPUT = e2;
  port_1 : OUTPUT = e3;
  port_2 : INPUT = e5;
  port_3 : OUTPUT = e6;
}
actor Rx {
  name = "ref_actor";
  ref = "x";
  port_0 : INPUT = e3;
  port_1 : OUTPUT = e4;
}
actor Ry {
  name = "ref_actor";
  ref = "y";
  port_0 : INPUT = e4;
  port_1 : OUTPUT = e5;
}
actor Sip {
  name = "sca_static_loop";
  iterNum = "2";
  port_0 : INPUT = e6;
  port_1 : OUTPUT = e7;
  port_2 : INPUT = e8;
  port_3 : OUTPUT = e9;
}
actor Rip {
  name = "ref_actor";
  port_0 : INPUT = e7;
  port_1 : OUTPUT = e8;
}
actor Rout {
  name = "ref_actor";
  ref = "out";
  port_0 : INPUT = e9;
  port_1 : OUTPUT = e10;
}
}
```

Figure 4.3: An illustration of DIF Language code for a DSG: actor definitions.

4.1 Internal Representations for Dataflow Constructs in DIF-OCL

As described previously, application graphs and schedule graphs specified using the DIF Language are converted into internal representations within the DIF-OCL tool. These intermediate representations are utilized for synthesis and optimization of the generated software. Principal aspects of the information stored within these intermediate representations include:

- Graphical connectivity between actors and edges in the application graph and schedule graph.
- Relevant details about actors in both graphs.
- Relevant details about edges in both graphs.

For example, application graph actors within the DIF-based intermediate representation include the following fields:

- Instance name: the name of each specific actor instance.
- Type name: the actor type from which the instance is derived.
- GPU-enabled: a Boolean indicator about whether or not an actor can be executed on a GPU within the target platform.
- Details about input and output ports.

Examples of fields for schedule graph actors include:

- Instance name.
- Type name (e.g., to identify what kind of SCA each SCA instance is associated with).
- Referenced actor (for RAs).
- Details about input and output ports.

We presently use the same internal edge representation in DIF to represent LIDE edges that are contained in both application graphs and schedule graphs. Most fields associated with edges are similar between the two types of graphs. One major difference is that DSG edges have unit buffer size, while application graph edges can have any positive integer size. However, this difference is reflected primarily in the generated LIDE-OCL code, and does not play a major role in the intermediate representation management and analysis that is presently done within DIF-OCL.

4.2 DIF-OCL Scheduler Optimizations

To help schedule actors in DSG graphs efficiently, we incorporate a number of optimizations within DIF-OCL. Here, we provide a brief introduction to these optimizations as they relate to the architecture of DIF-OCL. We discuss the optimizations themselves in more detail in Chapter 6.

Specifically, we have developed two scheduler optimizations in DIF-OCL, which we refer to as *RA region clustering* and *loop SCA bypass*. Intuitively, RA region clustering exploits deterministic control flow within connected regions (or chains) of RAs

to eliminate certain forms of run-time scheduling overhead when such regions are executed. The loop SCA bypass optimization is used to streamline the scheduling of selected constructs for iterative scheduling in DSGs. Both of these operations operate on a common data structure, which we refer to as a *collection of clusters*, within the DIF-OCL tool. This data structure is related to the concept of clustering, which was briefly introduced in Chapter 3.

Fig. 4.4 provides an illustration of the collection of clusters data structure within DIF-OCL. The clusters managed by these data structures are all in the form of *directed chains* of RAs. By a directed chain in this context, we mean a directed graph consisting of a sequence of actors $(x_1, x_2, \ldots, x_N)$, where x_1 has no input edges within the cluster; x_N has no output edges within the cluster; for $i = 1, 2, \ldots, (N-1)$, the cluster contains the edges (x_i, x_{i+1}); and the cluster contains no other edges apart from these $(N-1)$ edges between successive elements of the actor sequence.

In this data structure, each cluster is stored as an array and the collection of all of the managed clusters is stored as an array list. Each element of the array list stores a distinct cluster. The data structure is implemented as an extension to the DIF Package.

4.3 Code Generation

In the previous sections of this chapter, we discussed how application graphs and schedule graphs are specified by the DIF-OCL user, and then how they are

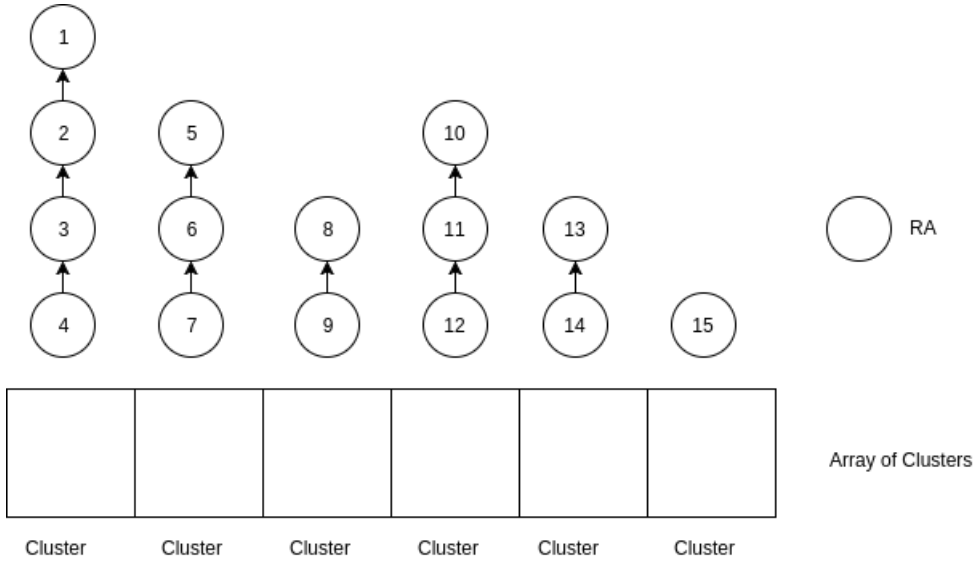

Figure 4.4: An illustration of the collection of clusters data structure in DIF-OCL.

represented internally inside the DIF-OCL framework. We also provided an overview of data structures that are applied in a number of scheduler optimizations. The internal representations of the application and schedule graphs within DIF-OCL are also useful in synthesizing software for the targeted implementation. In DIF-OCL, the synthesized software is in the form of an OpenCL program that integrates the application graph functionality together with the given schedule graph formulation for how execution of the application graph is to be coordinated.

The DIF-OCL framework generates well-structured, human readable code in LIDE-OCL format. DIF-OCL divides the generated code into different modules for the application and schedule graphs, which enhances the modularity of the derived implementation. For example, a common application graph implementation can be integrated efficiently with different schedule implementations to experiment with

alternative schedules for the application.

The OCL package synthesized by DIF-OCL includes two `.c` files — one for the application graph and the other for the schedule graph. A third `.c` file that contains scheduling functionality is also synthesized. Three header files (`.h` files) are also synthesized in correspondence with the synthesized application graph, DSG, and scheduler implementations. The synthesized application graph and DSG are constructed as abstract data types (ADTs). These ADTs include methods used for construction and termination of the graphs.

DIF-OCL also provides a Bash script called `dloclconfig` to aid in configuring the compilation of the generated code. This script is not synthesized by DIF-OCL; instead, it is provided as a standard template that the user can configure (through appropriate adjustment of right-hand-side parts in assignment statements). The `dloclconfig` file is structured based on cross-platform project organization conventions that are used in the DSPCAD Integrative Command Line Environment (DICE) [5], which is a software package that facilitates design and implementation of embedded signal processing systems. If one prefers to employ a different process to compile the synthesized OpenCL software (instead of a process that uses DICE with the generated `dloclconfig` file), such a user-defined process can be incorporated to build the generated package — there is no dependency in the synthesized software on any particular build process.

In the generated scheduler code (the third synthesized software file described above), DIF-OCL includes the code for alternative schedulers — that is, alternative methods for interpreting the synthesized DSG implementation. Presently, the set

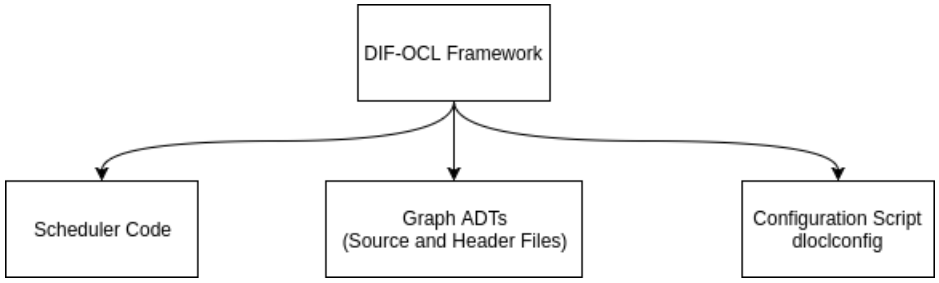

Figure 4.5: An illustration of the different components of synthesized code that are generated from DIF-OCL.

of generated schedulers includes those that are based on inclusion or exclusion of the clustering and loop bypass optimizations introduced in Section 4.2. The set of generated schedulers can easily be extended as more scheduler techniques are incorporated into DIF-OCL. The generation of code for alternative scheduler techniques facilitates experimentation with the techniques to help understand their trade-offs, and to select the most suitable one for a given application.

Fig. 4.5 provides an illustration of the different components of synthesized code that are generated from DIF-OCL.

Chapter 5: DSG Implementation in LIDE-OCL

In this section, we discuss the software implementation of various DSG constructs in LIDE-OCL. Recall from Chapter 4 that LIDE-OCL provides application programming interfaces (APIs) and libraries for implementing and integrating dataflow actors and edges using the OpenCL language, and DIF-OCL synthesizes software that utilizes the APIs and libraries provided by LIDE-OCL. Additionally, individual actors that are utilized in a DIF-OCL application graph are assumed to have corresponding LIDE-OCL implementations for their respective bodies of internal functionality.

A DSG (schedule graph) is specified as input to DIF-OCL using the DIF Language (i.e., as a `.dif` file). This DIF Language input is processed by the DIF parser and converted into an internal representation within DIF-OCL. During software synthesis, this intermediate representation is in turn converted into OpenCL code that implements the DSG based on an abstract data type (ADT) called the *DSG ADT* in LIDE-OCL. In the next section, we provide further discussion about the DSG ADT.

5.1 DSG ADT in LIDE-OCL

The lightweight dataflow (LWDF) DSG abstract data type (ADT) is a set of LWDF APIs that allows us to construct, terminate, and perform various operations using a DSG. For brevity, we refer to this ADT as simply the *DSG ADT*. The LWDF-based design of these APIs facilitates their retargetability across different implementation languages, and its integration with the LIDE package. For background on LWDF and LIDE, we refer the reader to [27, 28].

DIF-OCL currently provides an OpenCL-based implementation of the DSG ADT. This OpenCL-based implementation is compatible for use with LIDE-OCL. In the remainder of this thesis, when we discuss the DSG ADT, we refer specifically to our OpenCL-based implementation of the ADT, unless otherwise specified.

An instance of the DSG ADT represents a specific schedule model, and is implemented in a header (`.h`) file and C (`.c`) file. The header file contains definitions of unique integer constants for the graph elements (actors and edges). These constants can be used to index into tables associated with graph elements in the ADT. Such indexing allows a program to access specific graph elements. In other words, each actor (edge) is associated with a unique index that determines its place in a common table or array of actors (edges) that is included in a given DSG ADT instance.

The method for constructing a DSG comprises of four main parts:

- Variable initializations.

- Edge and actor definitions.
- Construction of a lookup table for determining the type of each actor.
- Construction of lookup tables for determining the source actor and sink actor associated with each edge.

As described in Chapter 2, there are two types of actors in the DSG model — reference actors (RAs) and schedule control actors (SCAs). In our implementations of RAs and SCAs in LIDE, we incorporate a number of additional features, which are not fundamental to the DSG model, but facilitate the construction and manipulation of efficient DSG implementations. For example, we incorporate an account of the last FIFO that is referenced by a DSG actor, which is useful in tracking the flow of control between actors as a DSG executes.

In Section 5.2 and Section 5.3 we discuss the implementation of RAs and SCAs in LIDE-OCL.

5.2 RA Implementation

As discussed in Section 2.3, RAs can be viewed as wrappers for firing specific application graph actors. In general, firing of an RA A in a DSG G corresponds to a guarded firing of the referenced actor $ref(A)$ in the associated application graph.

More specifically, the following steps are involved in the execution of an RA.

- Preprocessing: As described in Section 2.3, the preprocessing stage of an RA firing is carried out by the optional pre function that may be associated with

the RA. If the pre function is not defined for an RA then its pre processing stage is skipped. A pre function, when it is defined, is provided as a function pointer argument to the function that initializes an RA.

- Referenced actor invocation: When an RA is initialized, it is associated with a unique application graph actor as the referenced actor of the RA. A guarded invocation of the referenced actor is carried out in this core step of the RA execution process. The guard in this context corresponds to the result of the lightweight dataflow `enable` function for the associated actor (e.g., see [28]). Evaluation of the guard can be bypassed if it is known by some form of static or dynamic analysis that it will always be `true`. For example, when executing DSG-based schedule models of static schedules for synchronous dataflow (SDF) graphs, it is possible to bypass guard checking in RAs if appropriate methods are used in the construction of the underlying schedules.

- Post processing: In a manner similar to the preprocessing stage, the post-processing stage of an RA firing is carried out by the optional post function that may be associated with the RA. The postprocessing stage is skipped if no post function is associated with the RA. As with the pre function, when a post function is defined, it is provided as a function pointer argument in the initialization of the corresponding RA. One possible use of a post function is in the derivation of a value that is to be encapsulated in the DSG token that is produced on the output of the RA. If no such value is provided by the post function, then the DSG token that is produced can be viewed as having a null

value associated with it. In this case, the token helps to direct the flow of control through the DSG even though its value is not relevant.

5.3 SCA Actor Implementation

Schedule control actors (SCAs) in DSGs provide a general mechanism to direct the flow of DSG tokens in DSGs [31]. Since DSG tokens serve to enable actors that receive these tokens at their inputs, the direction of such tokens can be viewed as influencing the order in which actors within a DSG are executed. Currently, four specific SCA types are provided in DIF-OCL. This library of SCAs can be extended in future work to include other SCAs that are based on the general "SCA design rules" defined in [31].

The four types of SCAs that are currently available in DIF-OCL are summarized as follows.

- Loop SCAs. There are two types of loop SCAs in DIF-OCL — static and dynamic loop SCAs. Loop SCAs are used to direct DSG tokens through a specific output port for some number of successive iterations before the next DSG token output by the actor is directed through a second output port. Tokens on the first output port can be viewed as enabling the body of a loop, while the second output port can be viewed as being associated with the part of the DSG that follows the loop. For further details on this type of SCA, we refer the reader to [31].

  A static loop SCA is provided an iteration count as a static (compile time)

parameter. On the other hand, a dynamic loop SCA receives iteration counts from DSG tokens that it consumes on one of its input ports. Thus, the iteration counts of dynamic loop SCAs can vary at run-time, based on manipulations to the values of the DSG tokens that are provided to them.

- Conditional SCAs. There are two types of conditional SCAs in DIF-OCL — the IF SCA and the FI SCA. Intuitively, an IF SCA is used to route its output DSG token to one of two output ports, thereby enabling one of two different DSG actors for subsequent execution in the schedule graph. Conversely, the FI SCA consumes an input DSG token from one of two input ports, and produces output on a single output port. Thus, a common next actor in the DSG will be enabled regardless of which input the input DSG token came from. For further details on the semantics of the IF and FI SCAs, we again refer the reader to [31].

5.4 DSG Edges in LIDE-OCL

Recall from Chapter 1 that we are focused in this thesis on *sequential DSGs*, and when we write "DSG" in this thesis we are implicitly referring to a sequential DSG. An important property of sequential DSGs is that they contain *at most one token* at any given time. We refer to this as the *global token population* property of sequential DSGs. That is, given a sequential DSG G, the sum of the number of tokens across all of the FIFOs in G is at most equal to unity at any point during execution of G. The sum will be zero during times when the currently executing

DSG actor has consumed a DSG input token but has not yet produced a token during the same firing.

The DSG edges in LIDE-OCL are optimized to exploit the global token population property. In particular, each DSG edge in LIDE-OCL is implemented as a FIFO with unit size. Furthermore, we apply a streamlined FIFO ADT implementation from LIDE that is designed specifically for the case where the buffer size is 1.

We envision that further optimization is possible to provide a single FIFO that is shared across all of the edges in a DSG (rather than having separate unit-size FIFOs for each edge). Such further optimization is a useful direction for future work.

5.5 Execution of DSG graphs

As discussed previously, DIF-OCL provides capabilities to synthesize an OpenCL implementation of a DSG that is specified in the DIF Language. We refer to this synthesized OpenCL implementation as the "synthesized implementation" of the associated DSG specification. The synthesized implementation of a DSG is encapsulated within an instance of the LIDE-OCL DSG ADT, as described in Section 5.1.

To execute its associated DSG, a *DSG scheduler* can be applied to a DSG ADT instance. By a DSG scheduler, we mean a software module that takes as input a DSG, and executes the actors in a manner that preserves DSG semantics — e.g., by firing DSG actors only when they become enabled by the presence of DSG

tokens at their inputs. Since a DSG always has at most one DSG token during its execution, there is always a unique actor that is to be fired next, once the current firing completes. A DSG scheduler provides a mechanism to carry out this and other aspects of DSG semantics. When the "DSG" qualification is understood from context, we may sometimes write "scheduler" in place of "DSG scheduler".

The optimization of DSG schedulers is itself an interesting topic. We present a small set of optimized scheduler designs as a starting point in investigating this topic. These optimized schedulers are presented in Chapter 6. We envision that further exploration of optimized DSG scheduler design is an interesting topic for future work.

Since DSGs are themselves dataflow graphs, they can be executed by any dataflow scheduler that supports the variant of core functional dataflow (CFDF) semantics that DSGs adhere to. Thus, a scheduler can be used as a DSG scheduler even if it is not specialized to the semantics of DSGs. For details on the relationship between DSGs and CFDF semantics, we refer the reader to [31], and for details on the CFDF model of computation, we refer the reader to [25].

A general scheduler that we apply in our experiments, in addition to the optimized DSG-specialized schedulers described above, is an adaptation of the *simple scheduler* that is provided in LIDE [27]. The simple scheduler in LIDE applies a form of scheduling called *canonical scheduling* of CFDF graphs [26]. Canonical scheduling can be viewed as a kind of round-robin scheduling for CFDF graphs. This scheduling technique and the LIDE simple scheduler can be adapted easily to handle DSGs even though DSGs deviate from CFDF semantics in some respects. Advantages of

the simple scheduler include its simplicity and generality, which make it useful, for example, during functional validation and rapid prototyping. Its main drawback is that it can be highly inefficient. Thus, an investigation of efficient scheduling techniques should typically not be limited to use of the simple scheduler.

In the remainder of this thesis, when we write "simple scheduler", we mean (unless otherwise stated) the adapted version of the LIDE simple scheduler that we use in DIF-OCL.

5.6 Limitations

The current implementation of RAs in DIF-OCL gives the RA programmer access to the entire *context* associated with the associated referenced actor. The actor context is the standard data structure used in LIDE to encapsulate actor-specific information, including input FIFOs, output FIFOs, parameters, and state variables [28]. In this sense, the form of RA implemented in DIF-OCL is somewhat more general than the original definition of RAs in the DSG model of Wu [31].

In Wu's DSG model, RAs are restricted to manipulating only certain kinds of data. For example, the pre function can only access the value represented by the current DSG token, the state of the enclosing RA, and the state of the referenced actor. However, the RA programmer in DIF-OCL has access to all of the information in the actor context, as described above. In DIF-OCL, it is the responsibility of the RA programmer to ensure that the information in the actor context is used within pre and post functions in a way that does not lead to non-deterministic behavior.

Study of design rules or systematically-imposed restrictions on this more general class of RA implementations is a useful direction for future work.

Chapter 6: Optimizations for DSG Scheduler Design

In Chapter 5, we discussed the role of DSG schedulers in the execution of DSGs, and we discussed the simple scheduler as one DSG scheduler that we have applied in our experiments. When applied to a DSG, the simple scheduler operates by traversing the list of DSG actors at each scheduling step, and performing a guarded execution of each actor that it visits. That is, as each actor is visited during the traversal process, the enable function of the actor is first executed. If the enable function returns true, then the invoke function of the actor is executed, the current scheduling step is completed, and the scheduler moves to its next scheduling step.

This kind of "brute force" execution process can be very inefficient — for example, due to excessive calls to actor enable functions. In this chapter, we develop more efficient techniques for coordinating the execution of actors in DSGs. These techniques exploit specific characteristics of DSGs to streamline the run-time interpretation of DSGs.

Further improvements in efficiency may be possible through methods that generate code for directly executing DSGs. Such methods, for example, might have code for SCAs generated that is followed by appropriate branching code, where

this branching code is determined based on the type of SCA being used, and the particular output port on which the DSG token is produced during a given firing. Investigation of such *direct synthesis* methods for DSGs is a useful direction for future work.

Our interpreted approach for DSG execution has certain advantages, as well. These advantages relate to similar differences between interpreted and compiled execution of computer programs. When DSGs are interpreted, only the DSG itself needs to be synthesized, while the logic for executing the DSG can be re-used through pre-defined runtime library components, or through a separately provided scheduler. Similarly, interpreted DSGs facilitate efficient just-in-time scheduling techniques where DSGs may be constructed, adapted, or input at run-time, and then executed immediately after they become available. Recent work on just-in-time scheduling of Parameterized and Interfaced Synchronous DataFlow (PiSDF) graphs has shown promising results [11]. An interesting direction for future work is the study of just-time-time techniques for dataflow graphs that utilize the concepts of DSGs, DSG schedulers, and interpreted DSG execution.

6.1 DSG Token Tracking

The first approach that we present for DSG scheduler optimization is an approach that we refer to as *DSG token tracking* (*DTT*). DTT exploits the global token population property of DSGs that we discussed in Section 5.4. Recall that this property ensures that at any given time during execution of a properly con-

repeat
    Invoke the current DSG actor C
    Record the index I of the edge on which C has produced its output token
    $C = sinks[I]$
until $terminate() = true$

Figure 6.1: A pseudocode representation the DTT approach for DSG scheduler design.

structed DSG, there is at most one DSG token in the graph. Thus, once the firing of a DSG actor completes, we need only to determine the edge e on which the firing produces its output DSG token. The sink actor of this edge e is the unique actor that needs to be executed as the next firing in the interpretation of the enclosing DSG.

DTT utilizes these concepts, and is based on efficiently tracking DSG tokens as they are produced during interpretation of a DSG. Here, by "tracking", we simply mean determining the edge on which the associated token resides.

The DTT approach can be summarized by the pseudocode-style representation shown in Figure 6.1. Here, each iteration of the loop executes a single scheduling step, where each scheduling step is responsible for carrying out a single firing of a DSG actor and determining the next DSG actor that is to be fired. It is assumed that the "current DSG actor C" is initialized to be the first actor that is to be executed when interpretation of the DSG is launched. Similarly, it is assumed that there is some termination condition $terminate()$ that can be checked to determine when execution of the DSG is complete. Alternatively, the scheduler can be enclosed within an infinite loop that is executed indefinitely and may be terminated asynchronously through some kind of interrupt mechanism.

To support our implementation of the DTT approach, as represented in Figure 6.1, we associate (in the synthesized code) a unique index in the range $\{0, 1, \ldots, (N-1)\}$ for each DSG edge, where N is the total number of DSG edges. We also synthesize a table (array) $sinks[I]$ that maps these "edge indices" into unique indices that are associated with the DSG actors. Specifically, $sinks[I]$ gives the index of the sink actor that is associated with the DSG edge whose index is I.

6.2 RA Region Clustering

RA region clustering, which we briefly introduced in Section 4.2, exploits deterministic control flow within directed chains of RAs. This determinism is exploited to eliminate overhead associated with tracking of DSG tokens within the chains, including the costs of table lookups to the *sinks* array. RA region clustering effectively transforms a directed chain of RAs into a single, *hierarchical RA* that can be treated as a single unit when the DSG is interpreted by a scheduler, such as the DTT scheduler illustrated in Figure 6.1.

RA region clustering is related in some ways to techniques developed by Gu et al. for managing statically schedulable regions in dataflow programs based on the CAL language [10]. One significant difference is that RA region clustering is defined in the context of schedule graphs rather than dataflow-based of application representations. RA region clustering is also related to the problem of constructing basic blocks in conventional compiler analysis [1, 2]. RA regions in our context are different from basic blocks in that they are defined in the context of pure dataflow

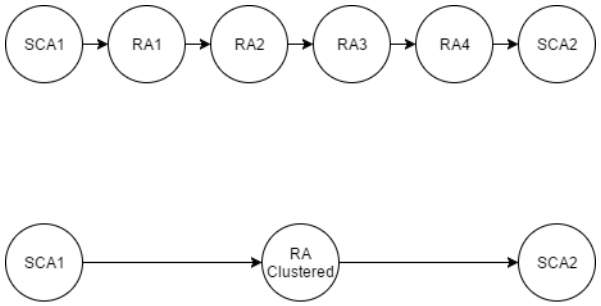

Figure 6.2: A simple illustration of RA region clustering.

representations (e.g., as opposed to hybrid control/dataflow representations); they are defined in the concept of schedule graphs as opposed to program representations (similar to the difference with the CAL-based analysis of Gu); and their basic elements (RAs) encapsulate execution of dataflow actors of arbitrary complexity as opposed to the primitive program statements that form the elements of conventional basic blocks.

RA region clustering is a general technique that can be used to improve the performance of a variety of DSG schedulers, including the simple scheduler introduced in Section 5.5 and the DTT scheduler discussed in Section 6.1.

Fig. 6.2 provides a simple illustration of RA region clustering. This is a simplified diagram in which only one port is shown for each of the SCAs in the diagram — the other ports are omitted since no specific type of SCA is assumed in the illustration.

Our algorithm for RA region clustering, which we have implemented as part of DIF-OCL, is sketched in the pseudocode shown in Fig. 6.3. The algorithm can

```
startNode denotes the starting node.
s = newStack()
s.push(startNode)
clustered = []
while s.length != 0 do
    curr_node = s.pop()
    mark curr_node
    tempCluster = []
    if curr_node.type == ref_actor then
        tempCluster.add(curr_node)
        if curr_node has a successor whose type is not ref_actor then
            if tempCluster.size() > 1 then
                clustered.add(tempCluster)
                tempCluster = []
    for each node N that is a successor of curr_node do
        if N is not marked then
            s.push(N)
```

Figure 6.3: A pseudocode representation of the RA region clustering optimization for DSG scheduler design.

be viewed as a form of depth first search that aggregates maximal directed chains of RAs into individual clusters (hierarchical RAs). Here, we say that given a directed graph G and two vertices (nodes) x and y in G, y is a *successor* of x if the directed edge (x, y) is in G.

6.3 Loop SCA Bypass Optimization

A general class of optimizations when synthesizing software or designing schedulers for DSGs is the class of *SCA-specific optimizations*. By an SCA-specific optimization, we mean an optimization that exploits the specialized characteristics of a specific type of SCA or a specific subset of SCA types. SCA-specific optimizations enable use of the formal DSG-based model for schedule management, while improving the efficiency of interpreting or generating code for this class of schedules.

In this thesis, we introduce one optimization technique, called *loop SCA bypass*, that provides an example of an SCA-specific optimization. In the loop SCA bypass technique, we implement streamlined scheduling of RA chains that are controlled by loop SCAs, and bypass the treatment of those loop SCAs as general DSG actors.

We have developed loop SCA bypass optimization in a restricted form in which the optimization can be applied only to a loop SCA that is connected to an RA chain or to a single hierarchical RA that represents an RA region. Thus, for example, the technique that we have developed cannot be applied to nested loops. Generalizing the loop SCA bypass optimization to work with more general uses of SCAs is a useful direction for future work.

A simple illustration of the loop SCA bypass optimization is shown in Fig. 6.4. In this example, Actor 1 is a loop SCA, Actor 2 is an RA, and edges incident to Actor 1 and Actor 2 that are not of direct relevance to this example are omitted.

Now suppose that based on the iteration count received by Actor 1, we need to execute 5 firings of Actor 2. Using the standard interpretation approach for loop SCAs, this would involve 10 firings of DSG actors — 5 firings each of Actor 1 and Actor 2. On the other hand, by using the loop SCA bypass optimization, we need one firing of Actor 1 (at the entry to the loop), followed by 5 firings of Actor 2, and then one more firing of Actor 1 (at the exit of the loop). As the iteration count associated with a loop SCA increases, the performance enhancement due to the loop SCA bypass optimization becomes more substantial.

In the loop SCA bypass optimization, we keep track of whether the previous DSG actor invoked is a loop SCA L that is connected to an RA chain X. If so,

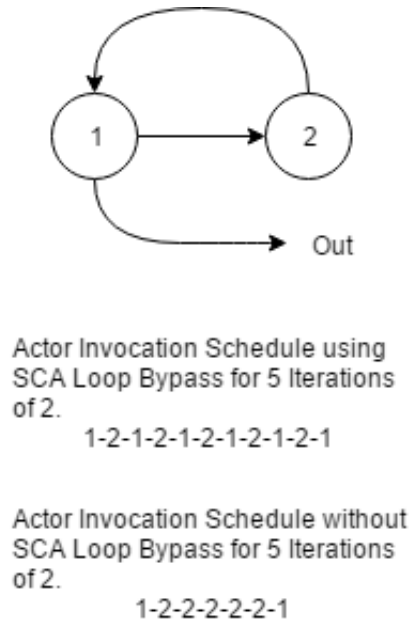

Figure 6.4: An illustration of the loop SCA bypass optimization.

then we iterate directly through the required iterations of X in the generated code or schedule interpreter rather than using DSG constructs to manage the iteration control. Once the required number of iterations of X is complete, the iteration count of L is set to 0. This in turn forces the DSG actor following L (outside of its associated loop) to be invoked using the standard process for executing the DSG.

6.4 Limitations

In our current implementation of DIF-OCL, RA region clustering and loop SCA bypass are incorporated into customized DSG schedulers that are synthesized as part of the software synthesis process. Thus, the optimizations are not applied in the form of general DSG interpreters that can be used to execute arbitrary DSGs

without need for synthesis of the scheduler functionality.

This software synthesis approach can be viewed as supporting a kind of hybrid between interpreted and compiled execution of DSGs. In particular, a standard interpretation-oriented DSG scheduler structure, based on the DTT technique, is used as a scheduler template. During software synthesis, the RA region clustering and loop SCA bypass techniques are integrated into the template in an application-specific (DSG-specific) manner. The synthesized scheduler is thus specific to the input DSG; however, the prototyping of the optimizations techniques is simplified because they are applied in the context of a compact scheduler template.

It is with this rapid prototyping consideration in mind that we have used this particular synthesis approach to experiment with our first-version designs of RA region clustering and loop SCA bypass. Integrating these optimizations into a general DSG interpretation framework is a useful direction for future work.

Chapter 7: Experiments

To demonstrate the utility of DIF-OCL on a practical application of significant complexity, we experiment with the jitter measurement application developed in [22]. This is a computationally intensive application that is designed for measurement of jitter in digital communication waveforms.

Our experiments are carried out on a hybrid CPU-GPU platform that includes an Intel Core i7-2600K Quad-core CPU with an NVIDIA GeForce GTX680 GPU running Ubuntu Linux 16.04.2 LTS.

The input to the application is a series of data points that represent successive samples of a communications waveform. The input is presented to the application in a text file. We specify configurable application parameters, including the window size W and number of iterations Z, in a separate file that is input to the application. For details on the meaning of these parameters, we refer the reader to [22]. In our experiments, we use $W = 13{,}1072$ and $Z = 12$. The output from the application is the derived clock period and time interval error (TIE), which are saved by the application in two text files, respectively.

We use the application graph as specified in [22] with some minor adaptations. The modified version that we use is illustrated in Fig. 7.1. This application graph is

in the form of a computation graph [14]. The computation graph model is similar to the SDF model in that actors are characterized by statically known production and consumption rates. However, in computation graphs, the number of tokens on an input edge that is required to enable a firing can be greater than the consumption rate. This number of tokens required to enable a firing is referred to as the *threshold* of the associated actor port or edge. In general, the threshold is greater than or equal to the consumption rate.

In Fig. 7.1, the numbers next to the output ports represent the production rates associated with these ports. For all input ports other than the input to DVL, the threshold T equals the consumption rate C; this common value of T and C is annotated next to each of these ports. The threshold and consumption rate for the input port of DVL are specified separately because these values may differ — here, c represents the consumption rate and τ represents the threshold. For details on how (c, τ) are determined, we refer the reader to [22].

Descriptions of these actors are reproduced here with minor adaptations from the corresponding descriptions in [22]:

- DVL: Finds the upper and lower voltage thresholds after sorting the input data of the current window.
- STR: Assigns high and low voltage states after performing analog to digital conversion based on the voltage thresholds.
- FSM: Locates transitions from high to low and low to high voltage states.

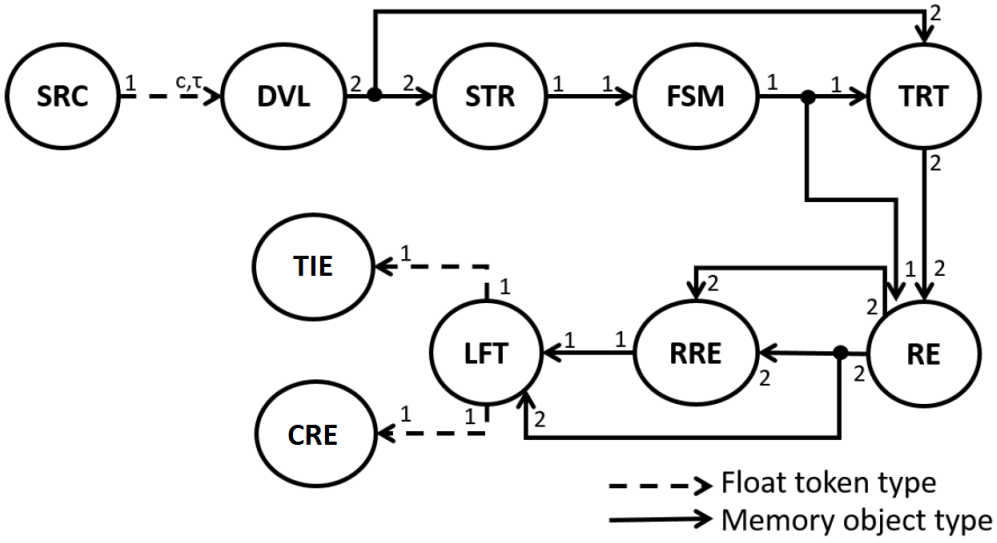

Figure 7.1: Application graph for jitter measurement system.

- TRT: Computes the transition time for each voltage transition in the current window.

- RE: Derives a preliminary estimate of the clock period.

- RRE: Improves the accuracy by refining the clock period estimate.

- LFT: Estimates the clock period using linear filtering and computes interval errors using the redefined clock period estimate.

- SRC, TIE, CRE: These are actors that interface to input/output files from which input data is injected into dataflow graph, and output data computed from the graph is stored, respectively.

Among these actors, the following ones are mapped to the GPU in our imple-

mentation: RE, RRE, DVL, TRT, LFT, FSM, STR.

We write a `.dif` file in the DIF Language for this application graph. We apply this file as input to the DIF-OCL framework as the application graph for our experiments.

As the schedule graph to be provided as input for this example, we develop a DSG model of the schedule applied in [22]. This schedule can be expressed as the sequence of executions

$$S = (W\,SRC)\,DVL\,STR\,FSM\,TRT\,RE\,RRE\,LFT\,TIE\,CRE, \tag{7.1}$$

where $(W\,SRC)$ represents the repeated execution W times of the actor SRC. Recall that W is the window size parameter for the application.

Our DSG model of this schedule is illustrated in Fig. 7.2. Here, a label of the form "Rxyz" represents an RA whose referenced actor is XYZ. For example, Rdvl is an RA whose referenced actor is DVL. There are three SCAs in the graph of Fig. 7.2: SCA1 is a static loop SCA, SCA2 is a dynamic loop SCA, and SCA3 is another static loop SCA. In SCA3, the output port corresponding to the exit from the loop is associated with code to exit the enclosing scheduler. This use of SCA3 allows the whole DSG to be executed for some pre-specified number of iterations before the DSG execution is terminated.

The "D" next to the input edge to SCA3 indicates a unit *delay* or initial token on the associated edge. A DSG needs to have exactly one delay on exactly one of the edges. The actor at the sink of this edge is the first actor that is to be fired when

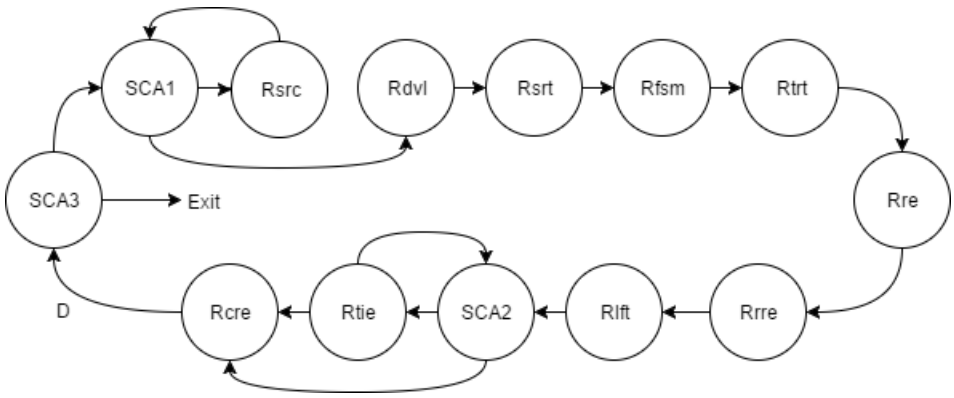

Figure 7.2: DSG model for the jitter measurement system.

the DSG is executed. In our implementation, we use a slightly different technique to determine the start actor (rather than using the unique delay in the graph). The technique is functionally equivalent to use of the delay, as illustrated in Fig. 7.2, and used in our design only for convenience of implementation.

We executed the scheduler graph with the following four DSG scheduler configurations:

- Simple Scheduler [SS]
- DSG Token Tracking scheduler [DTT]
- DSG Token Tracking scheduler with RA Region Clustering [DTTRRC]
- DSG Token Tracking scheduler with Loop SCA Bypass [DTTLSB]

Note that the DTTLSB configuration includes RA region clustering to a limited extent (see Section 6.2), so this can be viewed as the most "powerful" configuration in terms of the number of different optimizations that co-exist. In contrast, the

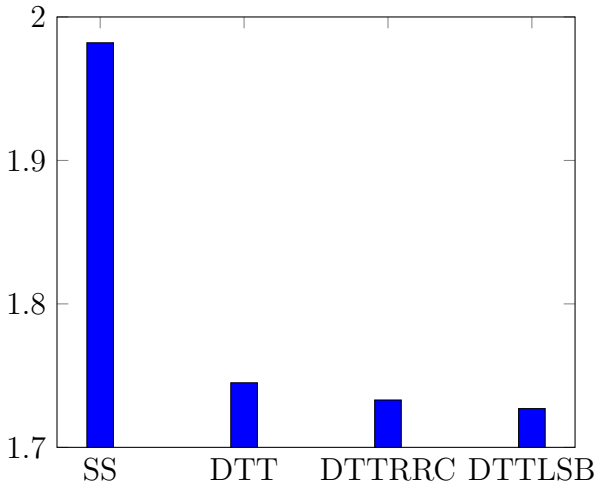

Figure 7.3: Comparison of run-times for the jitter measurement system using various DSG schedulers. The vertical axis has units of *seconds*.

DTTRRC configuration incorporates RA region clustering but does *not* incorporate the loop SCA bypass optimization.

In each case, we validated the functional correctness of the output by comparison to the output of the original application graph in [22].

The application run-times measured from our experiments are summarized in the Fig. 7.3. As we can see from these results, SS takes substantially greater time than the other three configurations, which are optimized in various ways. DTT eliminates the need to check enable conditions for DSG actors, and RA region clustering further improves performance by exploiting deterministic control flow across chains of RAs. DTTLSB can be useful in cases where we have extensive looping being performed inside a schedule. The jitter measurement system has two looping constructs that collectively involve over one million iterations. These looping constructs provide opportunities for additional improvement using DTTLSB.

The data in Figure 7.3 is summarized in tabular form in Table 7.1.

Table 7.1: Comparison in tabular form of run-times for the jitter measurement system using various DSG schedulers. The units of run-time in this table are *seconds*.

Scheduling Technique	Run-time	Percentage Speedup
SS	2.07	0
DTT	1.74	18.5
DTTRRC	1.73	19.3
DTTLSB	1.71	21.0

Chapter 8: Conclusions

In this thesis, we have developed new methods for synthesizing embedded software for dataflow-based models of signal processing applications. The novelty of our methods centers on their support for the dataflow schedule graph (DSG) as a formal model of schedules for dataflow-based application representations. In conventional dataflow-based software synthesis techniques, schedules are represented using formal representations that are very restricted in their applicability or using general representations that are constructed in ad-hoc ways, without any formal connection to dataflow. In this thesis, we develop software synthesis techniques that operate on the DSG model, which is both general in its applicability to a broad class of static and dynamic dataflow representations, and formal in its underpinnings in terms of dataflow semantics. We have demonstrated the utility of our proposed new software synthesis techniques through experiments involving a complex, practical dataflow application that performs jitter measurement on digital communication waveforms.

Many useful directions for future work emerge from the developments of this thesis. These include extension of the proposed software synthesis techniques to concurrent DSGs, adaptation to other implementation languages beyond OpenCL,

incorporation of support for a larger variety of SCAs in DIF-OCL, further streamlining of memory management to exploit the global token population property of DSGs, further exploration of optimized DSG scheduler design, development of design rules or systematically-imposed restrictions for reliable RA implementation, and deeper study of trade-offs between interpreted and compiled execution of DSGs.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1988.

[2] A. Appel. *Modern Compiler Implementation in C*. Cambridge University Press, 2004.

[3] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee. APGAN and RPMC: Complementary heuristics for translating DSP block diagrams into efficient software implementations. *Journal of Design Automation for Embedded Systems*, 2(1): 33–60, January 1997.

[4] S. S. Bhattacharyya, S. Sriram, and E. A. Lee. Optimizing synchronization in multiprocessor DSP systems. *IEEE Transactions on Signal Processing*, 45(6): 1605–1618, June 1997.

[5] S. S. Bhattacharyya, W. Plishker, C. Shen, N. Sane, and G. Zaki. The DSPCAD integrative command line environment: Introduction to DICE version 1.1. Technical Report UMIACS-TR-2011-10, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011. http://drum.lib.umd.edu/handle/1903/11422.

[6] S. S. Bhattacharyya, E. Deprettere, R. Leupers, and J. Takala, editors. *Handbook of Signal Processing Systems*. Springer, second edition, 2013. URL http://dx.doi.org/10.1007/978-1-4614-6859-2. ISBN: 978-1-4614-6858-5 (Print); 978-1-4614-6859-2 (Online).

[7] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, February 1996.

[8] J. Falk, J. Keinert, C. Haubelt, J. Teich, and S. S. Bhattacharyya. A generalized static data flow clustering algorithm for MPSoC scheduling of multimedia applications. In *Proceedings of the International Conference on Embedded Software*, pages 189–198, Atlanta, Georgia, October 2008.

[9] J. Falk, C. Zebelein, C. Haubelt, and J. Teich. A rule-based static dataflow clustering algorithm for efficient embedded software synthesis. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, 2011.

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

[11] J. Heulot, M. Pelcat, J.-F. Nezan, Y. Oliva, S. Aridhi, and S. S. Bhattacharyya. Just-in-time scheduling techniques for multicore signal processing systems. In *Proceedings of the IEEE Global Conference on Signal and Information Processing*, pages 175–179, Atlanta, Georgia, December 2014.

[12] C. Hsu, M. Ko, and S. S. Bhattacharyya. Software synthesis from the dataflow interchange format. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, pages 37–49, Dallas, Texas, September 2005.

[13] W. M. Johnston, J. Hanna, and R. J. Millar. Advances in dataflow programming languages. *ACM Computing Surveys*, 36(1):1–34, March 2004.

[14] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinacy, termination, queuing. *SIAM Journal of Applied Math*, 14(6), November 1966.

[15] K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System-level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 19, December 2000.

[16] V. Kianzad and S. S. Bhattacharyya. Efficient techniques for clustering and scheduling onto embedded multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):667–680, July 2006.

[17] M. Ko, C. Zissulescu, S. Puthenpurayil, S. S. Bhattacharyya, B. Kienhuis, and E. Deprettere. Parameterized looped schedules for compact representation of execution sequences in DSP hardware and software implementation. *IEEE Transactions on Signal Processing*, 55(6):3126–3138, June 2007.

[18] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real time DSP. In *Proceedings of the Global Telecommunications Conference*, volume 2, pages 1279–1283, 1989.

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

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

[21] S. Lin, Y. Liu, K. Lee, L. Li, W. Plishker, and S. S. Bhattacharyya. The DSPCAD framework for modeling and synthesis of signal processing systems. In S. Ha and J. Teich, editors, *Handbook of Hardware/Software Codesign*. Springer, 2017. To appear.

[22] Y. Liu, L. Barford, and S. S. Bhattacharyya. Jitter measurement on deep waveforms with constant memory. In *Proceedings of the IEEE International Instrumentation and Measurement Technology Conference*, pages 1161–1166, Taipei, Taiwan, May 2016.

[23] M. Pelcat, J. Piat, M. Wipliez, S. Aridhi, and J.-F. Nezan. An open framework for rapid prototyping of signal processing applications. *EURASIP Journal on Embedded Systems*, 2009, January 2009. Article No. 11.

[24] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck. Software synthesis for DSP using Ptolemy. *Journal of VLSI Signal Processing*, 9(1), January 1995.

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

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

[27] C. Shen, W. Plishker, H. Wu, and S. S. Bhattacharyya. A lightweight dataflow approach for design and implementation of SDR systems. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pages 640–645, Washington DC, USA, November 2010.

[28] C. Shen, W. Plishker, and S. S. Bhattacharyya. Dataflow-based design and implementation of image processing applications. In L. Guan, Y. He, and S.-Y. Kung, editors, *Multimedia Image and Video Processing*, pages 609–629. CRC Press, second edition, 2012. URL http://www.crcpress.com/product/isbn/9781439830864. Chapter 24.

[29] C. Shen, S. Wu, N. Sane, H. Wu, W. Plishker, and S. S. Bhattacharyya. Design and synthesis for multimedia systems using the targeted dataflow interchange format. *IEEE Transactions on Multimedia*, 14(3):630–640, June 2012. URL http://ieeexplore.ieee.org/Xplore/guesthome.jsp.

[30] W. Sung, J. Kim, and S. Ha. Memory efficient synthesis from dataflow graph. In *Proceedings of the International Symposium on System Synthesis*, 1998.

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

[32] H. Yviquel, A. Sanchez, P. Jaaskelainen, J. Takala, M. Raulet, and E. Casseau. Efficient software synthesis of dynamic dataflow programs. In *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, pages 4988–4992, 2014.

[33] C. Zebelein, C. Haubelt, J. Falk, and J. Teich. Model-based representation of schedules for dataflow graphs. In *Proceedings of Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*, pages 105–116, 2013.