ABSTRACT

Title of thesis: REPRESENTATION AND SCHEDULING OF SCALABLE DATAFLOW GRAPH TOPOLOGIES

Shenpei Wu, Master of Science, 2011

Thesis directed by: Professor Shuvra Bhattacharyya
Department of Electrical and Computer Engineering
University of Maryland at College Park

In dataflow-based application models, the underlying graph representations often consist of smaller sub-structures that repeat multiple times. In order to enable concise and scalable specification of digital signal processing (DSP) systems, a graphical modeling construct called "topological pattern" has been introduced in recent work [23].

In this thesis, we present new design capabilities for specifying and working with topological patterns in the dataflow interchange format (DIF) framework, which is a software tool for model-based design and implementation of signal processing systems. We also present a plug-in to the DIF framework for deriving parameterized schedules, and a code generation module for generating code that implements these schedules. A novel schedule model called the scalable schedule tree (SST) is formulated. The SST model represents an important class of parameterized schedule structures in a form that is intuitive for representation, efficient for code generation, and flexible to support powerful forms of adaptation. We demonstrate our meth-

ods for topological pattern representation, SST derivation, and associated dataflow graph code generation using a case study centered around an image registration application.

REPRESENTATION AND SCHEDULING OF SCALABLE DATAFLOW GRAPH TOPOLOGIES

by

Shenpei Wu

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
2011

Advisory Committee:
Professor Shuvra Bhattacharyya, Chair/Advisor
Professor Gang Qu
Professor Manoj Franklin

Acknowledgments

I would like to show my gratitude to those who made this thesis possible.

Foremost, I own my deepest gratitude to my advisor, Dr. Shuvra Bhattacharyya for his inspiration, enthusiasm, wise advice, and financial support. I have only been in DSPCAD group for one year, but this year has become one of the most cherishable experience in my life. None of these would have become true without his encouragement and great ideas.

It is my pleasure to thank my advisory committee members, Dr. Manoj Franklin and Dr. Gang Qu, not only for their invaluable time of serving on my thesis committee, but also for their support and guidance during my graduate study.

I also would like to express my sincere gratitude to Dr. Raj Shekhar for giving me the opportunity to pursue graduate study in University of Maryland.

My special thanks go out to my colleague Dr. Chung-Ching Shen. He has made available his support in a number of ways. He gave me sound advice, patiently helped me debugging programs and organizing paper etc. I learned a lot from his knowledge and skills. I am indebted to many of my other colleagues who created this exciting, stimulating, and fun environment of DSPCAD group. I am especially grateful to Nimish Sane, who has been crucial to my project which lead to this thesis, and Dr. William Plishker, who patiently guided me during the semester I took his class and answered detailed questions when I started to work on my project in DSPCAD group. I am also grateful to George Zaki, Hsiang-Huang Wu, Stephen Won, Inkeun Cho, Kishan Sudusinghe, Scott Kim, Ilya Chuckhman, undergraduate

student Kelly Davis, and visiting student Peter Ott for providing great environment in which to learn.

I wish to thank all my friends for support, entertainment, and caring they provided during my graduate study.

Lastly, and most importantly, I wish to thank my parents, Yong Shen and Youming Wu. It is hard to overstate my gratitude to them for standing by me all these years. To them I dedicate this thesis.

This work was funded by the US Air Force Research Laboratory (AFRL-FA87501110049), and US National Science Foundation (NSF-CNS0720596). Any opinions, findings, and conclusions or recommendations expressed in this article are those of the author and do not reflect the views of AFRL or NSF.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The behavior of many multimedia applications can be characterized by patterns of stream processing computation and modeled efficiently using dataflow models of computation. In multimedia and other signal processing intensive application domains, dataflow graph models are widely used to describe applications because of their natural correspondence to signal flow graphs, and important forms of computational structure that are exposed by such models.

A dataflow graph is a directed graph, where vertices (*actors*) represent computational functions and edges represent inter-actor communication channels that buffer data in a first-in first-out (FIFO) fashion. Dataflow actors can contain computations with arbitrary complexity as long as the interfaces of the computations conform to dataflow semantics. That is, actors produce and consume data from their input and output edges, respectively, and each actor execution (firing) depends on the availability of sufficient data from the input edges of the associated actor.

When implementing a dataflow-based multimedia application model on a target platform, scheduling plays an important role (e.g., see [2]). 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, and memory usage, scheduling often

has significant impact on implementation quality.

For dataflow models of large-scale digital signal processing (DSP) applications, the underlying graph representations 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 [23]. 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.

1.1 Contributions of this thesis

In this thesis, we present a formal design method for specifying topological patterns and deriving parameterized schedules from such patterns based on a novel schedule model called the *scalable schedule tree*. Our method ensures deterministic behavior of the system based on compile-time analysis of system behavior that may contain structured, parameterizable patterns of actors and edge instantiations. We have also developed an associated software plug-in and integrated it into the *dataflow interchange format* (*DIF*) framework and the associated cross-platform design and synthesis environment called *targeted DIF* (*TDIF*) [9, 24]. TDIF is a companion

design tool of the DIF framework that supports dynamic dataflow analysis, cross-platform actor design, and code generation on targeted platforms [24].

1.2 Outline of the thesis

The rest of the thesis is organized as follows. Chapter 2 provides summary of background on dataflow graph modeling, schedule representation, and design tool development as well as related work on parameterized scheduling. Chapter 3 presents the formalization of our proposed schedule model *scalable schedule tree*. Chapter 4 presents the integration of our design method into the DIF framework and the associated TDP software package. Chapter 5 demonstrate the contributions of this thesis using a case study of an image registration application. Lastly, conclusions and future work are discussed in Chapter 6.

Chapter 2

Background and Related Work

2.1 Background

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

2.1.1 Topological Patterns

For large-scale models of 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 [23]. 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 [23].

2.1.2 Generalized Schedule Trees

The *generalized schedule tree* (*GST*) is a compact, tree-structured graphical format that can represent a variety of dataflow graph schedules [13]. 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 [13]). In this thesis, 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. We demonstrate the utility of SSTs in the synthesis of software from topological pattern models of signal processing applications.

2.1.3 The Dataflow Interchange Format

The *Dataflow Interchange Format* (*DIF*) framework provides a standard language, i.e., *The DIF Language* (*TDL*), for specifying semantics of a broad class of dataflow model of computations for signal processing applications [9]. Forms of dataflow semantics that can be expressed using TDL include graph topologies, hierarchical design structures, dataflow-related design properties (e.g., delays, production rates, consumption rates, etc.), and actor-specific information. The associated software package in the DIF framework, called *The DIF Package* (*TDP*), provides intermediate representations for dataflow graphs that are specified by TDL, along with libraries of analysis techniques and transformations that operate on these representations. The analysis techniques can be used to enhance dataflow-based design flows based on TDL or, through generalized interchange capabilities provided by DIF, based on other dataflow environments that are interfaced to DIF (e.g., see [11, 8, 24]).

In this thesis, we demonstrate the implementation and integration into the DIF framework of 1) topological patterns for representing large-scale dataflow graphs using TDL, and 2) SST representations for modeling and manipulation of parameterized schedules based on topological patterns. Our implementation of the SST representation is integrated with the *Targeted Dataflow Interchange Format* (*TDIF*) environment for generating platform-specific code from DIF models [24].

More specifically, this thesis builds on the developments of [23, 9, 24] by introducing the SST model described above for expressing parameterized schedules

based on topological patterns. We also introduce a new syntax for TDL that provides a compact way for specifying topological patterns. Furthermore, we have developed a novel plug-in to TDP for generating code from our new schedule model, and thereby deriving platform-specific code from TDL programs that include specifications of topological patterns. Designers can use this tool for experimenting with parameterized scheduling and automated synthesis of implementations from scalable dataflow graph models.

2.2 Related Work

Parameterized schedules have been studied before (e.g., see [1, 13]), 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 [15], CSDF [3], FRDF [18], and BDF [4], for dynamically reconfiguring the behavior of dataflow actors, edges, subsystems, and graphs through dynamic reconfiguration of parameter values [1]. Quasi-static scheduling techniques

were developed for parameterized synchronous dataflow (PSDF) specifications, which is the integration of the parameterized dataflow meta-model with SDF as the base model. By *quasi-static scheduling*, we mean the derivation of schedule structures that are largely fixed at compile time, with relatively small numbers of decision points or symbolic adjustments made at run-time based on the values of relevant input data. This approach to PSDF scheduling improved the flexibility with which SDF techniques can be applied, and allowed, for example, dynamic adjustments to schedules as dataflow (token production and consumption) rates vary at run-time. 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 [7]. 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 [17]. PKPN also automates the design process through integration into the Compaan/Laura tool [25].

The operational semantics of the RPN and PKPN models can be viewed as extensions of the Kahn process network (KPN) modeling framework [10], 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 summary, our work addresses the issues of parameterized scheduling for scalable topologies, and introduces a novel schedule model that provides for intuitive representation and efficient code generation for our targeted class of parameterized schedules. Adapting the parameterized scheduling models and methods from this work to the frameworks of parameterized dataflow, RPN, and PKPN are interesting directions for further study.

Chapter 3

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 thesis. 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 2), 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 [14], parameterized dataflow [1], 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 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 [20].

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* $c_L : K \rightarrow \mathrm{Z}+$. 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.

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.

3.1 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 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 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 children_z has a positive integer size 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 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 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.

3.2 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 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.

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]
   }
}
```

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

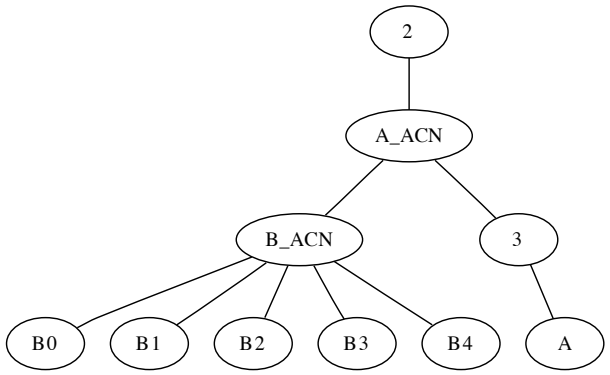

Figure 3.1: An example of an SST.

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. Exploring such generalized SSTs for more complex schedule control is an interesting direction for further study.

Figure 3.1 shows a synthetic example of a nested SST to help illustrate the SST model. In Figure 3.1, `A_ACN` and `B_ACN` are ACNs. Suppose that the evaluation results of cinit, cstep, and climit for `A_ACN` and `B_ACN` are: $\text{cinit}_A = 0$, $\text{cstep}_A = 1$, $\text{climit}_A = 1$, $\text{cinit}_B = 1$, $\text{cstep}_B = 2$, and $\text{climit}_B = 4$. Then the scheduling result from traversing the SST by following the SST traversal process is

```
S = B1 B3 A A A B1 B3 A A A.
```

This *traversed schedule* S shows the sequence of actor executions that results from traversing the given SST.

3.3 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 [21]. 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 [22, 12].

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 in conjunction with SSTs by having leaf nodes represent vectorized executions of the corresponding actors. Note that constructing an ACN with size equal to the vectorization degree N will not have an equivalent effect because control will alternate between each (scalar) actor firing and the control associated with ACN visitation instead of remaining dedicated to the actor for N consecutive firings as scalable dataflow ensures.

Vectorization (scalable dataflow) can be applied flexibly within SSTs. For example, an SST loop node L can be connected as an element of children_α, 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.

Chapter 4
Integration in the DIF Framework

In this section, we discuss our approach to integrating topological pattern modeling and SST-based schedule representation and analysis into the DIF framework and the associated TDP software package. This integration provides new capabilities for design and implementation of multimedia signal processing systems that employ repetitive graph structures.

4.1 Language Extensions

TDL in the DIF framework is a vendor-independent textual language that helps to transfer dataflow-based application models across different design tools, and also serves as a standalone language for specifying such models. TDL along with TDP captures essential dataflow modeling information and stores this information within intermediate representations, which can be analyzed and mapped into implementations on different platforms.

We have extended TDL to incorporate support for topological patterns. This extension allows topological pattern constructions to be specified as first-class citizens in the language. The parser of TDL is generated by using the SableCC compiler construction framework [6]. We have extended the TDL grammar for SableCC by defining syntactic constructs and associated parsing actions for topological pattern instantiations in TDL. Topological patterns that are currently supported by TDL and defined as *pattern keywords* in the language include `chain`, `ring`, `merge`,

`broadcast`, `parallel`, and `butterfly`. Extending TDL with additional patterns is straightforward, and such extensions will be considered in future versions of the language as additional kinds of patterns are identified as being important in the context of relevant applications.

A topological pattern is instantiated in a TDL specification with a declaration of the form:

```
<edge declaration> -> <pattern keyword>(<node list>);
```

Here, `<edge declaration>` effectively declares a set of new edges in the graph that is being constructed. These edges can be defined as scalar edges (e.g., `e1`, `e2`, ...) or in the form of an array (e.g., `e1[2]`). The placeholder `<pattern keyword>` represents a TDL keyword that specifies the kind of topological pattern that is being instantiated (e.g., chain, ring, etc.). The placeholder `<node list>` provides a list of nodes (graph placeholders for dataflow actors) that have been previously instantiated. The topological pattern instantiation construct instantiates the newly defined edges (i.e., the edges listed in `<edge declaration>`) in such a way that they connect pairs of nodes in `<node list>`) so that that the resulting combination of the nodes in `<node list>`, and edges in `<edge declaration>` form the specified type of topological pattern. The orderings of edges in `<edge declaration>` and nodes in `<node list>` are significant in determining how specific nodes and edges are linked to form a new instance of the specified pattern.

As a simple example, an instance of a `chain` pattern can be specified using the new TDL syntax as follows.

```
e[3] -> chain(n1[0:3]);
```

Here, a `chain` pattern is created by linking four nodes, `n1[0]`, `n1[1]`, `n1[2]`, and `n[3]` with the three newly instantiated edges `e[0]`, `e[1]`, and `e[2]`. Figure 4.1 shows instantiation examples for all of the patterns we have supported so far and their corresponding TDL specifications.

4.2 SST Plug-In for the DIF Package

We have implemented a new plug-in to the DIF framework that 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. This plug-in integrates the SST formulations developed in Section 3 as a new internal representation format and associated set of manipulations within the DIF framework.

This plug-in is built based on two Java classes in DIF called `ScalableScheduleTree` and `ScalableScheduleTreeNode`. An object that is instantiated from the `ScalableScheduleTreeNode` class can be in the form of either a leaf node or an internal node, where the internal node can be configured with an iteration count or specified as an ACN node. A leaf node instance is associated with an actor from the original dataflow graph. An ACN node instance has private fields that store the values of cinit_z, cstep_z, and climit_z, as defined in Section 3. The `ScalableScheduleTree` class provides methods that allow designers to construct SSTs.

The following examples illustrate how the SST that is shown in Fig. 3.1 can be constructed using the proposed tools.

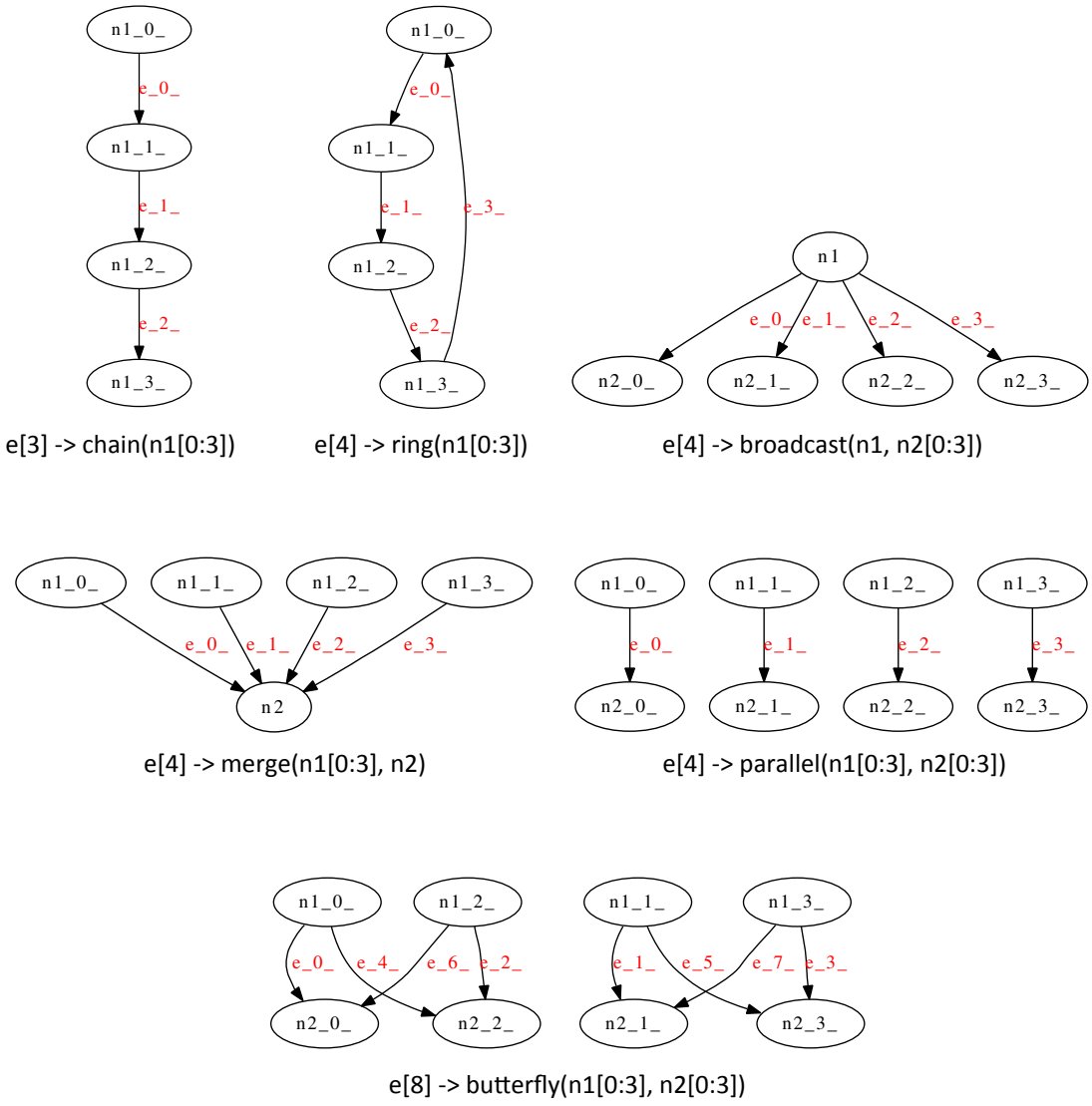

Figure 4.1: Example of topological pattern instantiations shown in terms of TDL code, and illustrations of the resulting edge instantiations together with their incident nodes.

Construction of a new SST subtree (`sst1`) is illustrated as follows. This SST is rooted at a node that is instantiated from the `ScalableScheduleTreeNode` class and configured with an iteration count value of 2. An ACN node (labeled as A_ACN) is also instantiated without any child node and added as a child node of the root node of `sst1`.

```
ScalableScheduleTree sst1 = new ScalableScheduleTree();
sst1.addACN("A", 0, 2);
```

Construction of another SST subtree (`sst2`) is illustrated as follows. This SST is rooted at an ACN node (labeled as B_ACN) that is instantiated with 5 children nodes from the `ScalableScheduleTreeNode` class, and added as a child node of the root node of `sst2`.

```
ScalableScheduleTree sst2 = new ScalableScheduleTree();
sst1.addACN("B", 5);
```

Construction of a third SST subtree (`sst3`) is shown as follows. This SST is rooted at a node that is configured with an iteration count value of 3. A leaf node is added as the child node of the root node of `sst3`.

```
ScalableScheduleTree sst3 = new ScalableScheduleTree();
sst2.addSchedule("A", 3);
```

To create the SST that is shown in Fig. 3.1, designers can insert the `sst2` subtree as the last child of the root node of the `sst1` subtree by using the method below. The same method can then be applied to insert the `sst3` subtree as the last child of the root node of the `sst1` subtree.

```
sst1.insertSchedule(sst2);
sst1.insertSchedule(sst3);
```

4.3 Code Generation and TDIF Version 0.2

In the first version of the targeted DIF (TDIF) environment, TDIF Version 0.1, designers construct schedules based on programming interfaces that are automatically generated from the TDIF tool [24]. These programming interfaces provide a consistent, formal dataflow abstraction layer between designer-constructed schedules and the actors that are executed by the schedules. Furthermore, the approach of automatically generating actor programming interfaces from target-independent actor interface specifications (in the TDIF language) allows the framework to be adapted efficiently to different target languages (presently, the environment supports both C and CUDA).

Although the designer-specified scheduling approach of TDIF 0.1 generally requires more effort compared to use of automatically generated schedules, it provides significant flexibility in terms of optimizing and fine-tuning the schedules based on specialized application constraints and objectives, and incorporating application-specific insights on schedule structure that may not exploited by available techniques for automated scheduling.

In the new version of TDIF, which we introduce here as Version 0.2, we have integrated specification and code generation support for SSTs. Thus, rather than having designers specify schedules in terms of arbitrary target-language code that connects to TDIF-generated actor interfaces (as in TDIF 0.1), we raise 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 same TDIF-

generated interfaces. SSTs are specified programmatically using graph construction APIs associated with the SST internal representation. Incorporating such specifications into TDL is a natural direction for future work that we plan to explore.

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). 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 TDIF environment, and actor code from the associated actor library to construct an executable that implements the application.

Algorithm 2 shows a pseudocode description of the SST traversal process for generating C code from the TDIF environment. Example of a generated code segment that implements a scheduler will be shown in Section 5.

Figure 4.2 illustrates the design flow of TDIF Version 0.2, which incorporates specification and code generation support for SSTs and parameterized scheduling.

Algorithm 2 Pseudocode description of the SST traversal process for code generation.

```
x is the root node of a given SST

function SSTTraversalProcess(x)
if (x is a leaf node) begin
    Generate C code for the TDIFC run-time APIs
            for the actor encapsulation.
end
else begin
    if (x is an ACN) begin
        Update the parameter settings for x.
        Evaluate cinit, cstep, and climit of x
                and store values in I, s, L , respectively.

        for (i = I; i <= L; i += s) begin
            Get y as the array children of x
            SSTTraversalProcess(y[i])
        end

    end
    else begin // x is a SST loop node
        Evaluate loop count of x.
        Generate C code for the loop structure of x
        for each child node z of x
            SSTTraversalProcess(z)
    end
end function
```

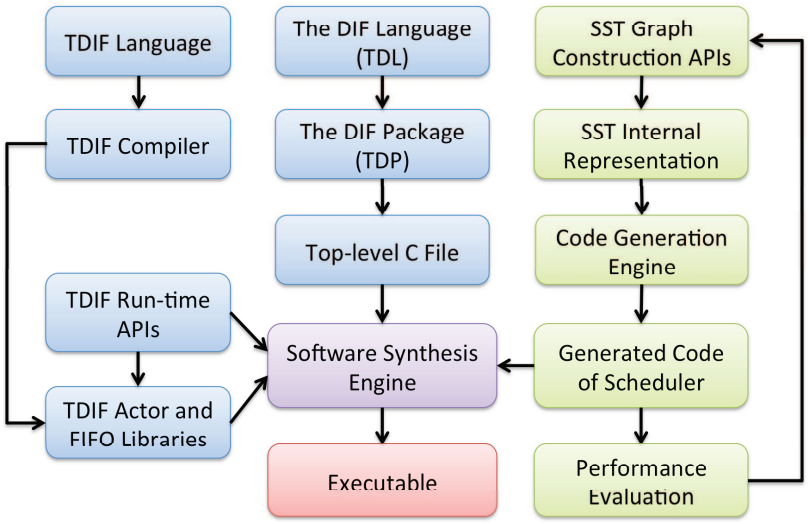

Figure 4.2: TDIF design flow.

Chapter 5
Case Study: Image Registration Application

To demonstrate our methods for representation of and code generation from schedules for dataflow graphs that employ topological patterns, and to demonstrate also the capabilities of our associated new plug-in to the DIF framework, we developed an image registration application based on the *Scale-Invariant Feature Transform* (*SIFT*) algorithm as a case study [16]. SIFT is a well-known algorithm in computer vision for feature detection in and matching of images.

5.1 Application Overview

Image registration is a process of geometrically aligning two or more images of the same scene so that they can be overlaid [28]. Here, one of the images is referred to as the `reference image` and the second image is referred to as the `target image`. Figure 5.1 shows the design flow of our proposed image registration system in terms of a dataflow graph.

5.1.1 Scale-Invariant Feature Transform

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. The dataflow graph in Figure 5.1 for the SIFT algorithm consists of five actors. These are actors for `Cascade Gaussian Filtering`, `Difference of Gaussian` computation, `Local Extrema Detection`, `Post Processing`, and `Descriptor Assignment`.

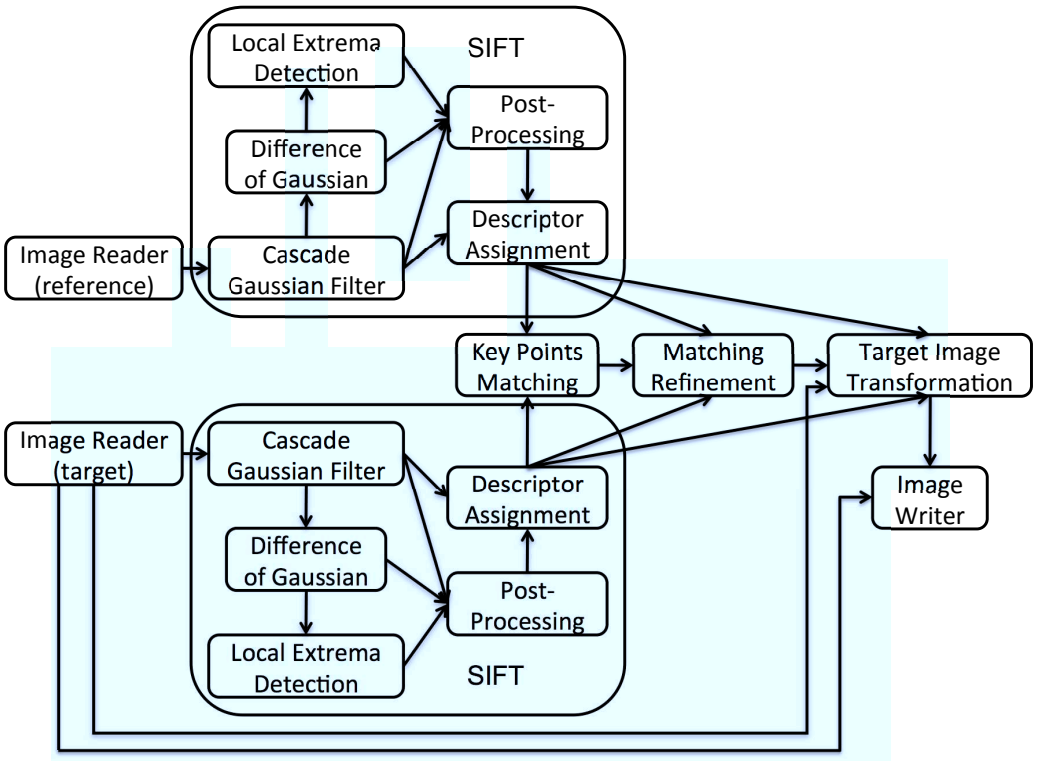

Figure 5.1: A dataflow graph model of the image registration application.

The `Cascade Gaussian Filtering` actor implements a *cascade Gaussian filtering* subsystem, which contains a number of Gaussian filters with different standard deviations. These filters produce a series of Gaussian filtered images. Neighboring images that are filtered by `Cascade Gaussian Filtering` (e.g., see Figure 5.2) are subtracted by the `Difference of Gaussian` actor to produce a series of differences 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

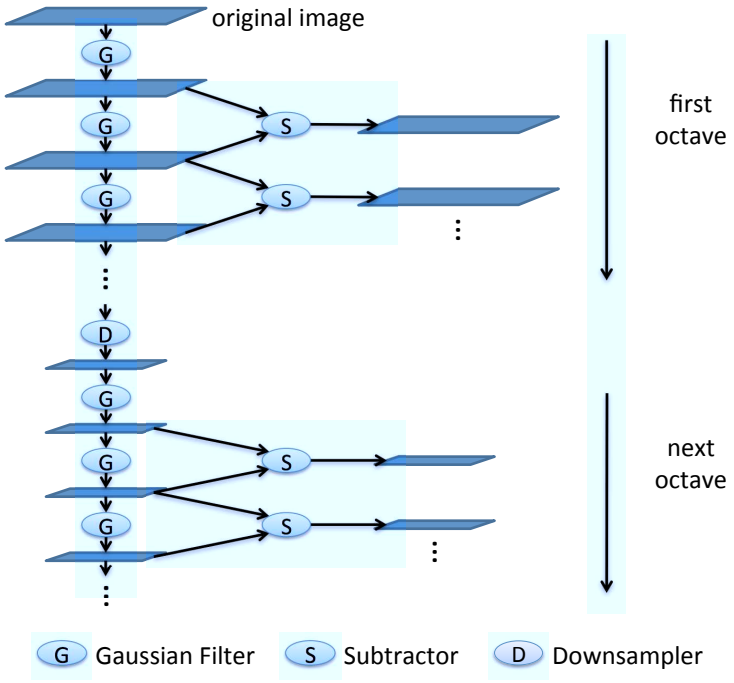

Figure 5.2: Cascade Gaussian filtering and the process for generating differences of Gaussian images.

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 are extracted and stored as key point descriptors by the `Descriptor Assignment` actor. We ported MATLAB implementations of the SIFT algorithm [27] to the dataflow actors and implemented them using C and CUDA.

5.1.2 Key Points Matching

When performing feature matching between two images, key point i in an image A is matched to 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.

5.1.3 Matching Refinement

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 [5]. RANSAC is an iterative method to estimate parameters of a mathematical model from a set of observed data consisting of both inliers and outliers. In our case, inliers are correct matches and outliers are false matches.

The pseudocode shown in Algorithm 3 outlines our implementation of the RANSAC algorithm. Both `iteration` and `threshold` are parameters that can be configured by the designer. As an example, Figure 5.4 shows the key points matching of the two images shown in Figure 5.3 before running RANSAC, and Figure 5.5 shows the key points matching after running RANSAC.

5.1.4 Target Image Transformation

As shown in Fig. 5.1, the `Target Image Transformation` actor performs the computation of target image transformation by taking the outputs produced by the SIFT computation, the refined matching result and the target image and producing the resulting registered image. For the computation of target image transformation, we use a rigid transformation process, which can be divided into three steps, `translation`, `rotation` and `scaling` (see Figure 5.6).

Algorithm 3 Outline of the RANSAC algorithm as we have applied it.

```
/*  track the number of match pairs in the best inliers */
count_best = 0;

for (i = 0; i < iteration; i++) {
    /* select one match pair (K1, K2). (x1, y1)
       and (x2, y2) are coordinates of the two
       key points in terms of the pixel position */
    rand_match = randomly selected match pair;

    /*  track the number of match pairs in the inliers */
    count = 0;

    for (j = 0; j < number_matches; j++) {
        /* (K3, k4) is jth match pair, (x3, y3)
           and (x4, y4) are their coordinates */
        deltax = (x1 - x2) - (x3 - x4);
        deltay = (y1 - y2) - (y3 - y4);
        /* error measures how likely this match pair is an inlier */
        error = pow(deltax, 2) + pow(deltay, 2);

        if (error < threshold) {
          add jth match pair into inliers;
          count++;
        }
    }

    if (count > count_best) {
      count_best = count;
      best_inliers = inliers;
    }
}
```

Figure 5.3: Original images for RANSAC example.

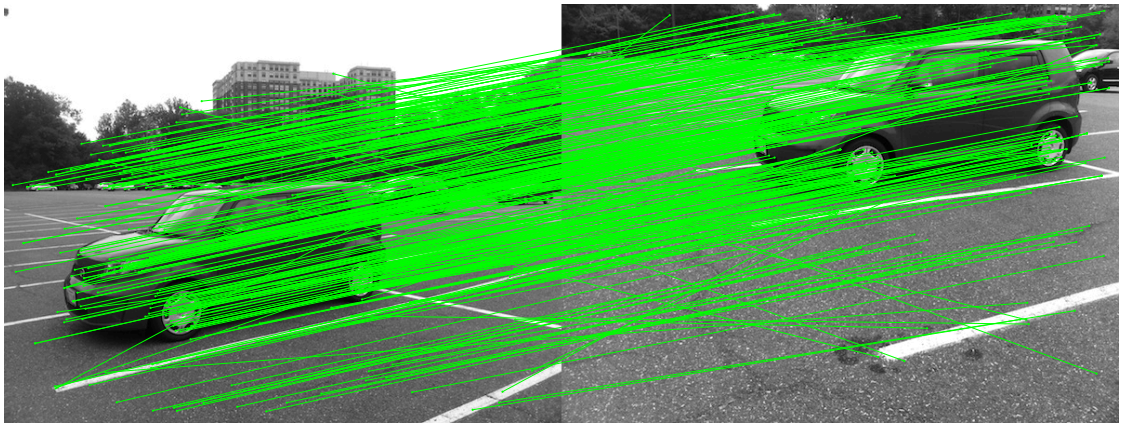

Figure 5.4: Key points matching before running RANSAC.

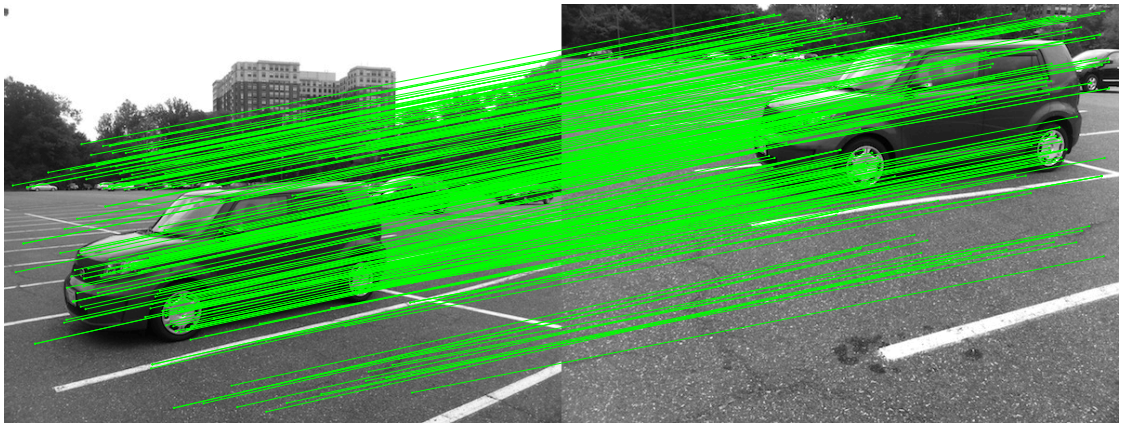

Figure 5.5: Key points matching after running RANSAC.

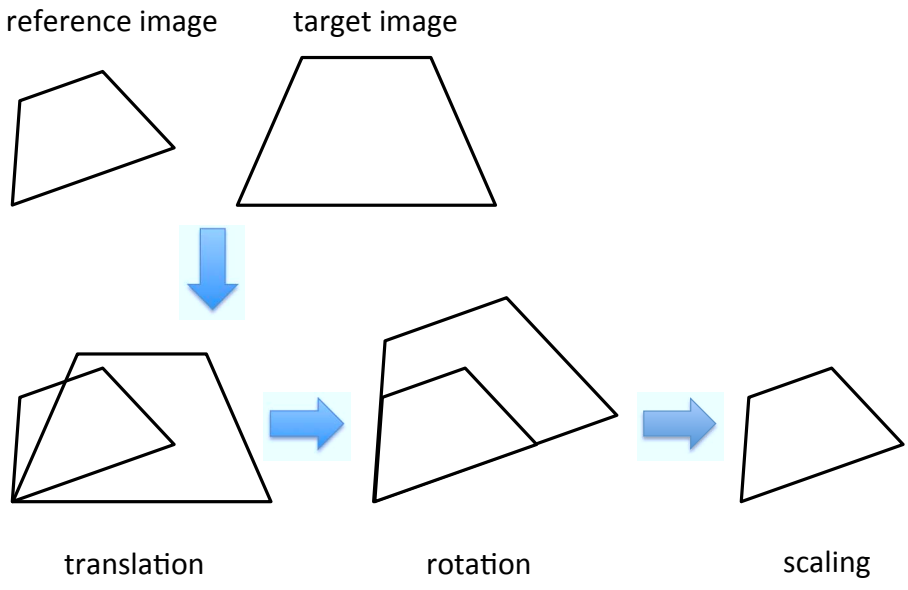

Figure 5.6: Steps in rigid target image transformation.

1. In the `translation` step, we use one key points match pair from the Matching Refinement step as the basis of translation. This pair is the randomly selected match pair at the beginning of each iteration of the RANSAC algorithm. The reason for using this pair as the basis of the target image transformation step is that this match pair is unlikely to be a false match. Suppose the coordinates of this match pair are $(x1, y1)$ and $(x2, y2)$. Then the translation vector is $(x1 - x2, y1 - y2)$.

2. The computation of the `rotation` step is carried out in the polar coordinate system. The pole of this polar coordinate system is the coordinate of the key point in the reference image of the match pair mentioned in our description of the translation step. First, we convert the coordinates of matched key points from the Cartesian coordinate system to the polar coordinate system. Then we determine the rotation angle θ using the key point matching information.

3. The `Scaling` step is computed in the polar coordinate system. Since the target image has been rotated, each key points match pair should be aligned with the pole. Therefore, the ratio of scaling is the ratio of the radius of the key points match pair.

Now that we have the translation vector, rotation angle and scaling ratio, we can use them to determine the corresponding positions (in the target image) of each pixel in the resulting image. These positions are coordinates with fractions. We use bilinear interpolation to determine each pixel value in the resulting image by taking weighted average values of four surrounding pixels in the target image to reduce

visual distortion.

5.2 Applying the Scalable Schedule Tree

Cascade Gaussian filtering is a relevant case study for experimenting with topological patterns and SSTs because it can be modeled naturally in terms of parameterized topologies. Here, we model the cascade Gaussian filtering actor as a subsystem. It can be modeled as a dataflow graph consisting of actors that perform Gaussian filtering and downsampling computations. These computations 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 Figure 5.2, 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 Figure 5.2, 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 Program 1. 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.

Program 1 TDL code for cascade Gaussian filtering.

```
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]);
}
```

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 [9]).

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 Figure 5.7. 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 Figure 5.7), 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. The Java code shown in Program 2 demonstrates how this SST

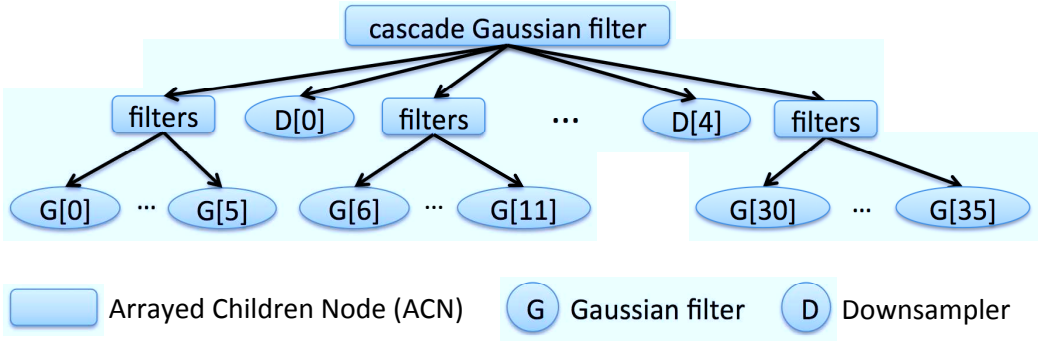

Figure 5.7: SST representation for the cascade Gaussian filtering application.

can be built by using the SST plug-in that is introduced in Section 4.

The generated code segment from the SST for the cascade Gaussian filtering application in the TDIF environment is shown as Program 3. In this code segment, `tdifc_ec_enable_check` is the TDIF run-time API that implements the *enable method* to test for sufficient input data for execution of a given actor, and `tdifc_ec_invoke` is the TDIF run-time API that implements the *invoke method* to execute a single invocation for that actor. Also, `tdifc_lib_<A>_ec` and `tdifc_lib_<A>_tc` implement the *execution context* and the *topological context*, respectively, which are instances of retargetable data structures that encapsulate relevant state information of an actor `<A>`.

To learn details on the `enable method` and the `invoke method`, we refer the reader to [19]. For details on the `execution context` and `topological context`, we refer the reader to [24].

5.3 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

Program 2 Java code for building an SST for cascade Gaussian filtering using our SST plug-in.

```
/* parameters */
int o = 6, s = 6;

/* cascade Gaussian filter ACN */
ScalableScheduleTree cgf
        = new ScalableScheduleTree();
cgf.addACN("CGF", 0);

/* filters ACN */
ScalableScheduleTree filters[] = new ScalableScheduleTree[o];

/* downsamplers */
ScalableScheduleTree d[] = new ScalableScheduleTree[s-1];

for (int i = 0; i < o-1; i++) {
    filters[i] = new ScalableScheduleTree();
    filters[i].addACN("G", s);
    cgf.insertSchedule(filters[i]);
    d[i] = new ScalableScheduleTree();
    d[i].addSchedule("D");
    cgf.insertSchedule(d[i]);
}

filters[o-1] = new ScalableScheduleTree();
filters[o-1].addACN("G", s);
cgf.insertSchedule(filters[o-1]);
```

Program 3 A segment of code that is generated in the TDIF environment from the SST for the cascade Gaussian filtering application.

```
if (tdifc_ec_enable_check(tdifc_lib_g0_ec, tdifc_lib_g0_tc)) {
    tdifc_ec_invoke(tdifc_lib_g0_ec, tdifc_lib_g0_tc);
}
...
if (tdifc_ec_enable_check(tdifc_lib_g5_ec, tdifc_lib_g5_tc)) {
    tdifc_ec_invoke(tdifc_lib_g5_ec, tdifc_lib_g5_tc);
}
if (tdifc_ec_enable_check(tdifc_lib_d0_ec, tdifc_lib_d0_tc)) {
    tdifc_ec_invoke(tdifc_lib_d0_ec, tdifc_lib_d0_tc);
}
if (tdifc_ec_enable_check(tdifc_lib_g6_ec, tdifc_lib_g6_tc)) {
    tdifc_ec_invoke(tdifc_lib_g6_ec, tdifc_lib_g6_tc);
}
...
if (tdifc_ec_enable_check(tdifc_lib_g11_ec, tdifc_lib_g11_tc)) {
    tdifc_ec_invoke(tdifc_lib_g11_ec, tdifc_lib_g11_tc);
}
...
if (tdifc_ec_enable_check(tdifc_lib_d4_ec, tdifc_lib_d4_tc)) {
    tdifc_ec_invoke(tdifc_lib_d4_ec, tdifc_lib_d4_tc);
}
if (tdifc_ec_enable_check(tdifc_lib_g30_ec, tdifc_lib_g30_tc)) {
    tdifc_ec_invoke(tdifc_lib_g30_ec, tdifc_lib_g30_tc);
}
...
if (tdifc_ec_enable_check(tdifc_lib_g35_ec, tdifc_lib_g35_tc)) {
    tdifc_ec_invoke(tdifc_lib_g35_ec, tdifc_lib_g35_tc);
}
```

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). We apply this metric on various applications, 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.

5.3.1 LOC Evaluation for Topological Patterns

We first compare LOC evaluation results by using TDL with and without the support of topological patterns. Table 5.1 shows a comparison result in terms of LOC for TDL specifications with and without the support of topological patterns for different applications. For the specifications in this comparison, each node and edge declaration occupies a separate line of code. As an example, Program 4 and Program 5 shows the TDL specifications of the image registration application (see Figure 5.1) without support for topological patterns and with support for topological patterns respectively.

Program 4 TDL code for the image registration application without support for topological patterns.

```
topology {
    nodes = IR_r, IR_t, CGF_r, CGF_t, DOG_r, DOG_t,
            LED_r, LED_t, PP_r, PP_t, DA_r, DA_t,
            KPM, MR, TIT, IW;
    edges = e0(IR_r, CGF_r),
            e1(IR_t, CGF_t),
            e2(CGF_r, DOG_r),
            e3(CGF_t, DOG_t),
            e4(DOG_r, LED_r),
            e5(DOG_t, LED_t),
            e6(LED_r, PP_r),
            e7(LED_t, PP_t),
            e8(PP_r, DA_r),
            e9(PP_t, DA_t),
            e10(DA_r, KPM),
            e11(DA_t, KPM),
            e12(CGF_r, PP_r),
            e13(CGF_t, PP_t),
            e14(CGF_r, DA_r),
            e15(CGF_t, DA_t),
            e16(DOG_r, PP_r),
            e17(DOG_t, PP_t),
            e18(KPM, MR),
            e19(MR, TIT),
            e20(TIT, IW),
            e21(DA_r, MR),
            e22(DA_t, MR),
            e23(DA_r, TIT),
            e24(DA_t, TIT),
            e25(IR_t, TIT),
            e26(IR_t, IW);
}
```

Program 5 TDL code for the image registration application with support for topological patterns.

```
topology {
    nodes = REF[6], TAR[6], REGIST[4];
    edges = e0[9] -> chain(REF[0:5], REGIST[0:3]),
            e1[6] -> chain(TAR[0:5], REGIST[0]),
            e2[2] -> broadcast(REF[1], REF[4:5]),
            e3[2] -> broadcast(TAR[1], TAR[4:5]),
            e4(REF[2], REF[4]),
            e5(TAR[2], TAR[4]),
            e6[2] -> broadcast(REF[5], REGIST[1:2]),
            e7[2] -> broadcast(TAR[5], REGIST[1:2]),
            e8[2] -> broadcast(TAR[0], REGIST[2:3]);
}
```

5.3.2 LOC Evaluation for TDIF Framework

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 4, TDIF Version 0.2 contains a code generator to translate SSTs into C code that implements the corresponding schedules. 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 [24]. Additionally, the TDIF environment provides translation from DIF specifications

Table 5.1: LOC comparisons for TDL specifications with and without the support of topological patterns (TPs).

Application	without TP	with TP
Cascade Gaussian filter	81	3
Image registration	43	12
JPEG encoder	37	9
FFT (size $N = 8$)	32	2

into top-level C language implementations that construct and execute the specified dataflow graphs.

Table 5.2 summarizes the LOC costs for different implementation components for the cascade Gaussian filter application when code generation is used — i.e., 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, and the total LOC costs c in the designer-written component of the actor implementations.

On the other hand, Table 5.3 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.

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. 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 help to reduce the LOC cost for the top-level DIF specification.

Table 5.2: 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 5.3: 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

5.4 Evaluation in Terms of Execution Time

5.4.1 TDP Processing Time

As shown in Section 5.3, support for topological patterns notably reduces the amount of input a designer needs to provide when using TDL to specify a system. In this section, we evaluate the TDP processing time with and without support for topological patterns. Here, by *TDP processing time*, we mean the execution time of TDP in reading the TDL specification file and storing the dataflow graph information within intermediate representations.

Table 5.4 shows our comparison results. The processing time is slightly faster for TDP with support for topological patterns. The input TDL specification specifies the dataflow graph of the image registration application shown in Figure 5.1. The corresponding TDL code is shown in Program 4 and Program 5. The results are obtained according to the average execution time for 100 runs in each of the two

Table 5.4: Execution time for reading the TDL specification file and storing the dataflow graph information within appropriate intermediate representations.

without support for TPs (sec)	with support for TPs (sec)
0.973	0.943

Table 5.5: Image registration application execution time for the dataflow-based implementation in the TDIF environment and a conventional implementation (without dataflow-based modeling).

Implementation in TDIF (sec)	Plain implementation (sec)
30.523	30.476

cases.

5.4.2 Application Execution Time

In this section, we compare the image registration application execution time for the dataflow-based implementation (see Figure 5.1) in the TDIF environment, and the "conventional" implementation without dataflow-based modeling. Table 5.5 shows the comparison results. The input images are 1200×900 gray-scale bitmap images. The results are obtained according to the average execution time for 10 runs in each of the two cases. We see that the execution times of the two cases are very close, which means that the coding efficiency of our new modeling approach does not come at significant performance cost when using this application in our design framework.

5.5 Cross-Platform Experimentation

5.5.1 Cascade Gaussian Filtering

TDIF includes capabilities for targeting CUDA-enabled graphics processing units (GPUs) in addition to pure C code ("CPU targeted") implementations [24]. As part of this application case study, we experimented with the CUDA-targeted synthesis capability of TDIF for the cascaded Gaussian filter application. As our experiments show, 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. Table 5.6 shows a performance comparison of CPU-targeted and GPU-targeted implementations for the cascade Gaussian filtering application. Both implementations were 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 100 runs in each of the two cases.

Table 5.6: Performance comparison for CPU-targeted and GPU-targeted cascade Gaussian filtering implementations.

CPU (sec)	GPU (sec)	Speedup
11.79282	0.46281	25.48

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. 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 5.3. 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, which are employed in related kinds of dataflow specifications.

5.5.2 Image Registration Results

In this section, we show experimental results for the whole image registration application and provide a performance comparison for CPU-targeted and GPU-targeted implementations. In these experiments, the cascade Gaussian filter is no longer modeled as a system that contains many actors. It is just one actor in the overall image registration application illustrated in Figure 5.1. We demonstrate image registration results with two examples. Figure 5.8, Figure 5.9 and Figure 5.10 show the reference image, target image and result image for the first example, respectively. Similarly, Figure 5.11, Figure 5.12 and Figure 5.13 show the reference

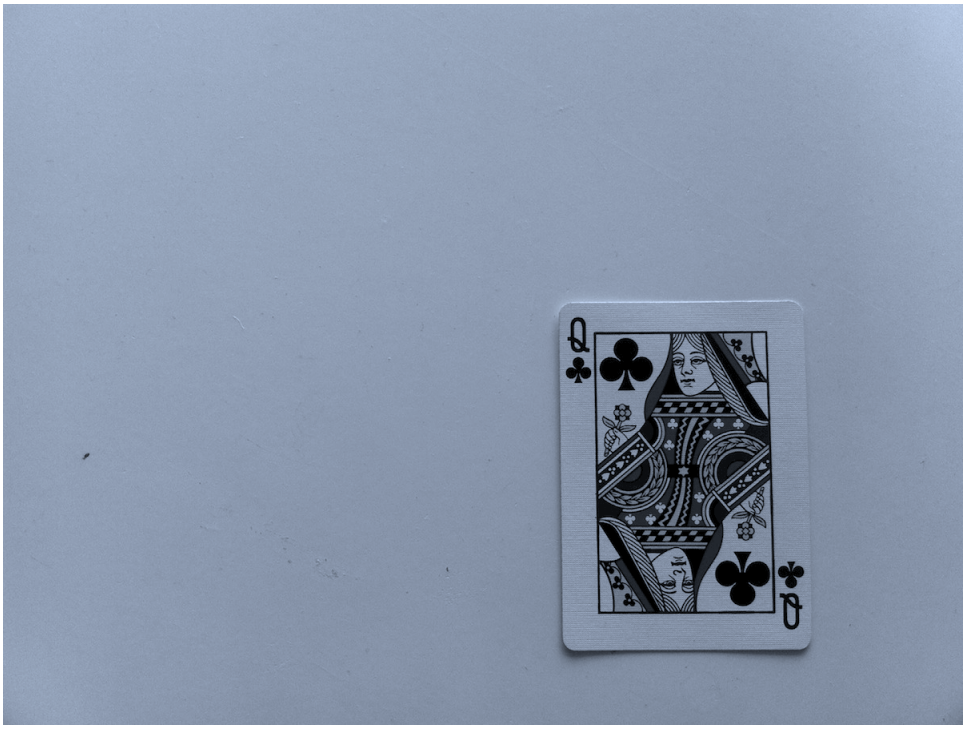

Figure 5.8: Reference image for Example 1.

image, target image and result image for the second example, respectively.

Table 5.7 shows a performance comparison between CPU-targeted and GPU-targeted implementations for the GPU-targetable actors and the overall image registration application. Inputs to the application are again 1200×900 gray-scale bitmap images, 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. The results are obtained according to the average execution time for 10 runs in each of the two cases.

Figure 5.9: Target image for Example 1.

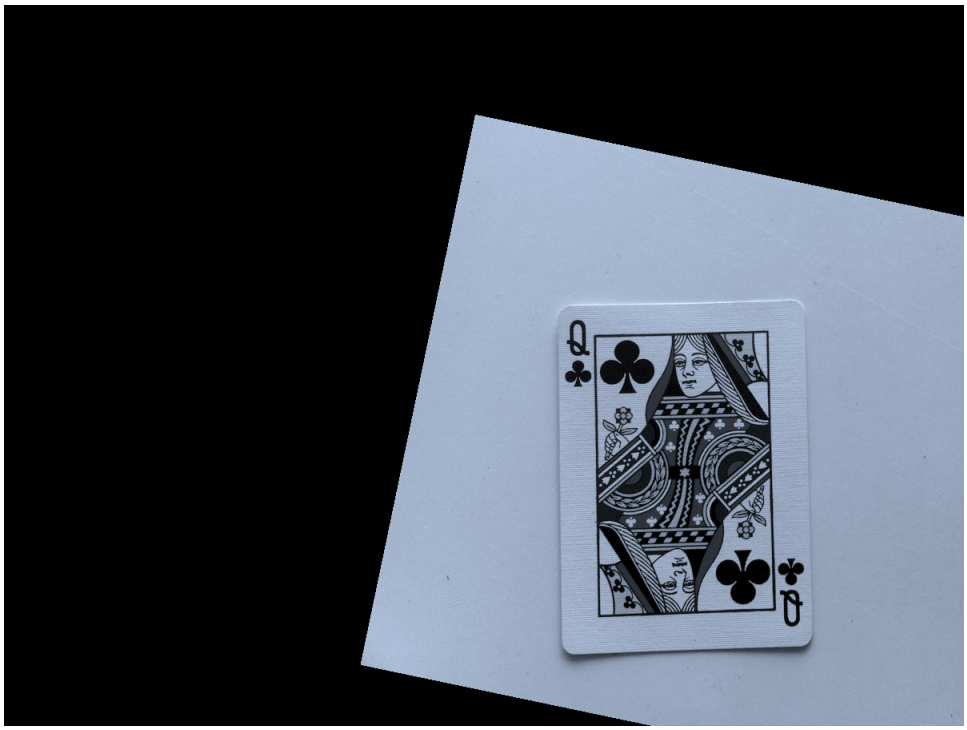

Figure 5.10: Result image for Example 1.

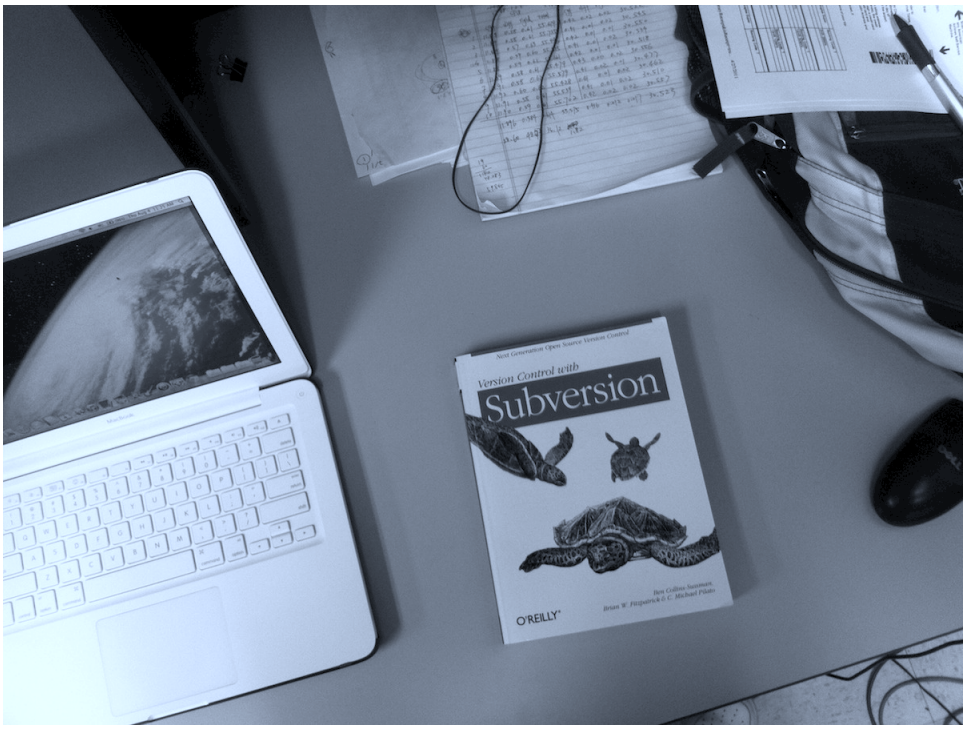

Figure 5.11: Reference image for Example 2.

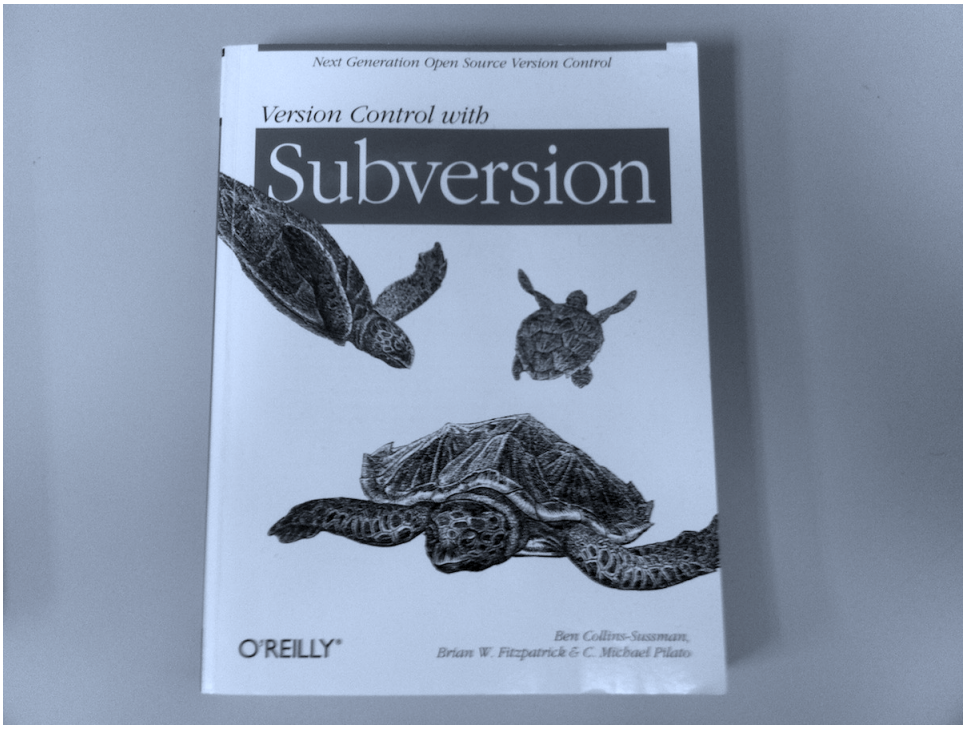

Figure 5.12: Target image for Example 2.

Figure 5.13: Result image for Example 2.

Table 5.7: Performance comparison between CPU-targeted and GPU-targeted implementations for the GPU-targetable actors and the overall image registration application.

	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
Whole application	55.575	30.523	1.82

Chapter 6
Conclusions and Future Work

6.1 Conclusions

In this thesis, we have presented a novel schedule model called the scalable schedule tree (SST) for representing parameterized schedule structures based on topological patterns. We have also presented language extensions for specifying topological patterns and a new plug-in to the dataflow interchange format (DIF) framework for specifying SSTs that execute dataflow models with topological patterns, and for generating C code that implements the parameterized schedules represented by these SSTs. 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.

6.2 Future Work

Useful directions for further work include the following:

- developing techniques for automated derivation of SSTs;
- exploring SSTs that incorporate more complex forms of adaptivity;
- supporting code generation on additional classes of platforms, such as field programmable gate arrays and multicore digital signal processors;
- incorporating into TDL the SST plug-in that we have developed in this work;
- extending TDL with additional topological patterns; *and*

- further development of non-rigid image registration applications based on the scale-invariant feature transform (SIFT) algorithm.

Bibliography

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

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

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

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

[5] M. A. Fischler and R. C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Comm. of the ACM*, 24:381–395, June 1981.

[6] E. Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, March 1998.

[7] M. Geilen and T. Basten. Reactive process networks. In *Proceedings of the International Workshop on Embedded Software*, pages 137–146, September 2004.

[8] R. Gu, J. W. Janneck, S. S. Bhattacharyya, M. Raulet, M. Wipliez, and W. Plishker. Exploring the concurrency of an MPEG RVC decoder based on dataflow program analysis. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(11):1646–1657, November 2009.

[9] 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.

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

[11] H. Kee, S. S. Bhattacharyya, and J. Kornerup. Efficient static buffering to guarantee throughput-optimal FPGA implementation of synchronous dataflow graphs. In *Proceedings of the International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 136–143, Samos, Greece, July 2010.

[12] M. Ko, C. Shen, and S. S. Bhattacharyya. Memory-constrained block processing for DSP software optimization. *Journal of Signal Processing Systems*, 50(2):163–177, February 2008.

[13] 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.

[14] 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*, 37(11):1751–1762, November 1989.

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

[16] D. G. Lowe. Distinctive image features from scale-invariant keypoints. 60(2):91–110, 2004.

[17] 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.

[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*, 37:41–51, May 2004.

[19] 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.

[20] W. Plishker, N. Sane, M. Kiemb, and S. S. Bhattacharyya. Heterogeneous design in functional DIF. *Transactions on High-Performance Embedded Architectures and Compilers*. Online version available from http://www.hipeac.net/node/3030, print version to appear.

[21] 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.

[22] 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.

[23] 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*, pages 13–18, San Francisco Bay Area, USA, October 2010.

[24] 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.

[25] 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.

[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.

[27] A. Vedaldi. An open implementation of the SIFT detector and descriptor. Technical Report 070012, UCLA CSD, 2007.

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