PARAMETERIZED SCHEDULING FOR SIGNAL PROCESSING SYSTEMS USING TOPOLOGICAL PATTERNS

Shenpei Wu, Chung-Ching Shen, Nimish Sane, Kelly Davis, and Shuvra S. Bhattacharyya

Department of Electrical and Computer Engineering, and
Institute for Advanced Computer Studies
University of Maryland, College Park, MD 20742, USA
{spwu, ccshen, nsane, kdavis18, ssb}@umd.edu

ABSTRACT

In recent work, a graphical modeling construct called "topological patterns" has been shown to enable concise representation and direct analysis of repetitive dataflow graph sub-structures in the context of design methods and tools for digital signal processing systems [1].

In this paper, 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*. The approach represents an important class of parameterized schedule structures in a form that is intuitive for representation and efficient for code generation. We demonstrate our methods for topological pattern representation, scalable schedule tree derivation, and associated dataflow graph code generation using a case study for image processing.

***Index Terms*—** scheduling, software tools, image registration.

1. INTRODUCTION

In 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 [2].

For dataflow models of large-scale DSP applications, the underlying graph representations often consist of smaller sub-structures that repeat multiple times. *Topological patterns* (*TPs*) 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 [1]. Furthermore, by allowing designers to explicitly identify such repeating structures, use of TPs 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 TP 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.

Scheduling is a critical aspect of implementing dataflow graphs (e.g., see [2]). Parameterized schedules have been studied before (e.g., see [3, 4]), and previously, production and consumption rates were key dataflow graph aspects that were used to generate parameterized schedules. 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 [5], FRDF [6], and CSDF [7], for dynamically reconfiguring the behavior of dataflow actors, edges, subsystems, and graphs through parameter values [3]. Quasi-static scheduling techniques were developed for parameterized synchronous dataflow (PSDF), which is the integration of the parameterized dataflow meta-model with SDF as the base model [3]. 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* and *parameterized Kahn process network* model of computation can be viewed as extensions of the Kahn process network (KPN) modeling framework [8, 9], 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 do not address the scheduling of scalable topologies.

In this paper, we present a formal design method for specifying TPs 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 its behavior that may contain parameterizable patterns of actors and edges instantiations. We have also developed an associated software plug-in and integrated it into the *dataflow interchange format* (*DIF*) framework [10] and the associated cross-platform design and synthesis environment called *targeted DIF* (*TDIF*) [11]. 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 [11].

2. SCALABLE SCHEDULE TREES

The *generalized schedule tree* (*GST*) is a compact, tree-structured graphical format that can represent a variety of dataflow graph schedules [4]. 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.

In this paper, we go significantly beyond the capabilities of GSTs by formulating and implementing a novel schedule tree model for representing scalable schedules (i.e., schedules that symbolically accommodate variations in the numbers of actors and edges in the associated dataflow graphs that employ TPs). We refer to this new form of schedule tree as the *scalable schedule tree* (*SST*) model.

2.1. SST Model

A *scalable schedule tree* (*SST*) has all of the features of a GST (e.g., see [4]) and additionally provides the following new features.

1. Parameterization. A node within an SST can be parameterized with a parameter set K. The semantics of how values associated with elements of K change is determined by the model of computation that is used for application specification (e.g., SDF with static graph parameters [12], parameterized dataflow [3], or scenario aware dataflow [13]), 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 different kinds of dataflow application models and design environments.

2. Guarded execution. An SST leaf node, which encapsulates a firing (execution) 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. The guarded attribute of SSTs is motivated by the enable-invoke dataflow model of computation, where guarded executions play a fundamental role [11].

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* $c_L : K \rightarrow \text{Z+}$. 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, a third kind of internal node, called an *arrayed children node* (*ACN*), is introduced to represent schedule structures related to TPs.

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.

2.2. 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.

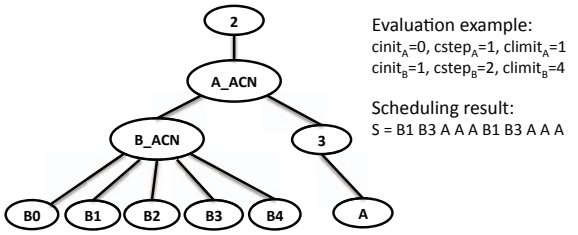

Fig. 1. An example of an SST.

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]
   }
}
```

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

2.3. Integration in the DIF Framework

We have implemented a new plug-in to the DIF framework that extends the DIF language (TDL) to incorporate support for TPs and allows designers to construct SSTs for schedules associated with dataflow graphs that are specified in TDL. This plug-in integrates the SST formulations developed in Section 2 as a new internal representation format and associated set of manipulations within the DIF framework. TPs that are currently supported by TDL and defined as *pattern keywords* in the language include `chain`, `ring`, `merge`, `broadcast`, `parallel`, and `butterfly`.

We have also integrated specification and code generation support into the TDIF environment for SSTs. In this integration, 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.

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.

3. CASE STUDY: CASCADE GAUSSIAN FILTERING

To demonstrate our methods and associated new plug-in for representation of and code generation from schedules for dataflow graphs that employ TPs , we use the *cascade Gaussian filtering* (*CGF*) subsystem in the *Scale-Invariant Feature Transform* (*SIFT*) algorithm as a case study [14]. SIFT is a well-known algorithm in computer vision for feature detection and matching of images.

The CGF subsystem contains a number of Gaussian filters with different standard deviations. These filters produce a series of Gaussian filtered images. CGF is a relevant case study for experimenting with TPs and SSTs because it can be modeled naturally in terms of parameterized topologies. As shown in Fig. 2(a), CGF 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 CGF process illustrated in Fig. 2(a), the original image is convolved with the first filter. The filtered image is saved and then convolved with the next filter, and so on. After one group of filtering operations is carried out, s different blurred Gaussian images are labeled as a separate octave. The next step is to downsample the last image of the previous octave by a factor of two. This process, as shown in Fig. 2(a), repeats until o octaves of images are produced.

3.1. Applying the Scalable Schedule Tree

The TP underlying the CGF application is a chain (linear arrangement of actors), which can be specified in TDL. Fig. 2(b) shows the TDL specification with $o = 6$ and $s = 6$. 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.

In this CGF example, since both o and s are parameters that can be configured, one can naturally derive a nested SST as shown in Fig. 2(c). Such a representation provides a formal, target-language-independent model of schedule structure that can be applied to coordinate execution for this subsystem in a manner that is parameterized across two dimensions.

As shown in Fig. 2(c), the `cascade Gaussian filter` ACN has 11 children nodes, which include 6 nested ACNs, each labeled as `filter`, and 5 `downsampler` actors encapsulated as leaf nodes, which are labeled as `D[0]`, `D[1]`, ..., `D[4]`. Each of these leaf nodes represents an encapsulation of a `downsampler` actor in the CGF 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 filter` actor in the application.

3.2. Evaluation in Terms of Coding Efficiency

Our design framework for specifying TPs enables concise and scalable representation of DSP applications. To help quantify this kind of benefit, we apply an evaluation metric called the *lines of code* (*LOC*), which is the number of lines of code required for an application. Unless otherwise specified, the LOC cost refers to code that the designer needs to manually provide (e.g., in contrast to code that is automatically generated or reused from some other part of an implementation). We apply this metric on various applications, including the CGF application, that are specified with and without use of TPs.

We first compare LOC evaluation results, as shown in Table 1(a), for different applications by using TDL with and without the support of TPs. For the specifications in this comparison, each node and edge declaration occupies a separate line of code.

App	w/o TP	w/ TP
CGF	81	3
JPEG encoder	37	9
FFT (N=8)	32	2

(a) Comparison for TDL specifications

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

(b) Designer-written code in TDIF

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

(c) Implementations generated by TDIF

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

(d) CGF Implementations

Table 1. LOC costs (a-c), and performance comparison (d).

We also compare the LOC cost of CGF implementation that uses code generation and the LOC cost of the generated code in the TDIF environment. 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. Due to space limitations, we omit the listings of these code implementations.

Table 1(b) summarizes the LOC costs for different implementation components for the CGF 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 1(c) shows the LOC costs of the complete generated implementation — i.e., the generated code together with the designer-written TDIF input code that is used directly (without translation) in the implementation.

In the CGF application, the underlying TP is a chain, and the number of edges is of the same order as the number of nodes. Thus, comparing the LOC listings in Table 1(b) and Table 1(c), 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 TPs, which help to reduce the LOC cost for the top-level DIF specification.

3.3. Cross-Platform Experimentation

TDIF includes capabilities for targeting CUDA-enabled graphics processing units (GPUs) in addition to pure C code ("CPU targeted") implementations [11]. As part of this case study, we experimented with the CUDA-targeted synthesis capability of TDIF for the CGF application. This aspect of our case study validates the utility of TPs 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.

In these experiments, input to the application is a 1200×900 gray-scale bitmap image, and the implementations are executed on a

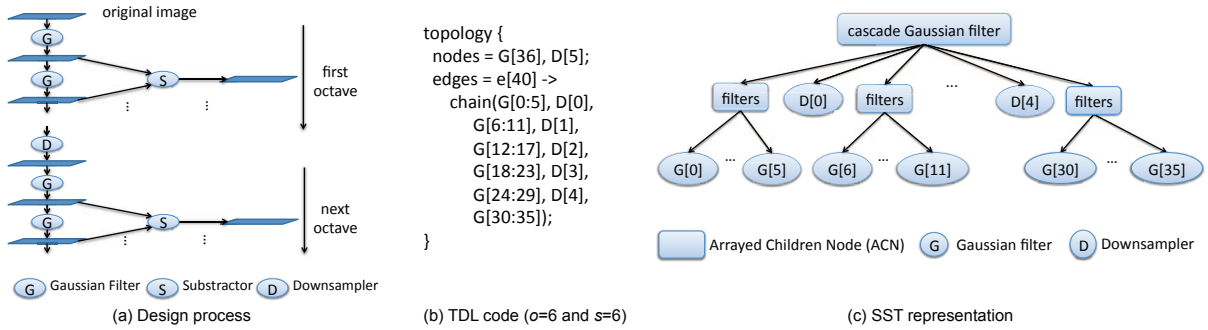

Fig. 2. Cascade Gaussian filtering.

3GHz PC with an Intel CPU that is equipped with 4GB RAM, and co-located with an NVIDIA GTX260 GPU. Table 1(d) shows a performance comparison of CPU- and GPU-targeted implementations for the CGF application. Both implementations were generated by TDIF based on SSTs that exploit TP structures in the application specifications. The results are obtained according to the average execution time for 100 runs in each of the two cases.

The results show that GPU acceleration provides significant benefit in this application, and validates the retargetability of our use of TPs 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 3.2. 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 TPs 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.

4. CONCLUSIONS

In this paper, we have presented a novel scalable schedule tree (SST) model for representing parameterized schedule structures based on topological patterns. We have also presented a new plug-in to the DIF framework for specifying SSTs that execute dataflow models with topological patterns, and for generating C code from traversing these SSTs. We have validated our new methods and tools, and demonstrated their utility through a case study centered around cascade Gaussian filtering for image processing. Useful directions for further work include exploring SSTs that incorporate more complex forms of adaptivity, and supporting code generation on additional platforms, such as FPGAs and multicore digital signal processors.

5. ACKNOWLEDGEMENT

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

6. REFERENCES

[1] 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*, October 2010.

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

[3] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Transactions on Signal Processing*, October 2001.

[4] 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*, June 2007.

[5] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proceedings of the IEEE*, September 1987.

[6] 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*, May 2004.

[7] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, February 1996.

[8] M. Geilen and T. Basten, "Reactive process networks," in *Proceedings of the International Workshop on Embedded Software*, September 2004.

[9] 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.

[10] C. Hsu, M. Ko, and S. S. Bhattacharyya, "Software synthesis from the dataflow interchange format," in *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, September 2005.

[11] 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*, July 2011.

[12] 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*, November 1989.

[13] 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.

[14] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International Journal of Computer Vision*, 2004.