In Proceedings of the IEEE Workshop on Signal Processing Systems, pages 13-18, San Francisco Bay Area, USA, October 2010.

SCALABLE REPRESENTATION OF DATAFLOW GRAPH STRUCTURES USING TOPOLOGICAL PATTERNS

Nimish Sane[1], *Hojin Kee*[1], *Gunasekaran Seetharaman*[2], *and Shuvra S. Bhattacharyya*[1]

[1]Department of Electrical and Computer Engineering, and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742, USA. {{nsane, hjkee, ssb}@umd.edu}
[2]Air Force Research Laboratory, Rome, NY, USA.{gunasekaran.seetharaman@rl.af.mil}

ABSTRACT

Tools for designing signal processing systems with their semantic foundation in dataflow modeling often use high-level graphical user interface (GUI) or text based languages that allow specifying applications as directed graphs. Such graphical representations serve as an initial reference point for further analysis and optimizations that lead to platform-specific implementations. For large-scale applications, the underlying graphs often consist of smaller substructures that repeat multiple times. To enable more concise representation and direct analysis of such substructures in the context of high level DSP specification languages and design tools, we develop the modeling concept of *topological patterns*, and propose ways for supporting this concept in a high-level language. We augment the DIF language — a language for specifying DSP-oriented dataflow graphs — with constructs for supporting topological patterns, and we show how topological patterns can be effective in various aspects of embedded signal processing design flows using specific application examples.

Index Terms— Dataflow graphs, high-level languages, model-based design, topological patterns, signal processing systems.

1. INTRODUCTION

Dataflow modeling is used extensively for designing signal processing systems. There are various existing design tools with their semantic foundations in dataflow modeling such as Agilent ADS [1], National Instruments LabVIEW [2], Compaan/Laura [3], and SysteMoc [4]. DSP-oriented dataflow design tools typically allow high-level application specification, software simulation, and possibly synthesis for hardware or software implementation. These tools employ high-level description languages for application specification. These languages, which may be either graphical user interface (GUI) or text based, provide syntactic and semantic constructs for specifying graphical representations of DSP applications. Such graphical representations are then parsed and converted into an intermediate representation suitable for further processing.

In this paper, we address the problem of representing large-scale and scalable dataflow graphs that have complex topologies. Such graphs comprise of various kinds of functional substructures that are parameterizable and can be represented in terms of concise, scalable specifications.

For example, the dataflow graph of an N-point fast Fourier transform (FFT) algorithm consists of a combination of scaled versions of a well-known pattern called the *butterfly diagram* [5], and a systolic array is a *mesh* of computing elements having a specific dataflow structure that can solve problems such as QR-decomposition based recursive least square adaptive filtering, and minimum variance distortionless response beamforming [6]. We identify such common structures in dataflow graphs as *topological patterns*, and treat this kind of patterns as first class citizens in the modeling process. Furthermore, we demonstrate and experiment with the use of topological patterns in the *dataflow interchange format* (*DIF*), a textual design language and associated software package for specification, analysis, and synthesis based on DSP-oriented dataflow models of computation [7, 8].

Topological patterns not only permit scalable specifications of dataflow substructures, but also expose the underlying graph structure explicitly to the corresponding design tool. This allows design tools to exploit any analysis or optimization advantages offered by the substructures without having to “discover” those structures through additional levels of pre-processing analysis. Some of the key components of the design flow that can potentially benefit from explicitly exposed patterns include various kinds of scheduling transformations, and techniques for buffer memory optimization. Furthermore, by making it easier and more efficient to apply substructure-specific analysis techniques, programming support for topological patterns will encourage the development of such analysis techniques, and provide a natural interface for reusing them across different applications and tools.

2. BACKGROUND

This section provides background on dataflow modeling and the DIF language. We also discuss earlier research efforts that are relevant to this work.

2.1. Dataflow Modeling

Dataflow modeling involves representing an application using a directed graph $G = (V, E)$, where V is a set of vertices (nodes) and E is a set of edges. Each vertex $u \in V$ in a dataflow graph is called an *actor*, and represents a specific computation block, while each directed edge $(u, v) \in E$ is a first-in-first-out (FIFO) buffer that represents a communication link between the *source* actor u and the *sink* actor v. A dataflow graph edge e can also have a non-negative integer *delay*, $\mathrm{del}(e)$, associated with it, which represents the number of initial data values (*tokens*) present in the associated buffer.

Dataflow graphs operate based on *data-driven execution*, where an actor can be executed (*fired*) whenever it has sufficient amounts of data (numbers of “samples” or “data tokens”) available on all of its inputs. During each firing, an actor consumes a certain number of tokens from each input and produces a certain number of tokens on each output. We refer to these numbers of tokens consumed and produced in each actor execution as the *consumption rate* and *pro-*

duction rate of the associated input and output, respectively. Usually production and consumption rate information is characterized in terms of individual input and output ports so that each port of an actor can in general have a different production or consumption rate characterization. Such characterizations can have constant values as in synchronous dataflow (SDF) [9]; periodic patterns of constant values, as in cyclo-static dataflow (CSDF) [10]; or more complex forms that are data-dependent (e.g., see [11, 12, 8]). A *schedule* for a dataflow graph G is a sequence of actors in G, and represents the order in which actors are fired during an execution of G.

2.2. The Dataflow Interchange Format

To describe dataflow applications for a wide range of DSP applications, application developers can use the DIF language, which is a standard language founded in dataflow semantics and tailored for DSP system design [7]. DIF provides an integrated set of syntactic and semantic features that can fully capture essential modeling information of DSP applications without over-specification. From a dataflow point of view, DIF is designed to describe mixed-grain graph topologies and hierarchies as well as to specify dataflow-related and actor-specific information. The dataflow semantic specification is based on dataflow modeling theory and independent of any design tool.

Fig. 1 illustrates some of the available constructs in the DIF language along with the syntax used for application specification. More details on the DIF language can be found in [7]. The *topology* block of the specification specifies the graph topology, which includes all of the *nodes* and *edges* in the graph. DIF supports *built-in attributes* such as *interface*, *refinement*, *parameter*, and *actor*, which identify specifications related to graph interfaces, hierarchical subsystems, dataflow parameters, and actor configurations, respectively. DIF also allows *user-defined attributes*, which have a similar syntax as built-in attributes except that they need to be declared with the `attribute` keyword.

To facilitate use of the DIF language, the *DIF package* (*TDP*) has been built. Along with the ability to transform DIF descriptions into manipulable internal representation, TDP contains graph utilities, optimization engines, verification techniques, a comprehensive functional simulation framework, and a software synthesis framework for generating C code [7, 8]. These facilities make TDP an effective environment for modeling dataflow applications, providing interoperability with other design environments, and developing and experimenting with new tools and dataflow techniques. Beyond these features, DIF is also suitable as a design environment for implementing dataflow-based application representations. Describing an application graph is done by listing nodes and edges, and then annotating dataflow specific information.

2.3. Related Work

Block diagrams are a natural and convenient way of describing DSP algorithms, and hence, DSP systems designers find it intuitive to have a high-level application specification that captures such a description. GUI based dataflow languages try to capture this intuition using visually appealing representations, while text based languages provide syntax that looks similar to common procedural languages, such as C, but with semantic constructs that model the dataflow structure of DSP block diagrams. To effectively handle the increasing complexity of signal processing system design, these languages must provide frameworks for modular and scalable representations with sufficient expressive power.

```
[dataflowModel] graphID {
basedon { graphID; }
[topology] {
nodes = ndID, ...;
edges = edgeID(srcNdID, snkNdID), ...; }
[builtInAttr] {
elementID = value;
elementID = id;
elementID = id1, id2, ...; }
[attribute] usrDefAttr {
elementID = value;
elementID = id;
elementID = id1, id2, ...; } }
```

Fig. 1. The DIF language

Earlier research efforts have focused on supporting commonly used and highly expressive constructs from procedural languages, such as recurrences, iteration, and conditionals, in dataflow-oriented languages [13]. Subsequent work includes evolution of various textual languages for DSP system design, such as SILAGE [14], StreamIt [15], and CAL [16]. The StreamIt language provides high-level, architecture-independent abstractions for streaming applications geared toward large-scale program development. The CAL language is an actor-oriented language, which has been applied actively for field programmable gate array (FPGA) implementation and reconfigurable video coding applications. The SILAGE language has been developed with an emphasis on support for high level synthesis and multidimensional signal processing.

While these previous efforts have employed useful techniques for deriving and exploiting various types of specialized dataflow substructures within their respective compilers, they lack a general method for explicit and scalable representation of such substructures by the programmer. Such a programming interface for topological patterns is essential to capture the broad range of relevant patterns in ways that are scalable, and flexibly extensible to accommodate new types of patterns as they emerge from new applications and modeling techniques. Our concept of topological patterns is designed precisely to bridge this gap.

In other prior work, higher-order functions have been shown to permit elegant construction of structured subsystems in dataflow representations [17]. Higher-order functions are functions that take functions as inputs or produce functions as outputs. Topological patterns provide a related but technically different approach since topological patterns operate on generic directed graph vertices (e.g., `nodes` in DIF), where the actual binding to actor functionality and associated actor parameter values is specified separately, possibly through additional *parameter propagation patterns* (*PPPs*). Thus, unlike higher-order functions that take functions as arguments, topological patterns take only generic graph vertices (or arrays of such vertices) as arguments. Furthermore, our development of topological patterns is tightly integrated with textual graph representation and arrays of graph vertices and edges, which are useful for providing scalable representations and managing large-scale designs.

Perhaps the most closely related prior work is that on support for arrays of vertices and edges in the DIF language with array construction syntax and semantics similar to those in the C language [18]. These constructs provide a useful shorthand notation for specifying related groups of graph elements (nodes or edges) as arrays in which individual elements can be easily indexed. A typical $elementID$ in the DIF specification (see Fig. 1) when referred to as $baseName[N]$, generates an array of N elements. For example, `tap[`N`]` in DIF specifies an array `tap` of N nodes. The ith node can be accessed using its index as $tap[i-1]$. However, in this *first-version* array support within DIF, there is no mechanism for instantiating (declaring) collections of related edges automatically as structured mappings among corresponding subsets of nodes. It is also not possible to configure parameters across arrays of actors as functions of the array indices. These two features — scalable, programmatic instantiation of graphical substructures, and association of parameter values — are provided by our development of topological patterns. This development is orthogonal to the existing support for syntactic and semantic hierarchy in the DIF language, which allows constructing hierarchical dataflow graphs. The focus here is to allow the designer to specify already identified topological patterns in the design and expose such patterns to the tool, which is generally not achieved through conventional methods for using hierarchical dataflow graphs.

In this paper, we formulate the concept of topological patterns and its application to dataflow modeling, and to prototype this concept in DIF, we build upon the first-version framework of arrays in DIF, and introduce new modeling and language constructs that are dedicated to topological patterns. We also demonstrate the use of topological patterns to derive efficient implementations.

3. TOPOLOGICAL PATTERNS

We have developed a concept of topological patterns for concise specification of functional structures at the dataflow graph (inter-actor) level. Topological patterns provide a scalable approach to specifying regular functional structures in a manner that is analogous in some ways to the use of design patterns in object oriented software [19], but with additional properties associated with being 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.

Topological patterns build on the concepts of *graph element arrays*, which allow indexed families of graph elements to be declared and treated as single units for purposes of graph construction and analysis. As with arrays in conventional programming languages, graph element arrays can be single- or multi-dimensional. Additionally, they can be parameterized in terms of dataflow graph attributes so that their sizes and other characteristics can be conveniently adapted.

3.1. Topological Patterns in Signal Processing

We motivate the utility of incorporating topological patterns into dataflow frameworks for DSP system design by illustrating the pervasive nature of these patterns in the domain of DSP. We have already discussed a few such patterns in Section 1 — in particular, the butterfly and mesh patterns, which have applications in FFTs and systolic arrays, respectively. Additionally, the *chain* pattern is one of the most commonly found topological patterns. This pattern finds applications in modeling multi-stage sample rate converters, delay lines in finite impulse response (FIR) filters, or configurations of pipeline stages. A chain of delay blocks, a chain of adders, and an *array* of filter taps collectively specify a complete FIR filter when connected together. A natural extension of this pattern is a 2-dimensional mesh structure. Such a structure is of particular use to model DSP architectures in which data flows across a network of processing elements connected to form a 2-D grid such as the systolic array, as discussed earlier in Section 1 [6].

A *ring* pattern represents a cycle in the graph as may be introduced by a phase-locked loop [20] or more generally, a *feedback loop* in the system. The FFT block is one of the most abundantly found blocks in DSP systems. An N-point FFT computation involves FFT computation stages of smaller dimensions that can be implemented as scaled versions of 2-point FFT. These FFT stages resemble a butterfly like pattern [5]. Such patterns can also be found in other applications, such as sorting networks [21]. Entropy encoding algorithms such as Huffman coding make use of the *binary tree* structure, a commonly found data structure in many computer algorithms [22]. A pattern in which edges connect a source node to multiple sink nodes could be termed as a *broadcast* pattern. This pattern finds its use in applications that have computation blocks in multiple stages with blocks in one stage connected to those in the subsequent stage. Such patterns are observed in multi-layer neural networks used for pattern classification [23] and trellis coding algorithms used in digital communication [20]. It is also common to find its dual, a *merge* pattern, which connects multiple source nodes to a single sink node. Such applications may also have parallel connections between corresponding nodes in adjacent stages. We identify this pattern as a *parallel* pattern in which edges form a one-to-one correspondence between nodes in two different sets.

3.2. Formulation of Topological Patterns

Although we are interested in the graph theoretic interpretation of topological patterns, we envision them as a more general concept. They provide a suitable abstraction for creating well-defined patterns from ordered collections of objects. A topological pattern can be thought of as a mapping that, when applied to an ordered sequence of objects, establishes a relation between any two objects in the sequence and in turn, generates a structured pattern of inter-object relationships. When represented graphically, these relationships take the form of graph edges. In a dataflow graph, the ordered sequences of objects typically correspond to arrays (or sub-arrays) of dataflow graph nodes.

Let Z_{pos} and Z_{nn} denote the sets of positive and non-negative integers, respectively. For $m \in Z_{\text{pos}}$, $M(m)$ denotes the set of all $m \times m$ matrices over Z_{nn}. For $x \in Z_{\text{nn}}$, let $U(x)$ be defined as $U(x) = \bigcup_{y=1}^{x} M(y)$.

We restrict our discussion to "bounded topological patterns," where each pattern p has an associated bound $b(p) \in Z_{\text{pos}}$ on the number of objects it can handle. This development can be extended naturally to unbounded patterns, but for simplicity, brevity, and reasons of practicality, we restrict our discussion here to finite patterns. Therefore, by the term "topological pattern", we implicitly mean "bounded topological pattern" in the rest of the paper.

For $n \in Z_{\text{pos}}$, let $S_n = \{1, 2, \ldots, n\}$. We can now define a topological pattern p as a mapping from positive integers (indices of ordered sequences) into matrices over Z_{nn}:

$$p : S_{b(p)} \to U(b(p)). \tag{1}$$

Thus, suppose that we have a pattern p with associated bound $b(p) = B \in Z_{\text{pos}}$, and an ordered sequence of objects (e.g., graph vertices) $a_1, a_2, \ldots, a_N$ such that $N \leq B$. Then we have that $p(S_N) \in M(N)$. Thus, $p(S_N)$ is an $N \times N$ matrix $K \in M(N)$, where the interpretation is that any element of K, denoted by $K[i, j]$, $1 \leq i, j \leq N$, gives the number of connections (directed edges) from object a_i to object a_j.

Note that this formulation allows for multiple ("parallel") directed edges from the same pair of source and sink nodes. Such a provision for parallel edges is important to accommodate in the modeling of practical signal processing systems. That is, practical dataflow graphs are technically directed *multigraphs* instead of pure directed graphs.

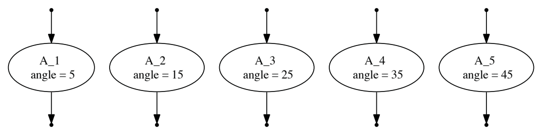


Fig. 2. Configuring an array of nodes with a PPP.

3.3. Parameter Propagation

An important feature to support in conjunction with topological patterns is a mechanism for structured *parameter propagation*, whereby any parameters associated with the vertices in a topological pattern can be set as a function of the vertex indices (i.e., indices associated with the underlying vertex ordering that is input to the pattern instance). For example, Fig. 2 shows an array of 5 actors identified as $A\_1, A\_2, \ldots, A\_5$, where each actor actor has a parameter *angle* associated with it that is an affine function of its index in the array. Such a parameter assignment can be implemented in a scalable, reusable, and explicitly-recognizable form as a designated PPP — in particular, a PPP for affine mappings of parameter values across ordered vertices. Such an affine PPP can find use in specifying elements of a *steering vector* corresponding to each sensor in a sensor array while estimating the direction of arrival of the received signal [24].

In terms of implementation in the DIF language, just as component attributes and topological patterns can be either user-defined or built-in, similarly commonly-used PPPs can be absorbed into the language as built-in PPPs, while users have the flexibility to incorporate specialized PPPs by linking their interpretation (propagation functionality) to segments of customized Java code.

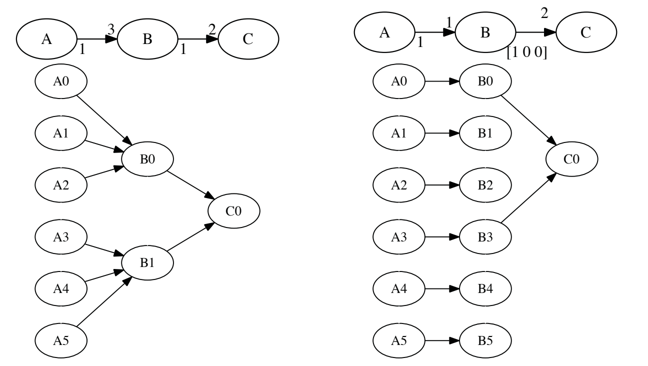
4. TOPOLOGICAL PATTERNS IN DIF

We extend the DIF language by supporting topological patterns as first class citizens in the modeling framework. These patterns can be defined as *built-in* patterns, which are recognized and processed through corresponding keywords in the language. To enable more flexible application of patterns, we also support declaring arbitrary (*user-defined*) patterns, whose associated graph construction functionality can be carried out through procedural language code (Java or C in the case of DIF) that is linked with the graph specification.

We have added, as built-in topological pattern specifiers, new keywords in DIF corresponding to topological patterns that are relatively common in signal processing systems. These keywords, such as `ring`, `parallel`, `merge`, `butterfly`, `broadcast`, and `chain`, allow specifying patterns explicitly as part of the `topology` block in a DIF specification. When declaring an instance of such a pattern, the designer must provide with an ordered sequence of vertices and an optional set of parameter values. The pattern construct, when parsed, generates the required edges, inserting the new edges into the graph that is being constructed. The pattern construct also configures the underlying nodes using the parameter propagation mechanism explained in Section 3.3.

A typical way to specify a sequence of nodes is through the use of DIF notation for representing nodes in an array. For example, for an array of 7 nodes specified as $A[7]$ (as in C, DIF arrays are indexed starting at 0), we can specify that 5 of its elements form a ring structure with the following DIF code

```
topology {
nodes = A[7];
edges = e0(A[0], A[1]), e1(A[5], A[6]),
ring(A[1:1:5]); }
```



(a) SDF graph to HSDF graph (b) CSDF graph to HSDF graph

```
topology {
nodes = A[6], B[2], C;
edges = merge(A[0:2], B[0]),
merge(A[3:5], B[1]), merge(B[0:1], C); }
```

(c) DIF `topology` block for HSDF graph in (a)

```
topology {
nodes = A[6], B[6], C;
edges = parallel(A[0:5], B[0:5]),
merge(B[0:3:3], C); }
```

(d) DIF `topology` block for HSDF graph in (b)

Fig. 3. A sample rate converter.

in the `topology` block. The argument `A[1:1:5]` to the construct `ring`, specifies an array of nodes starting from `A[1]`, ending at `A[5]`, and having an array index increment of 1. Note that, outside of the pattern instantiation construct, the nodes in the array `A` can be accessed by their indices to create edges that are not part of the ring pattern. Thus, one can flexibly embed patterns within arbitrary structures, including structures that contain other patterns.

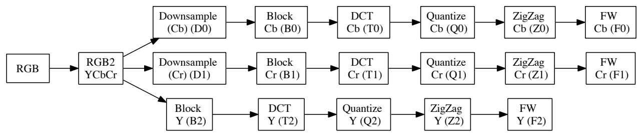
5. APPLICATIONS OF TOPOLOGICAL PATTERNS

As described earlier, we envision topological patterns to offer manifold advantages at various stages of the design flow from modeling to platform-specific implementation. In Sections 3 and 4, we have identified topological patterns in various DSP system specifications. In the following subsections, we examine other aspects of the design flow where topological patterns can be effectively used.

5.1. Topological Patterns for Representing HSDF Graphs

Many techniques devised for generating multiprocessor schedules from SDF graphs require that the given dataflow graph be transformed into an equivalent *homogeneous SDF* (*HSDF*) graph. An HSDF graph is an SDF graph in which every actor consumes (produces) a single token from (on) each input (output) port. Techniques for converting an SDF graph into equivalent (for scheduling purposes) HSDF graphs have been developed in [9]. Such techniques are useful because equivalent HSDF graphs can expose parallelism much more effectively compared to their more compact SDF counterparts.

Unfortunately, equivalent HSDF representations can scale very inefficiently — the size of an equivalent HSDF is in general not polynomially bounded in the size of the corresponding SDF graph [25]. Representing and porting such HSDF graphs becomes a cumbersome exercise. It also makes it difficult for a design tool to traverse the HSDF representation and make effective use of it within a reasonable amount of time. Topological patterns can help in this



```
topology {
nodes = RGB, RGB2YCbCr, D[2], B[3], T[3],
Q[3], Z[3], F[3];
edges = e0(RGB, RGB2YCbCr),
broadcast(RGB2YCbCr, D[0:1:1], B[2]),
chain(D[0], B[0], T[0], Q[0], Z[0], F[0]),
chain(D[1], B[1], T[1], Q[1], Z[1], F[1]),
chain(B[2], T[2], Q[2], Z[2], F[2]);}
```

Fig. 4. JPEG encoder and `topology` block in its DIF specification.

situation by providing concise representations to expose repetitive structures within HSDF representations.

For example, Fig. 3(a) shows an SDF graph that models a simple sample rate converter, and its equivalent HSDF graph (below). Here, actor B is a decimator with a decimation factor of 3. Fig. 3(c) shows the DIF specification of this HSDF graph using topological patterns. Fig. 3(b) and (d) show an equivalent CSDF graph model with its HSDF graph and associated topological-pattern-based DIF specification. In the CSDF representation, actor B provides a decimation by a factor of 3 by consuming input tokens on every firing while producing an output token only on every third firing, starting with the first firing. As this example illustrates, topological patterns can provide a concise and scalable representation of equivalent HSDF graph representations for SDF graphs.

Structured representations of HSDF graphs can also enable efficient tuning of HSDF graph representations in terms of application parameters. For example, for the dataflow graph in Figure 3(b), it can be observed that if the decimation factor of actor B is changed, then the DIF representation for the HSDF graph can be updated by simply changing the numeric arguments to the topological patterns used in its representation. In general, for a decimation factor of γ, the production rate of actor B in Fig. 3(b) is $[1\ 0\ 0\ \cdots\ 0]_{1\times\gamma}$ and the equivalent HSDF graph for this CSDF graph has the following specification.

```
topology {
nodes = A[2γ], B[2γ], C;
edges = parallel(A[0:2γ-1], B[0:2γ-1]),
merge(B[0:γ:γ], C); }
```

Thus, topological patterns provide streamlined representations that are concise, tunable, and scalable, and are particularly useful for complex graph structures, such as those found in equivalent HSDF graphs arising from multirate SDF models.

5.2. Exploring Implementation Trade-offs using Topological Patterns

Fig. 4 shows a JPEG encoder along with the `topology` block in its DIF specification. It effectively employs the broadcast and chain patterns in its representation. The JPEG compression algorithm downsamples both the chroma (C_b and C_r) components before processing them. Except this, all three components (both chroma and luma Y) are processed through functionally similar chains of blocks. The input pixels are grouped into blocks that are then transformed using the discrete cosine transform (DCT), quantized, and scanned in a zigzag order. In particular, the chroma components may be processed using shared functional modules as being clearly exposed by the `topology` block. Without the use of topological

JPEG Encoder	Throughput (samples /cycle)	FPGA Resource Utilization		
		Slices (out of 13696)	18kB BRAM	18x18 MULT
Non-shared	0.159	8070 (58%)	41	30
Shared	0.159	6088 (44%)	37	22

Table 1. Performance and resource utilization trade-offs for FPGA implementation of JPEG encoder.

patterns, this observation may not have been clear until the entire graph was carefully traced — for a design tool, this observation may go entirely unexploited because such high level structure can be difficult to extract automatically from unstructured specifications.

We make use of this potential for resource sharing — which is exposed explicitly at a high-level through the use of topological patterns — to develop a streamlined FPGA implementation. Awareness of the high level topological pattern in this application allows for systematic trade-off analysis between two design options — one with shared resources for chroma component processing and another without shared resources. An analysis of the high level dataflow specification suggests that downsampling of chroma components would ensure that the chain processing Y component is the bottleneck and hence, the throughput should remain unaffected even when the C_b and C_r components are processed using shared functional modules. Precise modeling of the shared-resource implementation of the JPEG encoder requires that the SDF design in Fig. 4 be converted into an equivalent CSDF design in which buffers between functional modules are duplicated and alternate buffers are used in successive schedule iterations. For more background on this form of CSDF-based structural modeling, we refer the reader to [26]. From inspection of the CSDF intermediate model, it can be reasoned that the buffer requirement would remain unchanged across both designs (shared- versus separate-resource). However, we expect that the shared-resource version of the JPEG encoder would result in a net reduction in BRAM utilization.

This analysis can be confirmed from the resource utilization and throughput for shared- and separate-resource JPEG encoder implementations on the Xilinx Virtex-II Pro FPGA, as shown in Table 1. The base clock rate for our experiments is $40\,\mathrm{MHz}$. Even though actor-level resource sharing is often avoided in FPGA implementation due to the relatively high costs of multiplexing and routing resources (e.g., see [27]), resource sharing for a subgraph in a dataflow representation can result in conservation of FPGA resources that overrides the multiplexing overhead. The shared-resource JPEG encoder uses less BRAM than the separate-resource version, which can be attributed to the shared DCT block. Also, the shared-resource version uses fewer (18×18) multiplier units by employing shared downsampling, DCT, and quantization modules. As expected — from the aforementioned bottleneck analysis — both versions of the JPEG encoder achieve the same throughput. In particular, the Y component remains as the system bottleneck even when the C_b and C_r components are processed using shared FPGA resources. Our experiments thus demonstrate concretely how topological patterns can provide a *formal path* from scalable application analysis to the systematic exploration of implementation trade-offs in the design and implementation of signal processing systems on a relevant target platform.

6. CONCLUSION

We have introduced the concept of topological patterns, which can be used to identify and concisely iterate across arbitrary structures in a dataflow application graph. We have presented a general formulation of topological patterns as well as its interpretation in the context of dataflow modeling. We have shown how the types of flowgraph substructures that are pervasive in DSP application domain can be effectively represented in terms of topological patterns, and thereby used to generate compact, scalable application representations. We have also shown how an underlying design tool can exploit a high-level application specification consisting of topological patterns in various aspects of the design flow. In particular, we have demonstrated the efficacy of topological patterns in dataflow graph analysis, concise and scalable representation of HSDF graphs, and exploring implementation specific trade-offs. Automating the application and integration of topological patterns and pattern-specific analysis techniques is a useful direction for further investigation.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] J. L. Pino and K. Kalbasi, "Cosimulating synchronous DSP applications with analog RF circuits," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1998, vol. 2, pp. 1710–1714.

[2] H. Andrade and S. Kovner, "Software synthesis from dataflow models for embedded software design in the G programming language and the LabVIEW development environment," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, November 1998.

[3] 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.

[4] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubhr, A. Deyhle, A. Hadert, and J. Teich, "A SystemC-based design methodology for digital signal processing systems," *EURASIP Journal on Embedded Systems*, vol. 2007, pp. Article ID 47580, 22 pages, 2007.

[5] A. V. Oppenheim and R. W. Schafer, *Discrete-Time Signal Processing*, Prentice-Hall, Inc., 1989.

[6] S. Y. Kung, *VLSI Array processors*, Prentice Hall, 1988.

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

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

[9] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous dataflow programs for digital signal processing," *IEEE Transactions on Computers*, February 1987.

[10] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete, "Cyclo-static dataflow," *IEEE Transactions on Signal Processing*, vol. 44, no. 2, pp. 397–408, February 1996.

[11] J. T. Buck, *Scheduling dynamic dataflow graphs with bounded memory using the token flow model*, Ph.D. thesis, EECS Department, University of California, Berkeley, 1993.

[12] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling of DSP systems," in *Proceedings of the International Conference on Acoustics, Speech, and Signal Processing*, Istanbul, Turkey, June 2000, pp. 1948–1951.

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

[14] I. M. Verbauwhede, C. J. Scheers, and J. M. Rabaey, "Specification and support for multidimensional DSP in the SILAGE language," in *IEEE International Conference on Acoustics, Speech, and Signal Processing*, April 1994, vol. 2, pp. II/473 –II/476.

[15] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*, Grenoble, France, 2002.

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

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

[18] I. Corretjer, C. Hsu, and S. S. Bhattacharyya, "Configuration and representation of large-scale dataflow graphs using the dataflow interchange format," in *Proceedings of the IEEE Workshop on Signal Processing Systems*, Banff, Canada, October 2006, pp. 10–15.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[20] E. A. Lee and D. G. Messerschmitt, *Digital Communication*, Kluwer Academic Publishers, 1988.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, MIT Press and McGraw-Hill, second edition, 2001.

[22] D. A. Huffman, "A method for the construction of minimum-redundancy codes," in *Proceedings of the IRE*, September 1952, pp. 1098–1101.

[23] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification*, John Wiley and Sons, Inc., second edition, 2000.

[24] S. Haykin, *Adaptive filter theory*, Prentice-Hall, Inc., 1996.

[25] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proceedings of the IEEE Asilomar Conference on Signals, Systems, and Computers*, Pacific Grove, California, November 1995, pp. 122–126 vol.1.

[26] S. S. Bhattacharyya, R. Leupers, and P. Marwedel, "Software synthesis and code generation for DSP," *IEEE Transactions on Circuits and Systems — II: Analog and Digital Signal Processing*, vol. 47, no. 9, pp. 849–875, September 2000.

[27] W. Sun, M. J. Wirthlin, and S. Neuendorffer, "FPGA pipeline synthesis design exploration using module selection and resource sharing," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 254–265, 2007.