In Proceedings of the International Symposium on System-on-Chip, pages 131-136,
Tampere, Finland, November 2006.

Dataflow Transformations in High-level DSP System Design

(Invited Paper)

Sankalita Saha, Sebastian Puthenpurayil and Shuvra S. Bhattacharyya
Department of Electrical and Computer Engineering
University of Maryland, College Park, MD, 20742, USA
Email: (ssaha, purayil, ssb)@umd.edu

***Abstract*— Dataflow graph transformations are important in many contexts of SoC implementation, particularly in the domain of signal processing. Most previous work on dataflow graph transformations has focused on synchronous dataflow and closely-related representations (e.g., see [1], [2], [3]). However, because SDF is limited to expressing static dataflow behaviors, modern SoC applications are often not fully amenable to SDF. In this paper, we discuss methods of modeling and transformation for more flexible forms of dataflow that are better suited to high-level design of modern signal processing systems.**

I. Introduction

Implementation of digital signal processing (DSP) applications on SoCs is a multi-faceted problem. Such systems generally involve combinations of programmable digital signal processors (PDSPs), microcontrollers, and application-specific hardware subsystems. Such powerful and heterogeneous platform capabilities along with increases in the complexity of the targeted applications make it challenging to deliver competitive design solutions within today's tight time-to-market constraints.

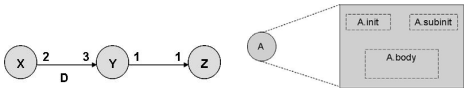
The typical design flow for such systems consists of several steps, each having high complexity and major ramifications for subsequent steps in the flow. Transformations provide an effective class of techniques for constraining subsequent steps in such a design flow towards more desirable solutions. These techniques involve taking a given description of the system, and deriving from it another description that is more suited for the targeted implementation constraints and objectives. Although the traditional emphasis for efficient embedded processor implementation has been on code-generation techniques and related compiler technology, high-level transformations have been gaining in importance in part because of their inherent portability and their boost in implementation quality when applied appropriately(e.g., see [4], [5]). Such transformations have been studied in various contexts, such as high level synthesis [6], synthesis of DSP software [3], and fault detection in parallel processing systems [7].

Our focus in this paper is on modeling the high level computational structure of signal processing applications, and transforming the resulting models to constrain subsequent steps in the design flow towards efficient implementations. Careful modeling of the application in terms of formal models of computation leads to more robust and modular designs, and exposes high level structure in the application that can be analyzed and transformed for effective optimization. Because of the high level of abstraction at which they operate, the transformations employed in this process have a large impact on key implementation metrics, including performance, power consumption, memory requirements, and overall cost.

The modeling and transformation approaches that we address in this paper are rooted in the methodology of coarse-grain dataflow modeling of signal processing applications. This form of dataflow is intuitive and efficient for design and implementation of signal processing systems, and is supported by a variety of commercial and research-oriented design tools (e.g., see [8]). Dataflow graph transformations for efficient implementation have been explored in the past; this earlier work has been oriented primarily to synchronous dataflow (SDF) and important special cases of SDF, especially homogenous synchronous dataflow. Parhi has presented an extensive overview of transformations in this context [2].

However, for complex signal processing systems, SDF representations suffer from lack of sufficient expressibility, especially for systems involving dynamic interactions between different computational modules. To address the limitations of SDF, more flexible dataflow modeling approaches have been proposed in recent years that allow for dynamic capabilities without excessively compromising the key properties of SDF — compile-time predictability and potential for rigorous optimization. Parameterized synchronous dataflow (PSDF) [9] and cyclo-static dataflow (CSDF) [10] are two such examples.

In this paper, we address transformation techniques for the more flexible classes of PSDF and CSDF representations. We also present the first in-depth analysis of parameterized cyclo-static dataflow (PCSDF) which is a model of computation that integrates features of PSDF and CSDF, and provides a powerful new form of dataflow modeling for DSP system design. We present new transformation and scheduling strategies for CSDF as well as PCSDF graphs. Our techniques and analysis are developed in a general fashion for these models. To demonstrate their properties and capabilities, we applied our techniques to an adaptive compression system for acoustic signals, which is a practical application that is not well-suited to conventional (static) dataflow transformation approaches. Our results demonstrate how appropriate transformations can be applied to high-level, dynamically-structured



(a) A simple SDF graph. (b) Outline of a PSDF subsystem.

Fig. 1.

dataflow graphs to derive optimized implementations.

II. Dataflow Graphs

Dataflow is one of the most widely-used modeling paradigms for DSP systems. In dataflow, a program is represented as a directed graph in which the vertices, (called *actors*), correspond to computational modules, and data values (called *tokens*) are passed between actors through FIFO queues that correspond to the edges of the graph. More specifically, the FIFO buffers associated with edges in a dataflow graph queue tokens, which are passed from the output of one actor to the input of another. When an actor is executed, it consumes a certain number of tokens from each of its inputs, and produces a certain number of tokens on each of its output edges.

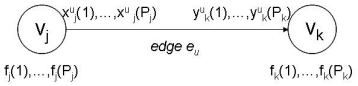
A. Synchronous Dataflow

In the context of DSP system design, the synchronous dataflow (SDF) model [1] is one of the most popular forms of dataflow. This popularity arises from the compile-time predictability of SDF, which results in a wide range of useful optimization possibilities (e.g., see [8]). In SDF, the number of tokens produced or consumed in one execution of an actor with respect to any of its input and output ports is constant. This property makes it possible to determine the execution order (*schedule*) of the actors, and the memory requirements of the FIFO buffers at compile time. Thus, SDF-based systems do not require the overhead of run-time scheduling, and have highly predictable run-time behavior. Figure 1(a) shows a simple SDF graph in which each edge is annotated with the number of tokens produced/consumed by its source/sink actor, and a unit delay on the edge connecting actors A and B (implemented as an initial token) is annotated with the symbol D.

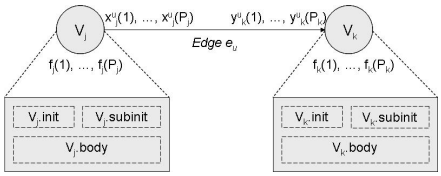
B. Parameterized Synchronous Dataflow

Parameterized dataflow is a meta-modeling technique that can be applied to a wide variety of specific dataflow models, such as synchronous dataflow (SDF) [9]. Parameterized dataflow provides for dynamic behavior by means of structured, dynamic parameter changes in the *base model* that it is applied to. Parameterized dataflow has been studied so far primarily in its application to SDF. The model of computation that results from integrating parameterized dataflow with SDF as the base model is called parameterized synchronous dataflow (PSDF) [9]. Useful formal analysis of PSDF graphs has been developed in [11].

A PSDF graph is composed of PSDF actors and PSDF edges. A PSDF actor is characterized by a set of parameters that can control the actor's functionality, including the actor's



(a) Example of a CSDF graph.



(b) Outline of a PCSDF subsystem.

Fig. 2.

dataflow behavior — that is, the numbers of tokens consumed and produced at its input and output ports. Similarly, a PSDF edge also has associated notions of parameterization and configuration. A PSDF subsystem consists of three distinct PSDF graphs — the $init$ graph, the $subinit$ graph and the $body$ graph. Intuitively, the body graph models the main functional behavior of the specification, whereas the init and subinit graphs control the behavior of the body graph by appropriately configuring the body graph parameters. Figure 1(b) shows a simple PSDF subsystem. Complete details of the syntax and semantics of PSDF modeling can be found in [9].

C. Parameterized Cyclo-static Dataflow (PCSDF)

One of the most popular extensions of SDF is cyclo-static dataflow (CSDF) [10]. In CSDF, token production and consumption can vary between actor firings as long as the variation forms a certain type of periodic pattern. Each component of this periodic pattern is called a *phase* of the actor. Actor behavior, including dataflow properties of the actor, can vary between phases, as long as the sequence of phases is periodic.

Formally, in CSDF, every actor v_j has an underlying execution sequence $f_j(1), f_j(2), ..., f_j(P_j)$ of length P_j, where each $f_j(i)$ is called a phase. Conceptually, the nth time actor v_j is fired, it executes the code of function $f_j((n-1) mod P_j + 1)$. As a consequence, in a CSDF graph production and consumption rates also follow periodic sequences. The amount of data (number of tokens) produced by actor v_j on edge e_u, is represented as a sequence of constant integers $[x_j^u(1), x_j^u(1), ..., x_j^u(P_j)]$. The nth time the actor is executed, it produces $x_j^u((n-1) mod P_j + 1)$ tokens on edge e_u. Representation of the amount of data consumed by actor v_k is completely analogous. The firing rule of a cyclo-static actor is evaluated as "true" (i.e., the actor is enabled for execution) for its nth firing if and only if every input edge e_u to actor v_k contains at least $y_k^u((n-1) mod P_k + 1)$ tokens. In Figure 2(a), the semantics of the cyclo-static dataflow model are illustrated.

Parameterizing CSDF graphs through the meta-modeling framework of parameterized dataflow creates a new class of graphs — PCSDF — that combines powerful optimization capabilities with strong expressibility properties. The syntax and semantics for PCSDF graphs are similar to that of PSDF

graphs. However, unlike a PSDF actor, a PCSDF actor's functionality as well as its *dataflow behavior* are not parameterized directly. Instead, they vary cyclically and it is this cyclic pattern that is parameterized.

Conceptually, there are two fundamental parameters involved in the dataflow properties of a PCSDF actor: the period of the cycle of phases, and the data rates (i.e., the rates of token production and consumption) associated with each phase. Values of these parameters must remain constant throughout any given iteration of the underlying (parameterized) CSDF graph; however, there is significant flexibility in reconfiguring the parameters between iterations. Dynamic behavior of a CSDF graph is therefore modeled by allowing dynamic parameterization of the period length, and of the individual phases associated with each actor. Thus, it is possible to have behaviors in which the sequence of phases is fixed for a single iteration, or for a group of successive iterations, while the sequences can generally vary between iterations. Figure 2(b) shows an example of a PCSDF subsystem.

III. Dataflow Clustering Transformations

The first step towards implementing a dataflow-based system is construction of a *schedule*, which may involve a statically constructed schedule, a dynamic scheduling mechanism, or a combination of both. For statically schedulable models, such as SDF and CSDF, static schedules can be employed through infinite iteration of *periodic schedules*. A periodic schedule is a finite sequence of actor invocations that invokes each actor at least once, does not deadlock, and produces no net change in the numbers of tokens that are queued in the FIFOs associated with the graph edges (the initial numbers of tokens on these FIFOs are given by the edge delays). Dataflow transformations may be applied before creating a schedule, as well as during the process of constructing a schedule. Of the various possible scheduling strategies our focus is on *looped schedules* [12] and *parameterized looped schedules* [9], [13], which construct schedules in terms of static and dynamic looping constructs, respectively. These types of schedules combine the advantages of efficient looping facilities in programmable digital signal processors [14], low complexity storage and manipulation of schedule information [8], [13], and potential for extensive analysis and optimization [8].

A form of transformation that is especially important in the derivation of schedules from high-level dataflow representations is *clustering*. A clustering transformation for a dataflow graph is one that replaces a set of multiple actors in the graph with a single actor, thereby constraining subsequent synthesis steps to view the chosen set of actors as a single "unit" for some relevant purpose. For example, the associated tasks may be constrained to execute on the same processor, or to be scheduled together without any interleaving from actors outside the cluster.

When clustering a set A of actors, characterizations for production and consumption rates of edges at the interface of a cluster are derived based on how much data is transferred with respect to the interfaces in one execution of the subsystem associated with A. A formal development of clustering for SDF graphs is provided in [15].

A. Clustering in PSDF Graphs

Since in a PSDF graph, the production and consumption rates of actors are variable and generally not known at compile time, it is neither possible to construct static schedules, nor develop exact characterizations of dataflow behavior at cluster interfaces. However, clustering is an effective tool for transforming PSDF graphs in situations where the characteristics of clusters can be expressed in terms of subsystem parameters. Fortunately, through careful analysis, this can be done for many practical PSDF systems.

One example of a useful clustering algorithm for PSDF graphs is the technique of parameterized acyclic pairwise grouping of adjacent nodes (*P-APGAN*) [9]. In P-APGAN, as in the SDF-based APGAN algorithm [16] from which it is derived, hierarchies of clusters are constructed in a bottom-up fashion by repeatedly selecting and clustering adjacent pairs of actors. The selection process is performed in a heuristic fashion based on clustering cost functions that are geared towards the key implementation objectives. For each clustered subgraph, a looped schedule is generated symbolically in terms of the dynamic parameters associated with the subgraph. This allows the subgraph schedule to adapt at run-time in correspondence with changes in the parameter values. After the cluster hierarchy is constructed, a schedule for the overall PSDF graph is derived by recursive traversal of the cluster hierarchy and substations of the schedules associated with each clustered subgraph [9].

B. Clustering in PCSDF Graphs

In this section, we introduce in detail a clustering transformation for the more complex class of PCSDF graphs. We develop this transformation by first deriving *multirate clustering* for CSDF graphs. Multirate clustering is useful for decomposing complex, high-level dataflow designs for scheduling, and has previously been studied primarily in the context of SDF graphs ([15]).

We define a production/consumption rate tuple $(x_j^u(1), x_j^u(2), ..., x_j^u(X_j))$ for an edge e^u of a CSDF actor A_j with X_j phases as a *data rate signature*(DRS) for the edge. We break down each such DRS into repeating sub-regions or sub-DRSs $(s_j^u(1), s_j^u(2), ..., s_j^u(m))$ such that each $s_j^u(i)$ covers exactly k tokens. We denote the new DRS resulting from the combination of the sub-DRSs by DRS_{phased}. Thus, DRS_{phased} is a DRS that generally consists of fewer phases compared to the original DRS. Furthermore, the data rate value associated with every phase is a constant k, which we call the $subDRSrate$. This grouping of phases can be viewed as imposing an SDF abstraction of finest possible granularity on top of the underlying CSDF input/output port .

Consider for example the edge e shown in figure 3 with DRS $(1, 0, 1, 2, 0, 1, 1, 0, 2)$. This can be decomposed into (s_1, s_2, s_3, s_4) where $s_1 = (1, 0, 1)$, $s_2 = (2, 0)$, $s_3 = (1, 1, 0)$

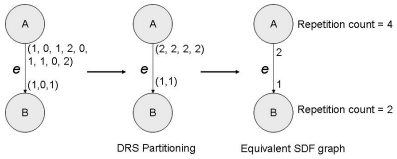


Fig. 3. Example of DRS partitioning for a 2-actor CSDF graph.

and $s_4 = (2)$. Note that such a decomposition is not unique because any trailing 0 of an intermediate s_i can be moved to the beginning of s_{i+1}. For conciseness, we impose the restriction that trailing 0s with respect to any intermediate s_i are always included in the s_i itself. Exploring the broader design space allowed by more flexible placement of such trailing 0s is a topic for further study.

As implied above, such a decomposition effectively creates an SDF abstraction of the input/output port that the DRS is associated with, but at a finer granularity compared to the trivial SDF abstraction that results from collapsing all the phases into one monolithic actor invocation. However, in general it is not always possible to find a valid decomposition that is of finer granularity than the trivial one — in these cases, the decomposition technique here degenerates to the trivial SDF abstraction. For the case of binary CSDF [17] (a binary CSDF graph has only 1s and 0s in the DRSs), it is generally always possible to obtain finer granularity decompositions compared to the trivial form (that is, whenever the sum of token rates in a DRS exceeds 1). We call the general process of creating sub-DRSs "DRS Partitioning" (DRSP).

Once a valid DRSP of the graph is formed, we obtain a resulting SDF graph, that is, a graph with constant, scalar production and consumption rates (Figure 3). Thus, now various SDF-oriented strategies for clustering and scheduling, such as APGAN, can be considered. However, after scheduling, a reverse decomposition of production and consumption rates needs to be performed for actors that have undergone DRSP. We call this reverse decomposition process *reverse DRSP* (RDRSP). Based on these concepts, a top level transformation for CSDF graphs can be expressed algorithmically as follows.

Algorithm III.1: DRSP TRANSFORMATION(ϕ_G)

$\phi_{DRSP} \leftarrow DRSP(\phi_G)$
$Schedule\ S_{DRS} \leftarrow APGAN(\phi_{DRSP})$
$Schedule\ S \leftarrow RDRSP(S_{DRS})$
return (S)

We now present a simple algorithm for DRSP for a single DRS. In the formulation of the algorithm, we make use of two observations: (1) The $subDRSrate$ for a valid sub-DRS partitioning of a DRS S cannot be less than the maximum production/consumption rate in S and (2) The $subDRSrate$ for a valid sub-DRS partitioning of a DRS S is a factor of the sum of the production/consumption rates of S. Let the original DRS be stored in an array S and have a length N. The algorithm takes S as input, examines whether DRSP can be performed, and if so returns DRS_{phased}.

Algorithm III.2: DRSP SINGLE($DRS\ S, N$)

$max \leftarrow findmax(S, N)$
$sum \leftarrow findsum(S, N)$
$(factors, m) \leftarrow factors(sum)$
for $i \leftarrow 0$ **to** m
  do { **if** $factors(i) >= max$
    do { $start \leftarrow factors(i)$
      break
for $f \leftarrow start$ **to** $factors(m)$
  do { $(DRS_{phased}, phases) \leftarrow partition(S, f)$
    if $DRS_{phased}\ != NULL$
    break
return (DRS_{phased})

In this pseudocode block, the function *findmax(A,n)* finds the maximum element of an array A of length n; *findsum(A,n)* finds the sum of the elements of an array A of length n; and *factors(M)* returns an array containing the factors of M. The function *partition(A,s)* takes as input an array A; determines whether it is possible to split the array into sub-sets, each of whose sum of elements is s; and if so, returns an array containing $phases$ elements, each having value s. Note that $phases$ can be 1, denoting the trivial SDF abstraction. If such a split of the input array cannot be performed, *partition(A,s)* returns an empty array. The complexity of this algorithm is dictated by the for loop, which attempts to find a valid partition. The complexity of the computation performed by this loop is $O(m)$ for m factors. The complexity is also dictated by the function *partition(A,s)* which is $O(n)$, where n is length of A. The overall complexity is therefore $O(mn)$.

The PCSDF graph transformation that we develop for clustering and for construction of looped schedules builds on the CSDF graph transformation introduced above. The clustering is performed by extracting a CSDF graph whenever dataflow-related parameter values change at run-time, and analyzing and performing clustering on this CSDF graph for proper operation until the next time the parameter values change. Building on this framework to develop efficient quasi-static scheduling for PSDF graphs — in particular, scheduling that employs symbolic loop control methods as P-APGAN does — is a promising and useful direction for further work.

IV. Experiments and Results

To demonstrate the capabilities of the transformation techniques discussed in the sections above, we apply our methods to a data compression algorithm for acoustic signals. The application inherently involves dynamic production and consumption rates. However, for comparison purposes, a static version was derived that lacks the flexibility and robustness of the original design.

A. Acoustic Data Compression

The compression algorithm that we studied is an LPC (linear predictive coding) based acoustic data compression (ADC). The basic components of LPC consist of framing, predictor

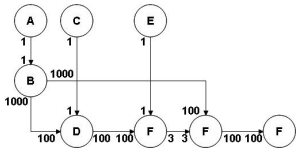


Fig. 4. Synchronous dataflow model of LPC.

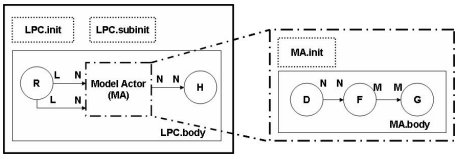


Fig. 5. Parameterized dataflow model of LPC.

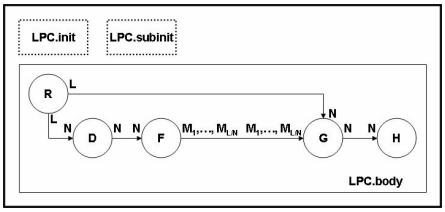


Fig. 6. Parameterized cyclo-static dataflow model of LPC.

coefficients, and coding. For the process of framing, the input signal for a particular duration of time contains L number of samples, and this set of L samples is divided into frames each of size N. Each frame is subjected to the linear predictor, which generates predictor coefficients. These coefficients then are used to determine the predicted value of the input signal. The prediction error and its coefficients are quantized, and this quantized data is the compressed data. To retrieve the original data in a lossless manner, a corresponding decompression algorithm can be applied in which the predictor coefficients and error data are processed.

B. SDF-based modeling and transformation of ADC

Given a segment of input data of size L, the framing process computes the optimum frame size N of the input data. Each frame is then passed to a linear predictor, which computes the predictor coefficients. The number of coefficients depends on the order M. In the SDF version of application, N and M values of the predictor are fixed and known beforehand. A detailed block diagram of the SDF model is given in Figure 4. The computational blocks are represented by the actors A *to* H. Actor A sets the sample size L of the input data, actor B reads a segment of input data and stores it in a buffer, actor D implements Fast Fourier transform (FFT) operation on the input samples, actor C calculates the size of the required FFT operation based on the frame size of the samples, actor E determines the fixed model order for the frames, actor F performs LU decomposition for determining the predictor coefficients, actor G generates the error on samples using AR model, and actor H implements Huffman coding on the error samples.

The clustering transformation and scheduling approach applied in our experiments to this SDF graph are based on the APGAN strategy [16]. For $L = 100$ and $N = 10$, this produced the schedule $AB(10\ CDEFGH)$. This is expressed in conventional looped schedule notation in which parenthesized terms correspond to schedule loops. The first component in a parenthesized term represents the iteration count of the associated loop (10 in this case), and the remaining components collectively give the sub-schedule that is to be iterated (in this case, $CDEFGH$ is to be iterated 10 times in succession).

C. PSDF-based modeling and transformation of ADC

A block diagram of the PSDF model is given in Figure 5. In this, LPC.init sets the segment size L while LPC.subinit sets the frame size N. LPC.body implements the body graph — i.e., the core, parameterized computation of the compression system. Actor R reads a segment of input data and stores it in a buffer while Model Actor (*MA*) is a hierarchical actor that implements the model order for each frame. More specifically, *MA.init* sets the model order for each frame and *MA.body* is an associated body graph that contains actors D, F, G. Actors D, F, G and H have the same functionality as described in the SDF-based model.

The P-APGAN scheduling strategy [9] was applied manually to derive the schedule the for the above PSDF model of the LPC system and thus did not involve any call to APGAN as in the DRSP transformation for CSDF. The resulting schedule is $R(L/N\ MA)H$, which further decomposed to $R(L/N\ DFG)H$ when the hierarchical actor MA was expanded. Note that this is a parameterized schedule with the iteration count quotient L/N varying dynamically as the application executes. As the underlying parameter values L and N change at run-time, the parameterized schedule naturally adjusts the actor execution sequence to be consistent with the new parameter values.

D. PCSDF-based modeling and transformation of ADC

The PSDF modeling of the LPC system involves a hierarchical actor MA. Since, a PSDF subsystem generally involves an *init* graph or a *subinit* graph (or both), such hierarchical subsystems can lead to noticeable overheads if they are non-trivial (as it turned out in our case). However, a careful analysis of the targeted ADC system reveals that the dynamic behavior of actor MA is a behavior that varies in a cyclic fashion, and thus one level of PSDF hierarchy can be removed by modeling the system using PCSDF. The block diagram of the resultant PCSDF-based model is given in Figure 6. In this model, the *LPC.subinit* graph gives the model order M for each frame. In the *LPC.body* graph for the actor F, the data count (which is configured by the *LPC.subinit* graph of the *LPC*subsystem) varies for each iteration. LPC.init sets the segment size L, LPC.subinit sets the frame size N, and the model order M_1 *to* $M_{L/N}$ for each frame. The LPC.body implements the body graph and contains actors R, F, D, G and H (explained in sections IV-B, IV-C).

TABLE I

CODE AND DATA SIZE COMPARISON.

TYPE	SDF	PSDF	PCSDF
Code Size (bytes)	74596	77476	90724
Data Size (bytes)	123152	133367	123332

TABLE II

EXECUTION CLOCK CYCLES COMPARISON.

No. of Data Samples	No. of clock cycles		
	SDF	PSDF	PCSDF
50	2519928	2299943	22935143
100	5166166	113232300	113132476
200	10589544	426023069	424780511
300	21314321	941318880	1122343748

The transformation of the PCSDF graph was performed by extracting a CSDF version at run time, and then applying the DRSP transformation to this CSDF version. This process can be repeated indefinitely for subsequent changes in parameter values initiated by the application. The particular values that we have used in our experiments for CSDF graph extraction and subsequent transformation are $L = 100$ and $N = 36$, respectively. The resulting schedule is $R(2\ DFGH)$.

E. Results

The three versions of ADC that we experimented with were implemented using the Texas Instrument's Code Composer Studio (version 2) with the TMSC320c64xx programmable digital signal processor as the target platform. Table I shows the resulting comparisons among the code sizes and data sizes for the SDF-, PSDF-, and PCSDF-based implementations. As one can observe from the table, the SDF-based implementation yields the least code and data size. However, this comes at the cost of significantly reduced flexibility in the system.

Allowing dynamic reconfigurability leads to run-time overheads, as shown by our performance results (Table II). The higher execution times for the PSDF and PCSDF models compared to SDF is due to the run-time overhead of the scheduling strategies, which is not present for the SDF model. The PCSDF and PSDF systems show comparable performance despite the overhead the PCSDF system incurs due to run-time invocation of APGAN. This is due to a two-fold effect: the PSDF-based system incurs overheads due to repeated invocation of the init graph of the MA actor, and the quasi-static schedule for PSDF involves 2 for loops compared to the schedule for the PCSDF system that has only 1 for (all of which depend on the number of samples L). However, the PCSDF system yields significantly less data size compared to PSDF. The higher code size of the PCSDF system is due to the scheduling code overhead, which is not present in PSDF modeling. This result provides strong motivation to explore quasi-static scheduling for PCSDF graphs, for example, by building on the quasi-static analysis framework that is available for PSDF [9]. This is an important direction for future work. With successful quasi-static scheduling, a PCSDF-based design will retain data memory efficiency, while benefiting from improved code size and performance.

V. CONCLUSIONS

In this paper, we have explored high-level transformation techniques for dataflow-based design of signal processing applications. Traditional transformations have been primarily limited to SDF modeling. However, SDF modeling is too statically-oriented to fully capture the high-level structure of modern signal processing systems. Of the various modeling techniques that support SDF but with enhanced capabilities to allow such dynamic behavior, we have explored two complementary methods, PSDF and CSDF. In addition, we have presented a transformation framework for parameterized cyclo-static dataflow, which is a powerful modeling paradigm that integrates the advantages of CSDF and PSDF.

REFERENCES

[1] E. A. Lee and D. G. Messerschmitt, "Synchronous dataflow," *Proc. of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[2] K. K. Parhi, "High-level algorithm and architecture transformations for DSP synthesis," *Jnl. of VLSI Signal Processing*, 1995.

[3] V. Zivojnovic, S. Ritz, and H. Meyr, "High performance DSP software using data-flow graph transformations," in *Proc. of the IEEE Asilomar Conf. on Signals, Systems, and Computers*, 1994.

[4] B. Franke and M. O'Boyle, "An empirical evaluation of high level transformations for embedded processors," in *Proc. of CASES*, Nov. 2001.

[5] P. Marwedel, "Embedded software: how to make it efficient," in *Proc. of the Euromico Symp. on Digital System Design*, 2002.

[6] R. Camposano, "From behavior to structure: high-level synthesis," in *Proc. of IEEE Design and Test in Europe*, Oct. 1990, pp. 8–19.

[7] C. Gong, R. Melhem, and R. Gupta, "Loop transformations for fault detection in regular loops on massively parallel system," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 12, 1996.

[8] P. K. Murthy and S. S. Bhattacharyya, *Memory management for synthesis of DSP software*. CRC Press, 2006.

[9] B. Bhattacharya and S. S. Bhattacharyya, "Parameterized dataflow modeling for DSP systems," *IEEE Trans. on Signal Processing*, vol. 49, no. 10, pp. 2408–2421, 2001.

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

[11] S. Neuendorffer and E. Lee, "Hierarchical reconfiguration of dataflow models," in *Proc. of the Intl. Conf. on Formal Methods and Models for Codesign*, June 2004.

[12] S. S. Bhattacharyya and E. A. Lee, "Looped schedules for dataflow descriptions of multirate signal processing algorithms," *Jnl. of Formal Methods in System Design*, pp. 183–205, Dec. 1994.

[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 *Proc. of the Intl. Conf. on Application Specific Systems, Architectures, and Processors*, Steamboat Springs, Colorado, Sept. 2006.

[14] P. Lapsley, J. Bier, A. Shoham, and E. A. Lee, *DSP Processor Fundamentals*. Berkeley Design Technology, Inc., 1994.

[15] J. L. Pino, S. S. Bhattacharyya, and E. A. Lee, "A hierarchical multiprocessor scheduling system for DSP applications," in *Proc. of the IEEE Asilomar Conf. on Signals, Systems, and Computers*, Nov. 1995, pp. 122–126 vol.1.

[16] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, "APGAN and RPMC: complementary heuristics for translating dsp block diagrams into efficient software implementations," *Jnl. of Design Automation for Embedded Systems*, vol. 2, no. 1, pp. 33–60, 1997.

[17] E. F. Deprettere, T. Stefanov, S. S. Bhattacharyya, and M. Sen, "Affine nested loop programs and their binary cyclo-static dataflow counterparts," in *Proc. of the Intl. Conf. on Application Specific Systems, Architectures, and Processors*, Steamboat Springs, Colorado, Sept. 2006.