In the Proceedings of the International Workshop on Rapid System Prototyping,
June, 2000. Paris, France.

Quasi-static Scheduling of Reconfigurable Dataflow Graphs for DSP Systems

Bishnupriya Bhattacharya and *Shuvra S. Bhattacharyya*,
Dept. of Electrical and Computer Engineering, and Institute for Advanced Computer Studies,
University of Maryland, College Park, MD 20742, USA
bpriya@cadence.com (The author is now with Cadence Design Systems, Inc.), *ssb@eng.umd.edu*

Abstract

Dataflow programming has proven to be popular for representing applications in rapid prototyping tools for digital signal processing (DSP); however, existing dataflow design tools are limited in their ability to effectively handle dynamic application behavior. In this paper, we develop efficient quasi-static scheduling techniques for a broad class of dynamically-reconfigurable dataflow specifications. We use a CD to DAT sample rate conversion system and a speech compression application to illustrate the efficacy of our scheduling techniques in real life DSP systems.

1 Background and motivation

Over the past several years, programmable digital signal processors have proven to be popular in rapid prototyping environments for DSP. In this context, software synthesis techniques for producing target code from DSP applications specified in the synchronous dataflow (SDF) model [8] have been studied extensively. SDF is a subset of the dataflow model of computation in which computational blocks (actors) are interconnected by edges (buffers) that represent sequences of data values. In SDF, the number of data values (*tokens*) produced and consumed by each actor is fixed and known at compile time. Fig. 1(a) shows a simple SDF graph. Each edge is annotated with the number of tokens produced (consumed) by its source (sink) actor, and the D on the edge from actor A to actor B specifies a unit *delay.* Each unit of delay is implemented as an initial token on the edge. Given an SDF edge e, the source actor, sink actor, and delay of e are denoted by $src(e)$, $snk(e)$, and $d(e)$. Also, $p(e)$ and $c(e)$ denote the number of tokens produced onto e by $src(e)$ and consumed from e by $snk(e)$.

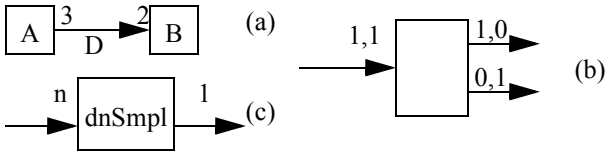

Figure 1: Dataflow actors and edges.

The static token flow pattern of SDF provides opportunities for thorough optimization, and indeed effective optimization techniques have been developed for data memory minimization [1], joint minimization of code and data [4], as well as a variety of other objectives. Many DSP block-diagram programming environments like [5] use the *threading* model to compile a SDF graph. The first step is to construct a *valid schedule* — a finite sequence of actor invocations that fires each actor at least once, does not deadlock, and produces no net change in the number of tokens queued on each edge. Corresponding to each actor in the schedule, a code block is instantiated that is obtained from a library of predefined actors. The resulting sequence of code blocks is encapsulated within an infinite loop to generate a software implementation of the SDF graph.

SDF graphs for which valid schedules exist are called *consistent* SDF graphs. In [8], efficient algorithms are presented to determine whether or not a given SDF graph is consistent, and to compute the minimum number of times that each actor must be fired in a valid schedule. These minimum numbers of firings are represented by a vector $\mathbf{q}_G$, indexed by the actors in G (the subscript is suppressed if G is understood), and can be derived by finding the minimum positive integer solution to the *balance equations* for G, which specify that $\mathbf{q}$ must satisfy $\mathbf{q}(src(e)) \times p(e) = \mathbf{q}(snk(e)) \times c(e)$, for every edge e in G. The vector $\mathbf{q}$, when it exists, is called the *repetitions vector* of G. A schedule S for G is a *minimal periodic schedule* if it invokes each actor A exactly $\mathbf{q}_G(A)$ times.

For Fig. 1(a), $\mathbf{q}(A, B) = (2, 3)$, and one minimal periodic, valid schedule is $A(2B)AB$. Here, a parenthesized term $(nS_1S_2\ldots S_k)$ specifies n successive firings of the "subschedule" $S_1S_2\ldots S_k$, and such a term can be translated into a loop in the target code. Each such parenthesized term $(nS_1S_2\ldots S_k)$ is called a *schedule loop*. A *looped schedule* is a finite sequence $V_1V_2\ldots V_k$, where each V_i is either an actor or a schedule loop. Given a looped schedule S, it is called a *single appearance schedule* if each actor in G appears only once in the schedule. A single appearance schedule minimizes code size for inline code generation.

Given a connected, consistent SDF graph $G = (V, E)$, and a subset of actors $Z \subseteq V$, the *repetition count* of Z is defined as $q_G(Z) \equiv gcd(\{\mathbf{q}_G(A) | (A \in Z)\})$, where *gcd* denotes the greatest common divisor. The concept of *clustering* in an SDF graph [3] is illustrated in Fig. 2(a). Here $subgraph(\{B, C\})$ is clustered into the hierarchical node Ω. Each input arc e to the clustered subgraph $subgraph(Z)$ is replaced by an arc $\bar{e}$ having $p(\bar{e}) = p(e)$, and $c(\bar{e}) = c(e) \times ((\mathbf{q}_G(snk(e)))/(q_G(Z)))$, the number of samples consumed from e in one invocation of subgraph Z. Similarly, each output arc e is replaced by $\bar{e}$ such that $c(\bar{e}) = c(e)$, and

$$p(\bar{e}) = p(e) \times ((\mathbf{q}_G(src(e)))/(q_G(Z))).$$

In Fig. 2(a), $\mathbf{q}_G(A, B, C, D) = (3, 30, 20, 2)$, and thus $q_G(\{B, C\}) = 10$.

This clustering technique is used in a scheduling algorithm developed for acyclic SDF graphs called *APGAN* (*Acyclic Pairwise Grouping of Adjacent Nodes*), geared towards joint code and data minimization objectives for software synthesis purposes. Given a consistent, acyclic SDF graph as input, APGAN clusters a pair of adjacent vertices in each step, producing a minimal periodic, single appearance looped schedule that is shown to be optimal for a certain class of SDF graphs [4].

As described above, the primary benefits that SDF provides are static scheduling and opportunities for thorough optimization, leading to a high degree of compile-time predictability. Although an important class of useful DSP applications can be modeled efficiently in SDF [5, 8], its expressive power is limited to fully static applications. Consequently, many extensions to the SDF model have been proposed, where the objective is to accommodate a broader range of applications, while maintaining a significant part of the compile-time predictability of SDF, and possibly, allowing representations that expose more optimization opportunities to a compiler. *Cyclo-static dataflow* (CSDF), is one example of such an extension of SDF. In CSDF, token production and consumption can vary between actor firings as long as the variation forms a certain type of periodic pattern [7]. Fig. 1(b) shows a CSDF actor that consumes one token on its input, and produces tokens according to the periodic pattern 1, 0, 1, 0, … (one token produced on the first invocation, none on the second, and so on) on one output edge, and according to the complementary periodic pattern 0, 1, 0, 1, … on the other output edge. A general CSDF graph can be compiled as a cyclic pattern of pure SDF graphs, and static periodic schedules can be constructed in this manner.

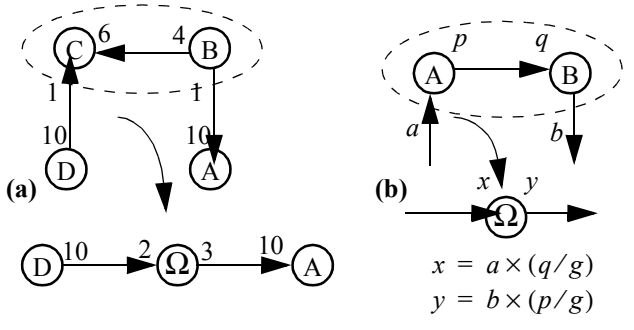

Figure 2: Examples of clustering.

In [2], we have introduced parameterized dataflow as a meta-modeling technique that can be applied to a wide range of dataflow models to significantly increase their expressive power. Our parameterization concepts can be incorporated into any dataflow model in which there is a notion of a graph *iteration*. For example, a minimal periodic schedule is a natural notion of a graph iteration in SDF, and CSDF. [2] develops the precise semantics of parameterized synchronous dataflow (PSDF), and demonstrates that data-dependent, dynamically re-configurable DSP systems can be modeled in a natural and intuitive fashion in PSDF.

In this paper, we develop efficient *quasi-static* scheduling techniques (techniques that apply compile time analysis to significantly reduce the amount of run-time scheduling involved) for a broad class of PSDF specifications in the software synthesis context. A PSDF quasi-static schedule is generated at compile-time, but may contain code for performing some data-dependent, scheduling-related computations at run-time. Thus, achieving a good balance in the trade-off between expressive power and compile-time predictability for PSDF is what motivates our quasi-static scheduling techniques.

2 Parameterized synchronous dataflow

In the PSDF model, a DSP application will typically be represented as a *PSDF subsystem* Φ, that is made up of three *PSDF graphs* — the *init* graph Φ_i, the *subinit* graph Φ_s, and the *body* graph Φ_b. A *set of parameters* are provided to control the behavior of the subsystem. In most cases, the subsystem parameters will be directly derived from the parameters of the application algorithm, e.g., in a block adaptive filtering system, the step size and the block size emerge as natural subsystem parameters. Intuitively, in a subsystem, the body graph is used to model its dataflow behavior, while the init and subinit graphs are responsible for configuring subsystem parameter values, thus controlling the body graph behavior.

A PSDF graph is a dataflow graph composed of PSDF actors and PSDF edges. A PSDF actor A is also characterized by a set of parameters ($params(A)$), that can control both the functional behavior as well as the dataflow behavior (number of tokens consumed and produced) at its input and output *ports*. An application programmer determines a configuration for A by assigning values to $params(A)$, where an actor parameter is assigned a fixed, static value or

a subsystem parameter value. In the latter case, it is possible for the associated init and subinit graphs to dynamically modify the subsystem parameter value, thus dynamically re-configuring the behavior of the PSDF actor. Fig. 1(c) shows a PSDF *downsampler* actor with two parameters {*factor*,*phase*}, where *factor* represents the decimation ratio, and *phase* denotes the index of the input token transferred to the output. Here, n is a subsystem parameter and *factor* has been assigned the value of n, giving rise to a dynamically re-configurable decimation ratio.

A PSDF subsystem Φ can be embedded within a "parent" PSDF graph abstracted as a *hierarchical PSDF actor* H, and we say that $H = subsystem(\Phi)$. In such a scenario, Φ can participate in dataflow communication with parent graph actors at its *interface ports* [2]. Φ_i does not participate in this dataflow; Φ_s may only accept dataflow inputs; while Φ_b both accepts dataflow inputs and produces dataflow outputs. The PSDF operational semantics [2] specify that Φ_i is invoked once at the beginning of each (minimal periodic) invocation of the hierarchical parent graph in which Φ is embedded; Φ_s is invoked at the beginning of each invocation of Φ; and Φ_b is invoked after each invocation of Φ_s.

Consistency issues in PSDF are based on the principle of *local SDF scheduling*, which allows every PSDF graph to assume the configuration of an SDF graph on each invocation. This ensures that a schedule for a PSDF graph can be constructed as a dynamically re-configurable SDF schedule. Local SDF scheduling leads to a set of *local synchrony* constraints for PSDF graphs and PSDF subsystems that need to be satisfied for consistent specifications [2].

A detailed discussion of the concepts introduced here can be found in [2], which also shows that the hierarchical, parameterized representation of PSDF supports increased design modularity (e.g., by naturally consolidating distinct actors, in some cases, into different configurations of the same actor), and thus, leads to increased design reuse in block diagram DSP design environments.

3 Parameterized looped schedules

In this section, we introduce some terminology for PSDF quasi-static schedules that is based on corresponding concepts for SDF graphs (Section 1). Given a PSDF graph G, a *parameterized schedule loop* L represents successive repetitions of an invocation sequence $T_1 T_2 \ldots T_m$, where each T_i is either a leaf actor in G, or a hierarchical actor in G, or another parameterized schedule loop. However, unlike SDF, the *iteration count* (number of repetitions of the invocation sequence) of the schedule loop, denoted as $loopcnt(L)$ is no longer an integer; rather, it is, in general, a symbolic expression consisting of integer constants, parameters, and compiler-generated variables. If T_i is a hierarchical actor in G representing subsystem Φ, then in order to compute a schedule for G, the (interface) token flow of T_i is necessary. This is obtained by pre-computing a parameterized looped schedule (defined shortly) for Φ_s and Φ_b, and appropriately evaluating the token flow at the interface ports of Φ [2]. An occurrence of T_i in L can be replaced by $(VLS_s)(S_s)(VLS_b)(S_b)$, where S_s and S_b represent parameterized looped schedules for the subinit graph of Φ and the body graph of Φ respectively. VLS_s and VLS_b consists of code that verifies local synchrony constraints that have to be satisfied by Φ_s and Φ_b for local synchrony of subsystem Φ.

Suppose that G is a PSDF graph that contains m hierarchical actors $H_1, H_2, \ldots, H_m$. A *parameterized looped schedule* S for G consists of three parts $S \equiv (S_1, S_2, \ldots, S_m)(preamble_S)(body_S)$, where each S_i represents a parameterized looped schedule for the init graph of $subsystem(H_i)$. Thus, in accordance with the PSDF operational semantics, the first part of the schedule for G, called the *initChild* phase of S, consists of successive invocations of the parameterized looped schedules of the init graphs of the child subsystems of G. The third part of the schedule $body_S$, called the *body* of S, is a sequence $V_1 V_2 \ldots V_k$, where each V_i is either an actor (leaf or hierarchical) in G or a parameterized schedule loop. The second part of the schedule $preamble_S$, called the *preamble* of S, consists of code that configures $loopcnt(L)$ of each schedule loop L that appears in $body_S$ by defining in a proper order every compiler-generated variable used in the symbolic expression of $loopcnt(L)$. In addition, the preamble code includes conditionals for checking local synchrony consistency of G. A parameterized looped schedule S for a PSDF graph G is a *single-appearance schedule* if each actor in G appears only once in $body_S$.

4 Quasi-static scheduling

The basic step in PSDF quasi-static scheduling is to determine the body ($body_S$) of the parameterized looped schedule S for a PSDF graph G. For acyclic graphs, we have developed an extension of the APGAN scheduling technique, called *P-APGAN* for *parameterized APGAN*.

In the APGAN technique for SDF graphs, a cluster hierarchy is constructed by clustering exactly two adjacent vertices at each step. At each clustering step, an adjacent pair is chosen for clustering, if clustering it does not introduce a cycle in the resulting graph, and its repetition count is greater than or equal to that of all other adjacent pairs that do not introduce cycles. Given an acyclic PSDF graph, it is not in general possible to select an adjacent pair of actors for clustering based on the maximum value of the repetition count of the cluster, as the repetition count can only be obtained symbolically. We use a heuristic that gives priority to *sin-*

gle-rate edges (edges for which it is statically known that $p(e)$ is equal to $c(e)$, but they are not necessarily known to be constant), *SDF edges* (edges for which $p(e)$ and $c(e)$ are constant and known at compile time), and *gcd-known edges* (edges for which the *gcd* of $p(e)$ and $c(e)$ is statically known, possibly through user assertions), in that order, with the goal of minimizing code size and run-time computations in the generated quasi-static schedule.

The schedule for the subgraph corresponding to each cluster is constructed by symbolic computation, and code is generated that does the actual computation at run-time. Clustering of two adjacent vertices in P-APGAN is shown in Fig. 2(b). Here, $subgraph(\{A, B\})$ is clustered into the single vertex Ω, and the topology of the graph, along with the token flow on the input and output edges is adjusted as shown in the figure. The symbols a, p, q, and b either represent parameters, or compiler-generated variables. The schedule of the subgraph corresponding to cluster Ω is constructed as $((q/g)(A))((p/g)(B))$ — (q/g) invocations of A, followed by (p/g) invocations of B, where g is a compiler-generated variable defined as $g = gcd(p, q)$. We say that $reps(A) = q/g$, and $reps(B) = p/g$, where $reps(A)$ denotes the *local repetition factor* of actor A, as opposed to the global repetition count $\mathbf{q}_G(A)$. This schedule for $subgraph(\{A, B\})$ is obtained from the APGAN technique [2].

For each input edge $\bar{e}$ of the clustered subgraph $G_{A,B} = subgraph(\{A, B\})$ incident on A, the token consumption on $\bar{e}$ is modified from $c(\bar{e})$ to $c(\bar{e}) \times reps(A)$, and the edge is re-directed to the cluster Ω — $snk(\bar{e}) = \Omega$. For each output edge $\bar{e}$ of $G_{A,B}$ incident on A, the token production $p(\bar{e})$ and the source vertex $src(\bar{e})$ are modified to $p(\bar{e}) \times reps(A)$ and to Ω, respectively. Edges incident to actor B are also processed similarly. Thus, a clustering step in P-APGAN is an entirely local computation, and does not need any global knowledge about the repetitions vector of the graph. For a detailed derivation of P-APGAN clustering, the reader is referred to [2].

After the cluster hierarchy is constructed in this fashion, P-APGAN outputs a schedule corresponding to the recursive traversal of the cluster hierarchy. Starting with a schedule for the top-level subgraph, P-APGAN recursively goes down one level of hierarchy into the subgraph corresponding to a child cluster, and the flattened schedule of this child subgraph replaces its corresponding hierarchical actor in the top-level schedule

We perform quasi-static scheduling for all PSDF acyclic graphs, and for a broad class of useful cyclic graphs, which we call *URSCC graphs* (*unit repetitions strongly connected components graphs*). A cyclic PSDF graph G is an URSCC graph if each strongly connected component (SCC) in G contains only single-rate edges, or can be configured to be so [2]; removing all edges e for which it is statically known that $d(e) \geq c(e)$ in an SCC makes the SCC acyclic; and every child graph (an init, subinit or body graph associated with a child subsystem embedded in the graph) is also an URSCC graph. For such cyclic graphs, each edge for which it is known that $d(e) \geq c(e)$ can effectively be "broken" for the purpose of scheduling the SCC. URSCC graphs arise frequently in practical systems that incorporate feedback loops, leading to graph cycles with feedback edges containing delay tokens, e.g., in adaptive filtering applications [2].

```
function compute_QS_schedule(graph G)
  foreach hierarchical actor H in G
    compute_QS_schedule(H.init( ))
    compute_QS_schedule(H.subinit( ))
    compute_QS_schedule(H.body( ))
    compute_local_synchrony_constr(subsystem(H))
  end for
  compute_local_synchrony_constr(G)
  if (is_cyclic(G)) cluster_SCCs(G)
  compute_P_APGAN_schedule(G)
end function
```

Figure 3: The quasi-static scheduling algorithm.

The algorithm compute_QS_schedule, for computing a quasi-static schedule for a PSDF graph G is given in Fig. 3. Scheduling is performed in a bottom-up fashion on a subsystem-by-subsystem basis. First, for every hierarchical actor H in G, the init, subinit, and body graphs of $subsystem(H)$ are scheduled, by recursively calling the compute_QS_schedule routine. G is then clustered in two steps. In the first step, for a cyclic graph, the routine cluster_SCCs checks if G is an URSCC graph, and if so, it removes each edge e in an SCC for which it is known that $d(e) \geq c(e)$, and clusters the resulting acyclic subgraph into a single vertex using P-APGAN. Thus, in this process each SCC is replaced by a single vertex, and G is transformed into an acyclic graph G'. In the second step, this acyclic graph G' is completely clustered, again by P-APGAN. If the routine cluster_SCCs determines that G is not an URSCC graph, then quasi-static scheduling is abandoned, and a fully-dynamic scheduler is invoked instead.

As outlined in the compute_QS_schedule algorithm, in addition to scheduling, the quasi-static compiler has to verify local synchrony of a PSDF graph G. The general strategy followed for local synchrony verification is to attempt to detect consistency compliance or violation at compile-time. If the compiler does not have sufficient information to reach a definitive conclusion at compile-time, then code is generated to verify consistency at run-time, and terminate execution in case of a run-time consistency violation. This consistency checking code is inserted in accordance with the structure of a parameterized looped

schedule (Section 3). Details of quasi-static local synchrony verification can be found in [2].

5 A DSP application in PSDF

Fig. 4 shows a PSDF model of a four-stage sample rate conversion system to interface a compact disc (CD) player to a digital audio tape (DAT) player. Each stage consists of a polyphase FIR filter that performs output-to-input conversion ratios i_k:d_k for $k = 1, 2, 3, 4$. The interpolation (i_k) and decimation (d_k) factors are modeled as subsystem parameters of the PSDF subsystem *CD-DAT*, and are configured in the init graph through the *setFac* actor (e.g., via user input during system prototyping). Actor parameters have been omitted from the illustration due to space constraints.

The sample rates of CD players and DAT players are respectively 44.1 kHz and 48 kHz. There are various ways in which to perform this conversion in four stages — e.g., (2:1 , 4:3 , 4:7 , 5:7) or (5:3 , 2:7 , 4:7 , 4:1).

The schedule generated by our quasi-static scheduler implementation is shown in Fig. 5. This schedule accommodates arbitrary choices of decimation and interpolation factors. Thus, the designer can experiment with different factors *without having to recompile the prototype software*. This flexibility is achieved systematically by the scheduler, and without the overhead of heavily-dynamic scheduling.

6 Meta-modeling using parameterization

To demonstrate the applicability of our dynamic-parameterization and quasi-static scheduling techniques to other dataflow models (beyond SDF), we present here a DSP speech compression application, represented by the *parameterized cyclo-static dataflow* (PCSDF) subsystem *Compress* in Fig. 6. Here, a speech instance of length L is transmitted from the sender side to the receiver side using as few bits as possible, applying *analysis-synthesis* techniques [6]. In the init graph, the *gHdr* actor generates a stream of header packets, where each header contains information about a speech instance, including its length L. The *setSpch* actor reads a header packet and accordingly configures L, which is modeled as a parameter of the *Compress* subsystem. The *s1* and *s2* actors are black boxes responsible for generating samples of this speech instance. Prior to analysis, the speech instance is broken up into small segments of length N, and each segment is analyzed separately in the *An* actor producing M AR coefficients (where M is the AR

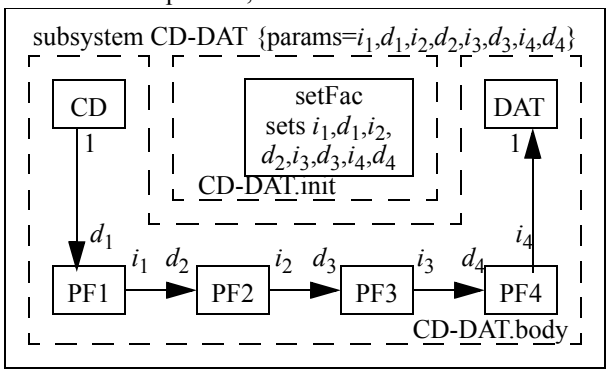

Figure 4: A CD to DAT sample rate conversion system in PSDF.

```
repeat 5 times {
  /* init graph schedule for CD-DAT */
  fire setFac /* sets i1, d1, i2, d2, i3, d3, i4, d4 */
  /* body graph schedule for CD-DAT */
  int _g1 = gcd(i1, d2)
  int _g2 = gcd((i2 × i1)/(_g1), d3)
  int _g3 = gcd((i3 × i2 × i1)/(_g2 × _g1), d4)
  repeat ((d4)/_g3) times {
    repeat ((d3)/_g2) times {
      repeat ((d2)/_g1) times {
        repeat (d1) times { fire CD }
        fire PF1
      }
      repeat ((i1)/_g1) times { fire PF2 }
    }
    repeat ((i2 × i1)/(_g2 × _g1)) times {
      fire PF3
    }
  }
  repeat ((i3 × i2 × i1)/(_g3 × _g2 × _g1)) times {
    fire PF4
  }
  repeat (i4) times { fire DAT }
}
```

Figure 5: A quasi-static schedule for Fig. 4

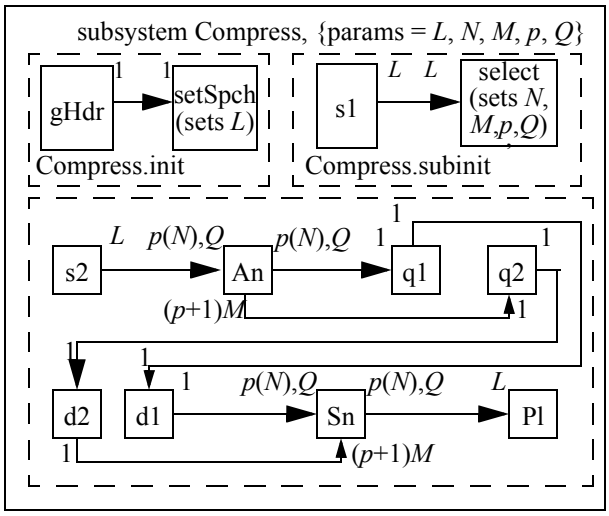

Figure 6: A speech compression application modeled in PCSDF.

model order) and the residual error signal of length N at its output. N and M are also modeled as subsystem parameters, along with two other parameters — p, which gives the number of segments of size N contained in a speech instance, and Q, which represents the size of the residual segment. Thus, if L is divided by N, then p represents the quotient, and Q represents the remainder. These parameters are configured in the subinit graph, where the *select* actor reads the entire speech instance, and examines it to determine N and M. In the body graph, the outputs of the *An* actor are quantized, encoded (actors *q1*, *q2*), and transmitted to the receiver side, where these are dequantized (actors *d1*, *d2*) and each segment is reconstructed in the *Sn* actor through AR modeling. Finally the *Pl* actor plays the entire reconstructed speech instance.

In the PCSDF specification *Compress*, the notation $p(N), Q$ denotes the parameterized cyclo-static token flow sequence $N, N, \ldots, N, Q$ with N repeated p times, signifying that the first p invocations of *An* consume N tokens each from the input port, and the $(p+1)$ th invocation consumes Q tokens.

A PCSDF quasi-static schedule for the body graph of *Compress* is shown in Fig. 7. The actor invocations enclosed in the first p firings process the first p segments of the speech sample, each of length N. The next block of actor invocations process the residual segment of length Q. The application is run five times, and in each run a different speech instance can be processed. Thus, PCSDF modeling naturally allows the same basic design to be re-configured with appropriate parameter values to process different sets of inputs in different ways, leading to more flexible dataflow behavior and increased design re-use.

```
repeat 5 times {
  /* init graph schedule for Compress */
  fire gHdr; fire setSpch; /* sets L */
  /* subinit graph schedule for Compress */
  fire s1; fire select /* sets N, M, p, Q */
  /* body graph schedule for Compress */
  fire s2
  repeat (p) times {
    fire An
    repeat (N) times { fire q1; fire d1 }
    repeat (M) times { fire q2; fire d2 }
    fire Sn
  }
  fire An
  repeat (Q) times { fire q1; fire d1 }
  repeat (M) times { fire q2; fire d2 }
  fire Sn
}
fire Pl
```

Figure 7: A quasi-static schedule for Fig. 6.

7 Conclusions and future work

This paper has presented efficient quasi-static scheduling techniques for compiling parameterized dataflow specifications. By using careful clustering, symbolic, and number-theoretic compile-time analysis, these techniques support flexible dynamic reconfiguration capability without resorting to the high-overhead and low-predictability of fully-dynamic scheduling. Our approach provides such low-overhead, dynamically-reconfigurable scheduling capability for all acyclic specification topologies, and for cyclic topologies that do not contain execution-rate changes within feedback paths. Many practical DSP applications conform to these restrictions [2, 3]. We have implemented a PSDF tool that implements the quasi-static scheduling techniques presented here. Useful directions for further work include application to multiprocessor architectures, and to other useful dataflow models such as BDDF [9] and SSDF [10].

Acknowledgements

This work was supported by the US National Science Foundation (#9734275), and by Northrop Grumman, Corp.

References

[1] M. Ade, R. Lauwereins, J. A. Peperstraete, "Buffer Memory Requirements in DSP Applications," *Intl. Wkshp. on Rapid Syst. Prototyping*, 1994.

[2] B. Bhattacharya, S.S. Bhattacharyya, "Parameterized Modeling and Scheduling for Dataflow Graphs", Tech. Rpt. UMI-ACS-TR-99-73, University of Maryland, December, 1999.

[3] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, *Software Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[4] S. S. Bhattacharyya, P. K. Murthy, E. A. Lee, "Converting Graphical DSP Programs into Memory Constrained Software Prototypes," *Intl. Wkshp. on Rapid Syst. Prototyping*, 1995.

[5] J.T. Buck, S. Ha, E. A. Lee, D.G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", *Intl. J. of Computer Simulation*, April, 1994.

[6] S. Haykin, *Adaptive Filter Theory*, third edition, Prentice Hall Information and System Sciences Series, 1996.

[7] R. Lauwereins, P. Wauters, M. Ade, J. A. Peperstraete, "Geometric Parallelism and Cyclo-Static Dataflow in GRAPE-II," *Intl. Wkshp. on Rapid Syst. Prototyping*, 1994.

[8] E.A. Lee, D.G. Messerschmitt, "Static Scheduling of Synchronous Dataflow Programs for Digital Signal Processing", *IEEE Transactions on Computers*, February, 1987.

[9] M. Pankert, O. Mauss, S. Ritz, H. Meyr, "Dynamic Data Flow and Control Flow in High Level DSP Code Synthesis," *Intl. Conf. on Acoustics, Speech, and Signal Processing*, 1994.

[10] S. Ritz, M. Pankert, and H. Meyr, "Optimum Vectorization of Scalable Synchronous Dataflow Graphs," Intl. Conf. *on Application Specific Array Processors*, 1993.