Affine Nested Loop Programs and their Binary Parameterized Dataflow Graph Counterparts

Ed F. Deprettere and Todor Stefanov
Leiden Institite of Advanced Computer Science
Leiden University
Leiden, The Netherlands
edd, stefanov@liacs.nl

Shuvra S. Bhattacharyya and Mainak Sen
Department of Electrical and Computer Engineering
University of Maryland
College Park, MD 20742
ssb, mainak@eng.umd.edu

Abstract

Parameterized static affine nested loop programs can be automatically converted to input-output equivalent Kahn Process Network specifications. These networks turn out to be close relatives of parameterized cyclo-static dataflow graphs. Token production and consumption can be cyclic with a finite number of cycles or finite non-cyclic. Moreover the token production and consumption sequences are binary.

1. Introduction

The behavior of signal processing applications is very often specified in terms of (parameterized) static affine nested loop programs which are nested loop programs in which the loop boundaries, the conditions, and the variable indexing functions are affine functions of the loop iterators and parameters. An example of such a program [1] is shown in Figure 1. Static affine nested loop programs can

```
for j = 1:1:N,

 for i = 1:1:M,

  [x(j), y(i)] = f(x(j), y(i));

 end

end
```

Figure 1. Body of an affine nested-loop program

be automatically converted to input-output equivalent Kahn Process Networs [6], [8] which are networks of processes that communicate point-to-point over unbounded unidirectional FIFO-type buffered channels, and synchronize by means of blocking reads. A KPN that is derived from an affine nested loop program [2] has limited expressive power, and has properties a general KPN does not have. Indeed, as we show in this paper, a CPN turns out to be a close relative of a parameterized cyclo-static dataflow graph (PCSDF) [1]. Cyclo-static because the steady-state behavior of the underlying affine nested loop program is cyclo-static, and parameterized for two reasons: Because the underlying program is in general parameterized, and because aprameterized dataflow (PDF) can model the initialization and termination that is in general part of the underlying program. Moreover, token production and consumption sequences are binary. For these reasons the dataflow model counterpart of the CPN model or its underlying affine nested loop program model is a form of *binary parameterized cyclo-static* dataflow (BPCSDF). BPCSDF can be viewed as the integration of parameterized dataflow meta-modeling framework [1] with cyclo-static dataflow graphs [2] that are restricted to having binary-valued token production/consumption rates on individual actor phases. Like the cyclo-static dataflow graphs [2], the restricred class of BPCSDF graphs that arise from CPNs obey balance equations [11], [10] and can be statically scheduled for bounded memory. To show all this, we first analize affine nested loop programs in two steps. The first step is to convert the affine nested loop program to an equivalent single assignment program [7] (SAP) which explicitely reveals the dependencies between the variables in the underlying affine nested loop program. The second step is to convert the SAP to a *Polyhedral Reduced Dependence Graph* (PRDG) which is a compact mathematical representation of the dependence graph counterpart of the SAP in terms of polyhedra and lattices [12], [4]. BPCSDF is built on this analysis.

2 Analysis of Affine Nested Loop Programs

The single assignment program (SAP) for the affine nested loop program in Figure 1 is shown in Figure 2.

[1] For lack of space we omit source and sink code

[2] Also called Compaan process network (CPN) because the first affine nested loop program to KPN translator was called COMPAAN [8].

```
for j = 1:1:N,

 for i = 1:1:M,

   if i-2 >= 0,

    [  in0  ] = ipd( x2( j , i-1 ) );

   else  %  if -i+1 >=0

    [ in0] = ipd( x1( j ) );

   end

   if j-2>= >0
    [ in1 ] = ipd( y2( j-1, i ) );
   else % if -j+1 >= 0
    [ in1 ] = ipd( y1( i ) );
   end
   [ out0, out1 ] = f( in0, in1 );
   [ x2( j, i) ]  = opd( out0 );
   [ y2( j, i) ]  = opd( out1 );
 end
end
```

Figure 2. The Single Assignment Program version of the Affine Nested Loop Program in Figure 1

In this program the functions `ipd()` and `opd()` are the identity function binding input variables to arguments of the function `f()`, and results of the function `f()` to output variables, respectively. The name `ipd` refers to the fact that an input variable is taken from a certain *domain*. For example, the variable `x2(j, i-1)` is taken from the domain $1 \leq j \leq N \wedge 1 \leq i \leq M \wedge i-2 \geq 0$. Because the SAP is in output normal form, output variables are all of the form $v(j,i)$. The actual domain of an output variable is obtained by substituting the input-to-output mapping or dependence function in the domain of an input variable that reads this output variable. Thus, for the variable with name $x_2(j,i)$, the dependence function is $(j,i-1)-(j,i)=(0,-1)$, and its output domain is $1 \leq j \leq N \wedge 1 \leq i \leq M-1$. With the SAP goes a dependence graph which can be compactly represented as a *polyhedral reduced depenence graph* (RPDG) $G=(\mathcal{N},\mathcal{E})$ consisting of *Nodes* $N=(I_N,O_N,f_N,\mathcal{I}_N) \in \mathcal{N}$ [3] and *Edges* $E(e,\mathcal{I}_E) \in \mathcal{E}$ between *output Ports* $Q=(q,\mathcal{I}_Q)$ and *input Ports* $P=(p,\mathcal{I}_P)$ of Nodes, where $\mathcal{I}_N$, $\mathcal{I}_E, \mathcal{I}_P$), and $\mathcal{I}_Q$ are polyhedral domains of atomic (functional) nodes f_N, atomic edges $e=(Q,P)$, and atomic input ports p and output ports q, respectively, all derived from the iteration spaces and conditions on these spaces in the SAP. (I_N,O_N) is the pair of input and output Port sets of the Node $N=(I_N,O_N,f_N,\mathcal{I}_N)$. There is a Node in the PRDG for each and every function call and its context in the underlying SAP. A Node has an input Port for each variable and its context that binds to an input argument of the node's atomic function, and an output Port for each variable and its context to which an output argument of the node's atomic function binds. The contexts of input variables and output variables are related through dependence functions as explained above. The topology of the PRDG [4] for the example in Figure. 2 is shown in Figure 3. To each polyhedral domain in

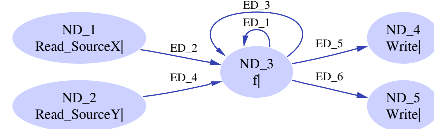

Figure 3. The PRDG corresponding to the SAP in Figure 2

the PRDG is also assigned an ordering of its integral points that may be based on the schedule and affine conditions in the SAP. For example, Node ND_3 in Figure 3 is characterized by the polytope $\{(j,i)|1 \leq j \leq N \wedge 1 \leq i \leq M\}$ with possible ordering $(M-1)j+i$ based on the loop nest schedule $j=1:1:N$, $i=1:1:M$, and has four input ports corresponding to the four conditions $(i-2) \geq 0$, $(-i+1) \geq 0$, $(j-2) \geq 0$, and $(-j+1) \geq 0$, respectively. The polynomial $(M-1)j+i$ is called the *ranking poynomial* of the Node and is the result of a parameterized counting of the integral points in the domain of the Node [5], [3], [13]. The ranking polynomial of an output Port domain $\mathcal{I}_Q$ is also called the *write ploynomial* $w(J)$ of that domain because an output Port writes tokens in the order given by its ranking polynomial. An input Port has a ranking polynomial and a *read polynomial* $r(I)$, $I \in \mathcal{I}_\mathcal{P}$, which is obtained from the write polynomial $w(J)$, $J \in \mathcal{J}_Q$, of the domain it reads from, by substituting the input Port to output Port affine dependence or mapping function $J=MI+m$. That is, $r(I)=w(J=MI+m)$. An input Port's read polynomial need not necessary be equal to its ranking polynomial. For the PRDG structure in Figure 3 and the underlying SAP in Figure 2, the output ports to edges ED_1 and ED_3 have write poynomials $w_1(j,i)=(j-1)(M-1)+i$ and

[3] $\mathcal{I}$ stands for a polytope $\{x \in \mathbb{Q}^n || A \times x \geq B \times p\}$ with A and B being integral matrices of appropriate dimension, and $p \in \mathbb{Q}^m$ b eing a parameter vector.

[4] Including source and sink nodes.

$w_3(j,i) = (j-1)M + i$, respectively. The input ports from edges ED_1 and ED_3 have read polynomials $r_1(j,i) = w_1(j,i-1) = (j-1)(M-1) + i-1$ and $r_3(j,i) = w_3(j-1,i) = (j-2)M+1$, respectively.

3 Binary Parameterized Dataflow Graph

In this section we show that a PRDG can be converted to a parameterized cyclo-static dataflow graph (PCSDF) [1]. Parameterized dataflow can model the initializations and terminations that are in general part of the underlying affine nested loop program, that has otherwise a cyclo-static steady-state behavior. Moreover, the dataflow graph derived from a PRDG has token production and consumption sequences that are binary. For this reason we call these dataflow graphs *binary parametrized cyclo-static* dataflow graphs (BPCSDF).

The relation between the BPCSDF graph and the PRDG is simple: for every Node in the PRDG we have an actor in the BPCSDF graph, and for every edge in the PRDG we have an edge in the BPCSDF graph. Because actors in dataflow graphs are characterized by token production and consumption patterns, we have to derive these patterns from the PRDG. Thus let P be a parameterized static affine nested loop program. For each function call F_t and its context in the program there is an actor A_t. The actor has K_i^t input ports $p_{k,i}^t$, $k = 1, 2, \ldots, K_i^t$, and L_j^t output ports $q_{l,j}^t$, $l = 1, 2, \ldots, L_j^t$ corresponding to the program input variable $x_i^t()$ and output variable $y_j^t()$, respectively. Each actor input port has a port domain that is a subset of the actor function domain, and a read polynomial defined on that domain. Similarly, each actor output port has a port domain that is a subset of the actor function domain, and a write polynomial defined on that domain. Let $n_{k,i}^t$ be the cardinality of the domain of the input port $p_{k,i}^t$, and let $m_{l,j}^t$ be the cardinality of the domain of output port $q_{l,j}^t$, then actor A_t will read $n_{k,i}^t$ tokens from input port $p_{k,i}^t$, and write $m_{l,j}^t$ tokens to output port $q_{l,j}^t$. Finally, if s_t is the cardinality of the actor function domain, then the consumption pattern of the input port $p_{k,i}^t$ can be characterized by a consumption sequence of length s_t containing $n_{k,i}^t$ *ones* and $s_t - n_{k,i}^t$ *zeros*. Similar, the production pattern of the output port $q_{l,j}^t$ can be characterized by a production sequence of length s_t containing $m_{l,j}^t$ ones and $s_t - m_{l,j}^t$ zeros.

These production and consumption sequences s_t can in principle be generated as follows. Let actor A_t have a function domain $\mathcal{D} = \{I_k, k = 1,2,...,k_{max} | I_k \in \mathcal{P} \cap \mathbb{Z}^n \wedge I_k \prec I_{k+1}, k = 1,2,...,k_{max}-1\}$, and let $\mathcal{D}_{Opd} \subseteq \mathcal{D}$ be an output port domain. Start out with an empty sequence $S_O = \{\ \}$. For $k = 1,2,...,k_{max}$ if $I_k \in \mathcal{D}_{Opd}$, put a 1 in S_O, else put a 0 in S_O. Because k_{max} is independent of $\mathcal{D}_{Opd}$, all output ports have a sequence of the same length. Next let $\mathcal{D}_{Ipd} \subseteq \mathcal{D}$ be an input port domain. Start out again with an empty sequence $S_I = \{\ \}$. For $k = 1,2,...,k_{max}$ if $I_k \in \mathcal{D}_{Ipd}$, put a 1 in S_I, else put a 0 in S_I. Again, k_{max} is independent of $\mathcal{D}_{Ipd}$ and, therefore, also all input port sequences have the same length equal to the length of all output port sequences. Clearly, the production and consumption sequencies s_t are binary. The scanning of the domains $\mathcal{D}_{Ipd}$ and $\mathcal{D}_{Opd}$ will in general lead to production and consumption sequences that have compact representations, because the patterns of zeros and ones can be generated by means of counting polynomials. The patterns may take the form of a *repetitions period* and a *repetitions factor* in case the sequence is periodic and repeats for finitely many periods, or a *finite non-periodic sequence*. We illustrate this for the example in Figure 3 and the underlying program in Figure 2. The f-actor is shown in Figure 4 together with the compact consumption and production sequences.

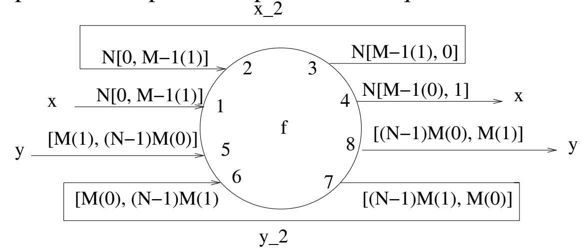

Figure 4. f-actor for the program in Figure 2 with superimposed production and consumption sequences

Input port 2 has domain $1 \le j \le N \wedge 2 \le i \le M$, and lexical order $j = 1:1:N, i = 1:1:M$. For any j, the *repetitions perid* is $[0 \;\; M-1(1)]$ because $2 \le i \le M$ is in the domain and $i = 1$ is not in the domain. Because $1 \le j \le N$, the *repetitions factor* is N whence the consumption sequence is $N[0 \;\; M-1(1)]$. Similarly, output port 3 has domain $1 \le j \le N \wedge 1 \le i \le M-1$, and lexical order $j = 1:1:N, i = 1:1:M$. For any j, the repetitions period is $[M-1(1) \;\; 0]$ because $1 \le i \le M-1$ is in the domain and $i = M$ is not in the domain. Because $1 \le j \le N$, the repetitions factor is N whence the production sequence is $N[M-1(1) \;\; 0]$. And similarly for the other ports, except for input Port 5 and output port 8

whose consumption and production sequences are compact but not factored into a repetitions period and a repetitions factor. Now, given the consumption and production sequences s_t for the actors A_t, we can construct the topology matrix whose $t-th$ column has entries $-\frac{n_{k,i}^t}{s_t}$ and $\frac{m_{l,j}^t}{s_t}$, respectively. We may thus conclude that a PRDG has a dataflow graph counterpart. The actors are characterized by parameterized finite length token production and consumption sequences which may or may not be cyclic. This is different from CSDF graphs in which the actors are characterized by constant token production and consumption vectors, called phase signatures, that repeat indefinitly. The difference arises because an affine nested loop program operates in principle on finite sequences and has neither initial tokens nor termination tokens in its buffers whereas a CSDF graph operates on infinite sequences modeling a steady-state behavior, whence it has in general initial and termination tokens in its buffers. PCSDF can easily model such program initializations and terminations [1]. In the example of Figure 4 and the underlying program in Figure 2, the initial state is taken from the y-source Node at $j = 1$, and the final state is sent to the y-sink Node at $j = N$. Therefore, $2 \leq j \leq N - 1$ is the steady-state region, and if we assume that $N \to \infty$, then we arrive at the infinite sequence steady-state equivalent shown in Figure 5 which is a genuine CSDF graph. In this figure, the repeti-

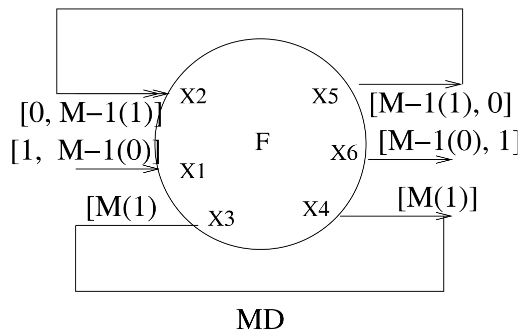

Figure 5. BPCSDF graph with CSDF behavior obtained by restricting the BPCSDF graph in Figure 4 to its steady-state region and assuming that $N \to \infty$

tions factor N does not appear because of the assumption that $N \to \infty$. If not, then a repetitions factor $N - 2$ will apply, and the steady-state graph will be *finite sequence* equivalent to a CSDF graph. Now assuming that the two source nodes and the two sink nodes in the example of Figure 4 and the underlying program in Figure 2 produce and consume one token at a time, the balance equations are as follows,

$$\begin{bmatrix} 1 & 0 & -\frac{N}{NM} & 0 & 0 \\ 0 & 1 & -\frac{M}{NM} & 0 & 0 \\ 0 & 0 & \frac{N}{NM} & -1 & 0 \\ 0 & 0 & \frac{M}{NM} & 0 & -1 \end{bmatrix} \begin{bmatrix} \mathrm{N} \\ \mathrm{M} \\ \mathrm{NM} \\ \mathrm{N} \\ \mathrm{M} \end{bmatrix} = 0. \tag{1}$$

Similarly, the balance equations for the steady-state model of Figure 5 are

$$\begin{bmatrix} 1 & -\frac{1}{M} & 0 \\ 0 & \frac{1}{M} & -1 \end{bmatrix} \begin{bmatrix} 1 \\ \mathrm{M} \\ 1 \end{bmatrix} = 0. \tag{2}$$

In the first case, in a single cycle, the source nodes fire N times and M times, respectively, the f Node fires NM times, and the sink nodes fire N and M times, respectively. In the second case, the f Node fires M times for every single firing of the source node and the sink node. BPCSDF graphs not only obey balance equations, they also can be statically scheduled for bounded memory [14]. The procedure starts out with the embedding of all function domains, one at a time, in a common domain at offsets that guarantee minimal causal dependence mapping functions (write before read). Ordering in the common domain is lexical and defines a schedule. After this scheduling, edges can simply be considered self-loops in the common domain. Buffer sizes for self-loops are relatively easy to determine. Of course the buffer sizes that are computed this way depend on the underlying schedule and may not be the absolute minimum buffer sizes but at best the minimum buffer sizes for the computed schedule.

4 Application example:Speech Coding

As an application example, consider the following speech coding algorithm in Figure 6 [9]. In this

```
for k = 1 : 1 : K,
 [e[k], r[k]] = F1(s[k]);
 for p = 1 : 1 : P,
  [rho(k)] = F2(e[k], r[k]);
  [e[k], r[k]] = F3(e[k], r[k], rho(k));
 end
 [v[k]] = F4(e[k]);
 [b[k], N(k) , phase(k)] = F5(v[k]);
end
```

Figure 6. A speech coding algorithm

program, all variables are vectors of length L [5], except for rho(k) which is a scalar variable. The

[5]Corresponding to a partition of the speech signal in segments of length L.

function $F1$ generates the vectors $e[k] = s[k]$ and $r[k] = z^{-1}s[k]$, where $s[k]$ is the current speech segment and z^{-1} is the (right) shift operator. Function $F2$ computes filter coefficients (the so-called reflection coefficients) and function $F3$ performs inverse filtering. Function $F4$ computes an approximate excitation signal for the speech reproduction filter. Finally, function $F5$ finds the best downsampled version $b[k]$ of $v[n]$, where N is the downsampling factor, and *phase* is the downsampling phase which is a value between 1 and N. The BPCSDF graph for this program is shown in Figure 7.

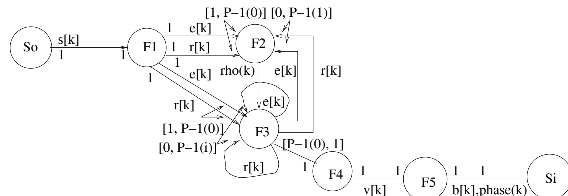

Figure 7. The BPCSDF graph for the speech coding algorithm in Figure 6

The balance equations for this algorithm are as follows.

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -\frac{1}{P} & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -\frac{1}{P} & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & -\frac{P-1}{P} & \frac{P-1}{P} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{P} & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ P \\ P \\ 1 \\ 1 \\ 1 \end{bmatrix} = 0. \quad (3)$$

We deduce from these equations that actors F_2 and F_3 fire P times for every firing of the other actors in the graph.

5 Conclusion

We have shown that affine nested loop programs can be converted to binary parameterized cyclo-static dataflow graphs (BPCSDF). In contrast with CSDF graphs, BPCSDF graphs have actors that need not fire for an unbounded number of cycles in a periodic firing pattern, but can fire cyclically for a finite number of cycles or in a finite non-cyclic way. Parametrized CSDF can deal with that. Finally, the addition *binary* refers to the fact that the graphs originating from affine nested loop programs have actor token production and consumption patterns that are binary.

References

[1] B. Bhattacharya and S. Bhattacharyya. Parameterized dataflow modeling for dsp systems. *IEEE Trans. Signal Processing*, 49(10):2408–2420, Oct. 2001.

[2] G. Bilsen, M. Engels, R. Lauewrijns, and J. Peperstraete. Parameterized dataflow modeling for dsp systems. *IEEE Trans. Signal Processing*, 44(2):397–408, Feb. 1996.

[3] P. Claus and V. Loechner. Parametric analysis of polyhedral iteration spaces. In *IEEE Int. Conf. on Application Specific Array Processors, ASAP96*, pages 415, 424, Aug 1996.

[4] E. F. Deprettere, E. Rijpkema, and B. Kienhuis. Translating imperative affine nested loop programs to process networks. In E. F. Deprettere, J. Teich, and S. Vassiliadis, editors, *Embedded Processor Design Challenges, LNCS 2268*, pages 89–111. Springer, Berlin, 2002.

[5] E. Ehrhart. Sur les polyèdres rationnels homothétiques à n dimensions. In *C.R. Acad. Sci. Paris*, volume 254, pages 616–618, 1962.

[6] G. Kahn. The semantics of a simple language for parallel programming. In *Proc. of the IFIP Congress 74*, pages 5 – 10. North-Holland Publishing Co., August 1974.

[7] B. Kienhuis. MatParser: An array dataflow analysis compiler. Technical report, University of California at Berkeley, 2000. UCB/ERL M00/9.

[8] B. Kienhuis, E. Rijpkema, and E. F. Deprettere. Compaan: Deriving process networks from matlab for embedded signal processing architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, San Diego, USA, May 2000.

[9] P. Kroon, E. Deprettere, and R. Sluyter. Regular pulse excitation. *IEEE Trans. Acoustics, Speech, and Signal Processing*, 34(5):179–194, October 1986.

[10] R. Lauwereins, P. Wauters, M. Adi, and J. Peperstraete. Geometric parallelism and cyclo-static dataflow in grape-ii. In *Proc. 5th Int. Workshop on Rapid System Prototyping*. North-Holland Publishing Co., June 1994.

[11] E. Lee and D. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, September 1987.

[12] E. Rijpkema. *Modeling Task Level Parallelism in Piece-wise Regular Programs*. PhD thesis, Leiden Institute of Advanced Computer Science, Leiden University, The Netherlands, September 2002.

[13] S. Verdoolaege, K. Beyis, M. Bruynooghe, R. Seghir, and V. Loechner. Analytical computation of ehrhart polynomials and its applications for embedded systems. In *2nd Workshop on Optimizations for DSP and Embedded Systems, (ODES'02)*, Palo Alto, CA, 2004.

[14] S. Verdoolaege, H. Nikolov, and T. Stefanov. Improved derivation of process networks. In *Proceedings 2nd Workshop on Optimization for DSP and Embedded Systems (Odes'06)*, Palo Alto, Mar. 2006.