

# STATIC SCHEDULING FOR CYCLO STATIC DATA FLOW GRAPHS

Sukumar Reddy Anapalli  
Dept. of Computer Science  
The University of Akron  
Akron, OH, 44325, USA

Krishna Chaithanya Chakilam  
Dept. of Computer Science  
The University of Akron  
Akron, OH, 44325, USA

Timothy W. O'Neil  
Dept. of Computer Science  
The University of Akron  
Akron, OH, 44325, USA

**Abstract** - *There are few processes which display cyclically changing but predefined behavior. These processes can be represented using cyclo static data flow graphs (CSDFG). This capability results in a higher degree of parallelism. In this paper we present the iteration bound for a CSDFG which is used to find the integral static schedule and determine whether a csdf is live or not based on some calculation. We also present an algorithm that schedules cyclo static data flow graphs without converting to their equivalent homogeneous graphs (EHG's) and demonstrate it with a suitable example.*

**Index Terms**—Static Scheduling, Cyclo-Static Data Flow Graphs.

## 1 Introduction

Digital signal processing applications such as speech synthesis and recognition, telecommunication, image and video processing, robotics, etc., are non-terminating in nature. DSP applications are best represented by cyclo-static dataflow (CSDF), which is a new model for the specification and implementation of digital signal processing algorithms. The CSDF paradigm is an extension of synchronous dataflow that still allows for static scheduling and, thus, a very efficient implementation of an application.

As discussed and shown in [1] and [6], CSDF graphs are more expressive than multi-rate or synchronous data flow graphs [7]. This means that both a larger class of applications can be modeled and that more detailed dependencies can be included, which often leads to lower resource requirements. From the CSDF graphs we can analytically derive the cycle that determines the throughput after conversion to a single-rate dataflow graph [1]. Tasks are modeled by the vertices of a CSDF graph, which are called actors.

In comparison with synchronous dataflow, CSDF is more versatile because it also supports algorithms with a cyclically changing, but predefined, behavior. In SDF, actors have static firing rules, they consume and produce

a fixed number of data tokens in each firing. This model is well suited to multirate signal processing applications and lends itself to efficient, static scheduling, avoiding the runtime scheduling overhead incurred by general implementations of process networks. The CSDF graph remains fully analyzable at design time and allows reasoning about the temporal behavior. Our examples show that this capability results in a higher degree of parallelism and, hence, a higher throughput, shorter delays, and less buffer memory.

The authors in [1] have introduced the CSDF paradigm and also derived necessary and sufficient conditions for the schedulability of a CSDF graph. In this they gave two methods for checking the liveness of a CSDF graph. The first one checks the liveness and the second one constructs the single rate equivalent homogeneous graph (EHG) of a CSDF graph for one iteration. If a graph is in its EHG form then we have algorithms to schedule it. In this paper we develop an algorithm that takes the CSDF graph as input and produces a static schedule as output directly without converting it to its EHG. In this paper we assume scheduling a CSDF graph on multiple processors. In the following sections we give fundamental definitions related to our work, a method for determining the liveness and then we give an iteration bound equation for a CSDF graph. Since our work in this paper is related to [1], all the terms and conditions applied there are used in our work. Finally we will demonstrate our CSDFG static scheduling algorithm with an example.

## 2 Synchronous Data Flow Graphs

There are a few processes where each of the operations needs to produce and consume multiple units of data. Data flow graphs cannot be used to represent these types of applications since each node in a DFG is capable of producing and consuming a single unit of data. In order to address this situation, Lee in [5] has introduced the concept of synchronous data flow graphs (SDFG's) where each node is capable of producing or consuming multiple tokens of data on every iteration. The number of tokens produced or consumed by each node is predetermined.

## 2.1 Definition

A *synchronous data flow graph* (SDFG) is represented by  $G=(V,E, d, t, p, c)$  such that

- $V$  is the set of operations denoted by nodes in the graph.
- $E$  is the set of edges that connect the nodes in the graph. They determine the dependencies between the operations.
- $d(e)$  denotes the number of delays on edge  $e$ , for any edge  $d(e) \geq 0$ .
- $t(v)$  denotes the number of time steps required to compute operation of a node .
- $p(e)$  denotes the number of data tokens produced by the source node on edge  $e$ .
- $c(e)$  denotes the number of data tokens consumed by the sink node.

For example, consider the SDFG in the Figure 2.2.

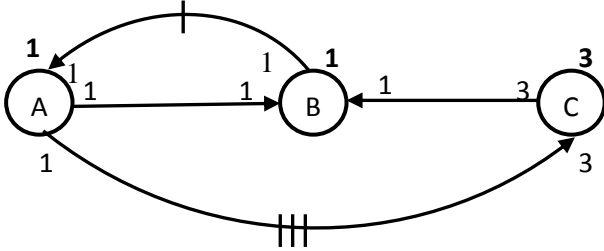


Figure 2.2 Synchronous data flow graph

In the above figure, it can be observed that the set  $\{A, B, C\}$  represents the set of vertices, and the numbers above the nodes represent the computation times of each of the operations. The number of delays  $d(e)$  on each edge is denoted by the vertical bars cutting across the edge. On each edge we can find two numbers representing the number of data tokens produced and consumed by the edge. In the synchronous data flow graph shown in Figure 2.2, we can see the node A produces 1 and C consumes 3 data items on the bottom edge.

## 3 Cyclo Static Data Flow Graphs

The cyclo static data flow graph (CSDF) paradigm is an extension of synchronous data flow graphs. These graphs are more versatile than SDFG's from [1], because they support algorithms which are critically changing in a cyclic manner but the behavior is predefined.

In the Figure 3.1 it can be observed that the set  $\{V_1, V_2, V_3\}$  represents the set of vertices. Assume the computation time of each of the operations is 1. The number of delays  $d(e)$  on each edge is denoted by the vertical bars cutting across the edge. On each edge we can find a sequence of numbers representing the number of data tokens produced and consumed by the edge.

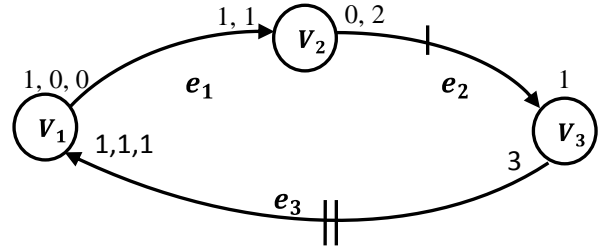


Figure 3.1 Cyclo-static data flow graph

In the cyclo-static data flow graph shown in this figure we can see the node  $V_1$  produces 1,0,0. This implies that when it is iterated for the *first* time it produces 1 token on the edge and in the *second* and *third* iterations it produces 0. It repeats in a cyclic fashion in 1, 4, 7... iterations it produces 1 token on an edge and in the remaining iterations it produces 0 tokens. In the case of consumption the same method is followed by each node when they have been repeating in a cyclic fashion.

### 3.1 Topology matrix

The author in [5] proved necessary and sufficient condition for a CSDFG to be consistent. For the existence of a schedule the rank of the topology matrix must be one less than the number of nodes in the cyclo static data flow (CSDF) graph as in multirate graphs. A *topology matrix*  $\Gamma$  is one that stores the information about every edge in the cyclo static data flow graph. Each row in  $\Gamma$  corresponds to one edge in  $G$ , and each column corresponds to a specific node in  $G$ . In the case of CSDF graphs a positive  $(i, j)$  entry in the topology matrix  $\Gamma$  indicates the number of tokens produced by the  $j_{th}$  node on the  $i_{th}$  edge for one complete execution of the production sequence at that node, while a negative entry gives the total number of tokens consumed by the node for one complete execution of the consumption sequence at that node. A value of zero indicates that the vertex has not used the edge to produce or consume data units.

For example the topology matrix for the CSDFG in Figure 3.1 is shown in figure 3.2

$$\Gamma = \begin{bmatrix} 1 & -2 & 0 \\ 0 & 2 & -1 \\ -3 & 0 & 3 \end{bmatrix}$$

Figure 3.2 Topology matrix

The CSDF graph is said to be consistent if there exists a *repetition vector*. The repetition vector for  $G$  with the smallest norm is called the *basic repetition vector*. Rank  $(\Gamma) = N_G - 1$  further implies that all repetition vectors are a multiple of the basic repetition vector, which can be derived from any arbitrary repetition vector  $\vec{q}_G$  as  $\vec{q}_G = \vec{q}_G / s$ .

where  $s = gcd_{v_j \in G} \{q_j/P_j\}$ . The graph in Figure 3.1 is consistent and has a basic repetition vector  $[6,2,2]^T$ .

## 4 Scheduling CSDFGs

In this section we give some basic definitions that hold for CSDFGs. We also show a way to determine the liveness of a CSDFG and find an equation for the iteration bound of a CSDFG. Finally we will develop an algorithm to find the static schedule for a CSDF graph.

### 4.1 Definitions

Here we give some definitions that the authors in [2] have given. A delay on an edge  $(u, v)$  shows the execution sequence between  $u$  and  $v$ . For a DFG to be schedulable, the total delay count should not be zero. The same holds for the CSDF graphs. Executing all nodes once in a graph  $G$  is referred to as *iteration*. The starting time of node  $v$  in the  $i^{th}$  iteration is  $S(v, i)$ . A schedule is *legal* if for every edge  $u \rightarrow v$  and iteration  $i$ , we have  $S(u, i) + t(u) \leq S(v, i + d(e))$ . A schedule is said to have an unfolding factor  $f$  and a cycle period  $c$  if  $S(v, i + f) = S(v, i) + c$  for every  $v$  in  $V$  and iteration  $i$ . Thus, such a schedule can be represented by the partial schedule of the first  $f$  iterations. A new instance of this partial schedule of  $f$  iterations can be initiated for every interval of length  $c$  to form a complete legal schedule. If in every instantiation, an operation of the partial schedule is assigned to the same processor, the schedule is called a *static schedule*. We consider only connected CSDF graphs because a valid schedule for any unconnected graph can be built by combining valid schedules of its connected subgraphs. A *valid static schedule* for  $G$  is a schedule that can be repeated infinitely on the incoming sample stream and where the amount of data in the buffers remains unbounded.

The *iteration period* of a repeating schedule is the average computation time per iteration, it is represented as  $c/f$ . If the value of the iteration bound equals that of the iteration period, then the schedule is called a *rate-optimal static schedule*. Now we see some results found in [3] to form a legal schedule.

For a graph (DFG or CSDFG)  $G$ , a legal schedule  $S$  with cycle period  $c$  and unfolding factor  $f$  can be implemented under a non-pipelined design if and only if  $c \geq \max_v t(v)$ . The proof of this is given in [3].

### 4.2 Live and consistent

A valid static schedule exists for a cyclo static data flow (CSDF) graph if the graph is both *live* and *consistent*. A cyclo static data flow graph (CSDF) is said to be *consistent* if there exists a *repetition vector*. If a *deadlock free* schedule can be found, then  $G$  is said to be *live*.

The authors in [1] have given a detailed description of constructing a basic repetition vector and also finding the liveness of a CSDF graph. However for constructing a static schedule for a CSDF graph, liveness becomes a problem if there is a cycle in it. Due to this reason as in [2] we develop the following result for a CSDFG.

**THEOREM 4.2.1:** For any edge  $(j, k)$  with delay count  $d(e)$  in the critical cycle of a CSDFG, the CSDFG is live regardless how other delays are distributed on edges among graph, if  $d(e) \geq \left(\frac{q_j}{p_j}\right) * \text{no}(v_j)$ .

**PROOF:** If we have  $q_j$  instances of node  $v_1$  with period length  $p_1$ ,  $q_k$  instances of node  $v_2$  with period length  $p_k$  and  $e$  edges connecting the  $v_1$  and  $v_2$ , we require  $\left(\frac{q_j}{p_j}\right) * \text{no}(v_j)$  edges connecting both the nodes in the EHG,  $\text{no}(v_j) = \sum_{i=1}^{p_j} P_i$  where  $p_i$  is the number of data tokens produced at vertex  $j$ , and  $\text{no}(v_j)$  is one complete periodic cycle at that node. If  $d(e) \geq \left(\frac{q_j}{p_j}\right) * \text{no}(v_j)$  then every edge will have atleast one delay connecting  $v_1$  and  $v_2$ . If  $d(e) < \left(\frac{q_j}{p_j}\right) * \text{no}(v_j)$  then this will lead us to an edge which has no delay and that leads to a deadlocked graph. This procedure is repeated for all the edges present in the CSDFG. If this holds for any one edge then the graph is said to be live.

Let's check the derived conditions by applying then to the CSDFG shown in Figure 3.1. The basic repetition vector BRV  $q=[6,2,2]$ , the period of each node  $P=[3,2,1]$  and  $\text{no}(v_j)=[1,2,3]$ . By applying the above condition  $\left(\frac{q_j}{p_j}\right) * \text{no}(v_j)$  we get 2 edges connecting  $v_1$  and  $v_2$ , 2 edges connecting  $v_2$  and  $v_3$ , and 6 edges connecting  $v_3$  and  $v_1$ . The delays on edges  $v_1, v_2$  and  $v_3$  are 0, 1 and 0 and now  $0 < 2$ ,  $1 < 2$  and  $0 < 6$ , which implies the graph is not live. By adding one delay on edge  $e_2$  we can make the CSDF graph in Figure 3.1 as a live graph.

### 4.3 Iteration Bound for SDFG

In this section we develop an equation to find the iteration bound for CSDFGs without converting them to their single rate equivalents. The iteration bound is represented as  $\beta(G^*)$ .

**THEOREM 4.3.1:** Let  $G$  be a CSDFG with a repetition vector  $RV = [q_1, \dots, q_k]$ , then the iteration bound for the CSDFG is calculated by:

$$\beta(G^*) = \max_{\text{loop } l \in G} \frac{\sum_{v \in l} t(v)}{\sum_{e: j \rightarrow k} \left\lfloor \frac{d(e)}{\left(\frac{q_j}{p_j}\right) * \text{no}(v_j)} \right\rfloor}$$

**PROOF:** The numerator  $t(v)$  in the above equation is the execution time of a node. It is same as the one for the SDFG, because while converting from CSDFG to single-rate equivalent the nodes in a cycle remains the same. In the denominator, if an edge  $e$  in the CSDFG connects two nodes  $(v_j, v_k)$  with periodic cycle at the nodes  $P = [P_j, P_k]$ , with repetition vector  $RV = [q_j, q_k]$ , and a delay count of  $d$ , this means that we can have  $\left(\frac{q_j}{p_j}\right) * \text{no}(v_j)$  edges going from  $v_j$  to  $v_k$  in its *single-rate graph* with  $d$  delays divided among the edges, where  $\text{no}(v_j) = \sum_{i=1}^{p_j} P_i$  where  $p_i$  is the number of data tokens produced at vertex  $j$ , and  $\text{no}(v_j)$  is one complete periodic cycle at that node. This procedure is repeated for every edge in the CSDFG. Since we are interested in the cycle with minimum number of delays we apply the floor operation.

Let's take the CSDF graph in Figure 4.3.1 to illustrate this idea. The basis repetition vector (BRV) is  $[4, 2, 1]$ . The length of the period at each node  $P = [4, 2, 1]$  which means from the BRV that we have four copies of  $V_1$ , two copies of  $V_2$  and one copy of  $V_3$  in its EHG. There should be  $\left(\frac{q_1}{p_1}\right) * \text{no}(v_1)$  edges going from  $V_1$  instances to  $V_2$  instances in its EHG, so we have two edges connecting  $V_1$  instances and  $V_2$  instances. The distribution of the delays on the edges are calculated by dividing the total number of delays equally on the total number of edges going from  $V_1$  instances to  $V_2$  instances. Since there are two delays on edge  $V_1 \rightarrow V_2$  in the CSDF graph, the edges in its equivalent graph each contain a delay. Similarly from  $V_2 \rightarrow V_3$  and  $V_3 \rightarrow V_1$  we have 2 and 4 edges in its EHG, respectively.

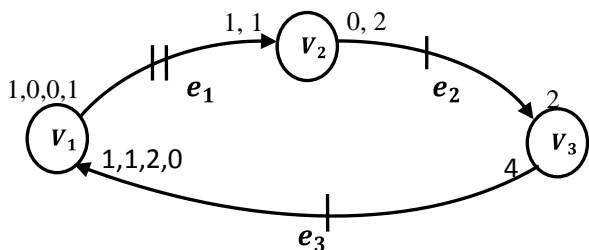


Figure 4.3.1 CSDF graph.

#### 4.4 Scheduling Algorithm for CSDF Graphs

From the above results, we can now develop an algorithm for scheduling the CSDFG. Here we assume scheduling for only the *integral model*, where every node is issued at the starting of a time unit, which as a result gives us an integral cycle period.

A similar algorithm can be found in [2]. The algorithm present there is applied for SDFGs only. By calculating the iteration bound equation for CSDFGs we modified the algorithm to schedule CSDFGs.

```

Input: A CSDFG G, an unfolding factor  $f$ , a cycle period
Repetition Vector  $q$ , and a period  $p_j$ .
Output: A schedule for the first  $f$  iterations such that
every  $f$  iterations can start in an interval of length  $c$ 
Begin /* Q is a first-in first-out queue. Pass is used to
prevent the algorithm being trapped in negative cycles.
*/
  For every node  $v \in V$  do begin
     $Sh(v) \leftarrow 0$ ; PUSH  $v$  to Q;
  End for
  Pas  $\leftarrow 0$ ;
  Last  $\leftarrow$  the last element in Q;
  While Q is not empty and pass  $< \lfloor V \rfloor$  do begin
    Pop  $v$  from Q
    For every edge  $e = (v, w) \in E$  do begin
       $\text{no}(V_j) = \sum_{i=1}^j P_i$  where P is the no of data
      tokens produced at vertex j in one
      complete periodic cycle at that node.
       $sh(w^*) \leftarrow \min \left\{ sh(v) + \left\lfloor \frac{d(e)}{\left(\frac{q_j}{p_j}\right) * \text{no}(V_j)} \right\rfloor - t(j) \times f / c, sh(w) \right\}$ 
      If  $sh(w^*) < sh(w)$  then
         $sh(w) \leftarrow sh(w^*)$ 
        If  $w$  is not in Q then
          Push  $w$ 
        End if
      End if
    End for
    If  $v = \text{last}$  then begin
      Pas  $\leftarrow$  pass + 1;
      Last  $\leftarrow$  the last element in Q;
    End if
  End while
  if Q is empty then
    For  $i = 0$  to  $f - 1$  do
      For every  $v \in V$  do
        For  $y = 1$  to  $q_j$ 
           $S(v_i, y) \leftarrow \lfloor -sh(v) * c/f + i * c/f \rfloor$ 
        End for
      End for
    End for
    Else "No such schedule exists."
  END if
END begin

```

Figure 4.4.1 Algorithm for finding a static schedule for a given CSDFG

## 4.5 Results

In this section, we illustrate our results further by applying them to CSDFGs and show the above algorithm can find the static schedule for a CSDFG. For the below example in Figure 4.3.2 we will find the static schedule assuming the cycle period  $c$  and unfolding factor  $f$  are 3 and 2, respectively. The iteration bound is also calculated and since it is equal to  $3/2$ , a rate optimal static schedule can be constructed.

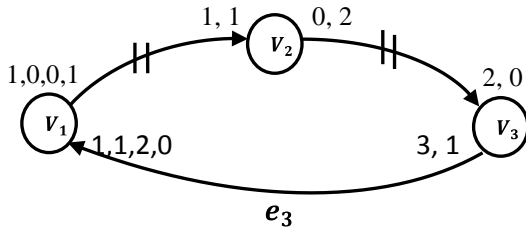


Figure 4.5.1 A CSDFG

The CSDF graph is converted to a scheduling graph, with the added node  $v_0$  and the weights of those edges

calculated by  $w(e) = \left\lfloor \frac{d(e)}{\left(\frac{q_j}{p_j}\right) * \text{no}(V_j)} \right\rfloor - t(j) \times f / c$  for edge  $e: j \rightarrow k$ .

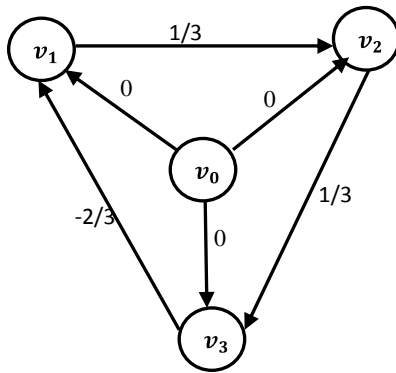


Figure 4.5.2 Scheduling graph for Figure 4.5.1

From the scheduling graph, we can find the shortest paths for the nodes using the methods in [3]. They are  $sh(v_1) = -2/3$ ,  $sh(v_2) = -1/3$  and  $sh(v_3) = 0$ . The repetition vector for the above CSDF graph is  $RV = [4, 2, 1]$  and the cyclic period at each vertex is  $P = [4, 2, 1]$ . With these values known and given to the scheduling algorithm the static schedule look as shown in Figure 4.5.3.

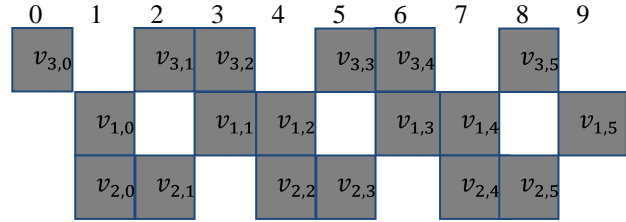


Figure 4.5.3 Static schedule for Figure 4.5.1

In Figure 4.5.3  $v_{1,0}$  implies four instances of  $v_1$  are scheduled at the same time,  $v_{2,0}$  implies that two instances of  $v_2$  are scheduled at the same time and  $v_{3,0}$  implies one copy of  $v_3$  is scheduled at the same time based on the repetition vector which gives how many instances are to be scheduled for each node.

## 5 Conclusion

In this paper we were able to find and present an iteration bound equation for cyclo static data flow graphs (CSDFGs). We derived an equation for checking the liveness of a CSDFG which is a key factor for constructing a static schedule. Finally we presented a static scheduling algorithm for CSDF graph which works based on the integral model, without converting to its EHG, by the use of the repetition vector and periodic vector of each node.

## References

- [1] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static data flow. *IEEE Transactions on signal processing*, vol 44(2) pages:397-408, February 1996.
- [2] Samer F. Khasawneh, M.E.Richter, T.W. O'Neil. Static Scheduling for synchronous data flow graphs. *Proceedings Of ISCA 22<sup>nd</sup> international conference on computers and their applications*, pages 38-43, 2007.
- [3] Liang-Fang Cha, Edwin Hsing-Mean Sha. Static scheduling for Synthesis of DSP Algorithms on Various Models. *Journal of VLSI Signal Processing*, vol 10, pages: 207-223 (1995).
- [4] G. Bilsen, M. Engels, R. Lauwereins, and J. A. Peperstraete. Cyclo-static data flow. In *IEEE Int. Conf. ASSP*, pages: 3255-3258, Detroit, Michigan, May 1995.
- [5] E. A. Lee and D. G. Messerschmitt, "static scheduling of synchronous dataflow programs for digital signal processing." *IEEE Trans, Comput.*, vol, C-36, pages: 24-35, jan 1987.
- [6] T. M. Parks *et al.* A Comparison of Synchronous and Cyclo-Static Data flow. In *Proc. IEEE Asilomar Conference on Signals, Systems and Computers*, pages: 204-210, October 1995.
- [7] E. A. Lee and D. G. Messerschmitt. Synchronous Data flow. *Proceedings of the IEEE* vol 75, Pages: 1235-1245, September 1987.