

# On Retiming Synchronous Data-Flow Graphs

Timothy W. O'Neil  
CSE Department  
Univ. of Notre Dame  
Notre Dame, IN 46556

Edwin H.-M. Sha  
CS Department  
Univ. of Texas at Dallas  
Richardson, TX 75083-0688

## Abstract

Many common iterative or recursive DSP applications can be represented by synchronous data-flow graphs (SDFGs). A great deal of research has been done attempting to optimize such applications through retiming. However, despite its proven effectiveness in transforming single-rate data-flow graphs to equivalent DFGs with smaller clock periods, the use of retiming for attempting to reduce the execution time of synchronous DFGs has never been explored. This paper accomplishes exactly this. It develops the basic definitions and results necessary for expressing and studying SDFGs. The problems faced when attempting to retime a SDFG in order to minimize clock period are reviewed, then a retiming algorithm presented. Finally, the effectiveness of our method is demonstrated on an example.

## 1 Introduction

Since the most time-critical parts of DSP applications are loops, we must explore the parallelism embedded in the repetitive pattern of a loop. One of the most useful models for representing DSP applications has proven to be the *multirate* or *synchronous data-flow graph* (SDFG) first proposed by Lee [4]. The nodes of a SDFG represent functional elements, while edges between nodes represent connections between them. Each node consumes and produces a predetermined fixed number of *delays* (i.e., data tokens) on each invocation. Additionally, each edge may contain some initial number of delays. This model has proven popular with designers of signal processing programming environments with its use leading to numerous important results.

A great deal of research has been done attempting to optimize various aspects of an application's execution by applying various graph transformation techniques to the application's SDFG. One of the more effective of these techniques is *retiming* [5], where delays

are redistributed among the edges so that hardware is optimized while the application's function remains unchanged. Retiming was initially applied to single-rate DFGs to optimize the application's schedule of tasks so that the *clock period* of the graph (i.e., the total computation time of the longest zero-delay path) was decreased in order for the application to be more efficiently scheduled for execution on multiprocessors. It was later extended to the more general SDFG model in order to extend vectorization capabilities or minimize the total delay count of a SDFG [10]. However, the problem of using retiming to minimize the clock period of a multirate DFG has remained unexplored. In this paper, we will discuss this problem and propose a method for accomplishing this task.

The benefits of retiming single-rate data-flow graphs are widely reported in the literature. Despite this, the application of retiming to SDFGs was explored only marginally prior to 1994 [4] before being studied by Zivojnovic *et al* primarily as a way to minimize the delay count of a SDFG [10]. In fact, the most popular method for retiming SDFGs has been to translate the SDFG to its single-rate equivalent and retime this new graph [3]. The possibility of then translating this new graph back to an equivalent retimed SDFG was mentioned in [10]. We will demonstrate the problem with this idea and show that is clearly preferable to work with the original SDFG as much as possible.

In this paper, we will develop the basic definitions and results necessary for specifying and manipulating a SDFG and its single-rate equivalent. We will review retiming and point out the problems which arise when it is applied to SDFGs. We will propose a polynomial-time algorithm which retimes a given SDFG to have a specified clock period. Finally, we will demonstrate the effectiveness of our algorithm by applying it to an example.

In the next section, we will formalize the fundamental concepts related to the study of synchronous data-flow graphs. We then discuss retiming and the problems we face as we apply it to SDFGs. Next is

our retiming algorithm, followed by a detailed example. Finally, we summarize our work and point to future directions for study.

## 2 Synchronous Data-Flow Graphs

The concept of a synchronous data-flow graph was developed and used extensively by Lee and Messerschmitt [4] and later by Zivojnovic *et al* [10]. In this section, we review their definitions and ideas in order to formalize these concepts.

A *synchronous data-flow graph (SDFG)* (sometimes called a *multirate* or *regular* data-flow graph) is a finite, directed, weighted graph  $G = \langle V, E, d, t, p, c \rangle$  where:  $V$  is the vertex set of nodes or *actors*, which transform input data streams into output streams;  $E \subseteq V \times V$  is the edge set, representing channels which carry data streams;  $d : E \rightarrow \mathbf{N} \cup \{0\}$  is a function with  $d(e)$  the number of initial tokens (*delays*) on edge  $e$ ;  $t : V \rightarrow \mathbf{N}$  is a function with  $t(v)$  the execution time of node  $v$ ;  $p : E \rightarrow \mathbf{N}$  is a function with  $p(e)$  the number of data tokens produced at  $e$ 's source node to be carried by  $e$ ; and  $c : E \rightarrow \mathbf{N}$  is a function with  $c(e)$  the number of data tokens consumed from  $e$  by  $e$ 's sink node. (In this definition  $\mathbf{N}$  is the set of natural numbers  $\{1, 2, 3, \dots\}$ .) If  $p(e) = c(e) = 1$  for all  $e \in E$ , we say that  $G$  is a *homogeneous data-flow graph (HDFG)*. HDFGs are also sometimes referred to as *single-rate data-flow graphs* or simply *data-flow graphs*.

To illustrate, consider the SDFG given in Figure 1(a) below. The numbers above the nodes represent the execution times for the individual tasks, while the smaller numbers at either end of an edge denote tokens produced or consumed. As an example,  $t(A) = 2$  while  $t(B) = t(C) = 1$  in the figure. Furthermore, the numbers at either end of the edge connecting  $A$  and  $B$  indicate that node  $A$  produces one token on this edge when it executes, while node  $B$  consumes two tokens from this edge each time it fires.

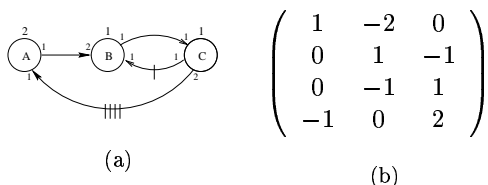


Figure 1: (a) A SDFG; (b) Its topology matrix  $M$

It is sometimes useful to characterize an SDFG by

its *topology matrix*, an  $|E| \times |V|$  matrix similar to an incidence matrix. Each row corresponds to one edge in the graph, while each column corresponds to a node. A positive  $(i, j)^{th}$  entry in the topology matrix indicates the number of tokens produced by the  $j^{th}$  node on the  $i^{th}$  edge, while a negative entry here gives the number of tokens consumed by node  $j$  from edge  $i$ . All other entries are zero. As an example, the topology matrix of Figure 1(a) is given in Figure 1(b).

In [4] it was demonstrated that a repeating sequential schedule can be constructed for a SDFG  $G$  if the rank of the graph's topology matrix is one less than the number of nodes in the SDFG. (The reverse is not necessarily true, as we will see shortly.) If this condition holds there is a positive integer vector  $q$  in the nullspace of the topology matrix called a *repetition vector* for  $G$ . The repetition vector for  $G$  with the smallest norm is called the *basic repetition vector (BRV)* for  $G$ . For example, the BRV for the SDFG in Figure 1(a) is  $q = [2 \ 1 \ 1]^T$ . The elements of a BRV  $q$  indicate that  $q_j$  copies of node  $v_j$  must be executed during every iteration of the static schedule. In our example we must schedule two copies of  $A$  and one copy each of  $B$  and  $C$  each time.

In order to study an SDFG, it is sometimes useful to create its *equivalent homogeneous data-flow graph (EHG)*. As the name implies, an EHG performs the same function as the original SDFG, but is constructed so that each edge carries at most one token. Since each node is expecting to either produce or consume more data than this, an EHG compensates by inserting multiple edges between nodes.

An algorithm for creating a graph's EHG appears in [6]. (Similar methods are outlined in [1] and [9].) It first creates enough copies of each node to satisfy the specifications of the BRV. It then inserts edges. If nodes in a SDFG are connected by a zero-delay edge, then the first data token produced by the first copy of the source must be consumed by the first copy of the sink in the EHG. If there are delays on an edge, the data contained here is consumed first, so that the first new token produced is in fact needed by a later copy of the sink. The algorithm determines which copies of source and sink to map to one another based on how much data has been created and used. As an example of our algorithm in action, the EHG of Figure 1(a) appears in Figure 2.

A *path* in either a SDFG or a HDFG is any sequence of nodes and edges. The *clock period*  $cl(G)$  of a HDFG  $G$  is then defined to be the length of the longest zero-delay path. As discussed in [6], we are forced to define the clock period of a SDFG to be equal to the clock

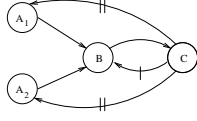


Figure 2: Figure 1(a)'s EHG

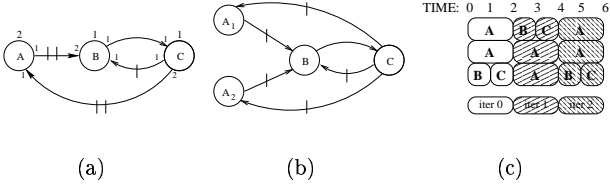


Figure 3: (a) Figure 1(a) retimed; (b) Its EHG; (c) Its repeating schedule

period of its EHG due to problems raised by the nature of synchronous graphs. As an example, the clock period of the SDFG in Figure 1(a) is 4 by our definition.

Similar problems arise when we attempt to minimize the clock period. We will say that an *iteration* of a SDFG is the execution of all nodes of its EHG once. The average computation time of one iteration is then called the *iteration period* of the SDFG and is equal to the iteration period of the EHG. (In Figure 1(a) the iteration period is also 4.) If a SDFG contains a loop, then the iteration period is bounded from below by the *iteration bound* [8], which is defined to be the maximum time-to-delay ratio of all cycles in the EHG. For example, the EHG in Figure 2 contains three loops:  $(A_1, B, C)$  and  $(A_2, B, C)$  each with total computation time of 4 and delay count 2; and  $(B, C)$  with computation time 2 and delay count 1. Thus the iteration period of the graph in Figure 1(a) is 2.

A *retiming*  $r : V \rightarrow \mathbf{N} \cup \{0\}$  is a function which specifies a transformation of a graph  $G$ . It labels each vertex with a number of delays to be removed from each incoming edge and placed on each outgoing edge, changing  $G$  into the retimed graph  $G_r = \langle V, E, d_r, t, p, c \rangle$  where  $d_r(e) = d(e) + p(e)r(u) - c(e)r(v)$  for each edge  $e = (u, v)$  in  $E$  [10]. As an example, a retiming with  $r(A) = 2$  and  $r(B) = r(C) = 0$  transforms the SDFG of Figure 1(a) into that of Figure 3(a). Examining the EHG in Figure 3(b), we see that we have now achieved an optimal clock period of 2 which translates into the more compact schedule of Figure 3(c).

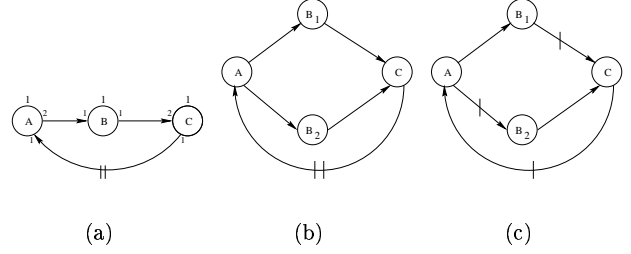


Figure 4: (a) A unit-time SDFG; (b) Its EHG; (c) Its retimed EHG

On first glance, it appears that we should just be able to retime the EHG via traditional methods and then map back to the original SDFG. Unfortunately, the initial translation from SDFG to EHG is too complex to permit this. As an example, consider the unit-time SDFG given in Figure 4(a), with its EHG appearing in Figure 4(b). A retiming with  $r(A) = r(B_1) = 1$  and  $r(B_2) = r(C) = 0$  transforms the EHG into the graph shown in Figure 4(c) with clock period 2. We now wish to try and match this with some retimed version of the original SDFG, but have a problem with the delay count of the edge between  $A$  and  $B$ . If the new delay count is 1, the EHG should have no delay on the edge  $(A, B_2)$  and one delay on  $(A, B_1)$ , exactly the opposite of what we actually have. On the other hand, if the retimed delay count is 2 or more, then both  $(A, B_1)$  and  $(A, B_2)$  should have non-zero delay counts, which also contradicts what we have. In any case, there can be no direct matching in this case. If we are to retime SDFGs, we must work directly on the original graph itself.

Since we cannot retime a SDFG by working with its EHG, we must develop a method for retiming SDFGs directly. We construct a mathematical solution based entirely on this idea which is proven in [6]:

**Theorem 2.1** *Let  $G = \langle V, E, d, t, p, c \rangle$  be a SDFG with BRV  $q$ . Let  $c$  be a potential clock period. Let  $H$  be the EHG for  $G$ . Define  $D(u, v)$  to be the minimum number of delays along any path from  $u$  to  $v$  in  $H$ , and  $T(u, v)$  to be the maximum total computation time of any path from  $u$  to  $v$  with delay count  $D(u, v)$  in  $H$ . Then  $r$  is a legal retiming of  $G$  with  $cl(G_r) \leq c$  if*

1.  $c(e)r(v) - p(e)r(u) \leq d(e)$  for all edges  $e : u \rightarrow v$  in  $G$ ;
2.  $c(e)r(w) - p(e)r(u) \leq d(e) - q_u \cdot p(e)$  for all paths

$\pi : u \Rightarrow v$  with  $T(u, v) - t(u) \leq c < T(u, v)$ , where  $e : u \rightarrow w$  is the edge in  $\pi$  with source node  $u$ ;

3.  $c(e)r(v) - p(e)r(w) \leq d(e) - q_v \cdot c(e)$  for all paths  $\pi : u \Rightarrow v$  with  $T(u, v) - t(v) \leq c < T(u, v)$ , where  $e : w \rightarrow v$  is the edge in  $\pi$  with source node  $w$ .

This result leads to an algorithm which may perform wasteful operations; it may be possible that the delay counts of the other edges in the path are sufficient so that retiming is unnecessary, but we will retime anyway. Because of this result, we can construct a system of linear inequalities which can be solved in polynomial time by the Bellman-Ford Algorithm [2]. Furthermore, because we are working with values derived from the EHG, we will avoid the false path problem discussed in [6].

Making use of this idea requires us to calculate  $T$  and  $D$ , the maximum computation time and minimal delay counts along paths between nodes, respectively. In [6] we construct an algorithm based on the method of [5], building a matrix and manipulating it via the Floyd-Warshall all-pairs shortest-path algorithm [2] to compute these values for an EHG. Once we have these figures, we compute  $T$  and  $D$  for the original SDFG by setting

$$D(u, v) = \min \{D(u_i, v_j)\}$$

and

$$T(u, v) = \max \{T(u_i, v_j) | D(u_i, v_j) = D(u, v)\}$$

where  $u_i$  is a copy of  $u$  and  $v_j$  is a copy of  $v$  in the EHG. We are forced to work with the EHG due to the problems we noted in [6] regarding the calculation of a path's delay count.

As an example of our method in action, consider Figure 5 below with  $c = 4$ . (Since all nodes in the graph take 4 time units to execute, this is the smallest clock period we can hope to achieve.) The  $T$  and  $D$  values derived from its EHG are given in [6]; suffice it to say that, for each pair of vertices  $u$  and  $v$  connected by an edge in the SDFG,  $T(u, v) = 8$  and  $D(u, v) = 0$  except that  $D(D, A) = 7$ . Let us begin by attempting to satisfy the second and third conditions of Theorem 2.1. Since all nodes have computation time 4 in this example, we need only consider paths with total computation time 8, which includes all edges in this graph. We derive 5 inequalities, one for each of the paths (edges) from  $A$  to  $B$ ,  $A$  to  $C$ ,  $B$  to  $D$ ,  $C$  to  $D$  and  $D$  to  $A$ :

$$2r(B) - 3r(A) \leq -6$$

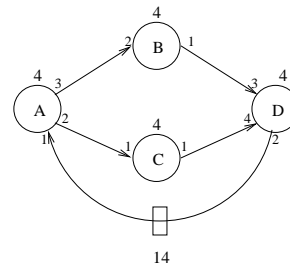


Figure 5: An example SDFG

$$\begin{aligned} r(C) - 2r(A) &\leq -4 \\ 3r(D) - r(B) &\leq -3 \\ 4r(D) - r(C) &\leq -4 \\ r(A) - 2r(D) &\leq 12 \end{aligned}$$

Condition 1 also gives us 5 inequalities to satisfy based on edges in the original SDFG. However each of these inequalities is replaced by one of the tougher restrictions we have just seen, so it suffices to consider only the constraints derived above.

Before we can proceed, we must multiply each of our equations by properly chosen constants so that the coefficients of each variable occurrence match. Completing this exercise leaves us with the system

$$\begin{aligned} 4r(B) - 6r(A) &\leq -12 \\ 3r(C) - 6r(A) &\leq -12 \\ 12r(D) - 4r(B) &\leq -12 \\ 12r(D) - 3r(C) &\leq -12 \\ 6r(A) - 12r(D) &\leq 72 \end{aligned}$$

As in [2], we can then model our set of inequalities by the constraint graph of Figure 6(a) and find the lengths of the shortest paths from  $v_0$  to all other nodes to get an answer of  $6r(A) = 0$ ,  $4r(B) = -12$ ,  $3r(C) = -12$  and  $12r(D) = -24$ . Since we prefer positive retimings, we normalize this answer by adding 24 to all values. Note that we are permitted to do this because adding such a constant to all values has no effect on our system of inequalities; it merely presents us with an alternate solution of  $6r(A) = 24$ ,  $4r(B) = 12$ ,  $3r(C) = 12$  and  $12r(D) = 0$ . We then divide to produce our final answer of  $r(A) = 4$ ,  $r(B) = 3$ ,  $r(C) = 4$  and  $r(D) = 0$ . (The sequence of events at this step is crucial; the reader can verify that dividing and then normalizing yields an incorrect answer.) Applying this function to the SDFG of Figure 5 yields the retimed graph of Figure 6(b). An examination of this EHG reveals that we have indeed found a retiming which



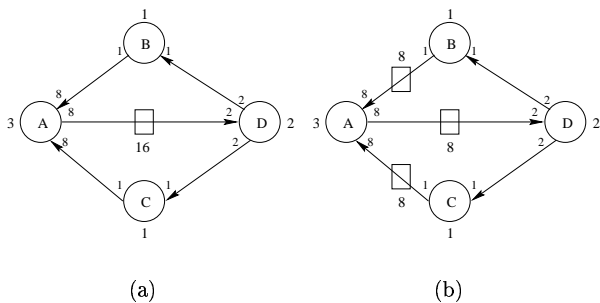


Figure 8: (a) Another sample SDFG; (b) Figure 8(a) retimed

Finally, Theorem 2.1's third condition is satisfied by two paths, neither of which contributes anything new. In short, we have five inequalities which must be solved in order to find our retiming. We now solve our system for an initial answer of  $8r(A) = -8$  and  $r(B) = r(C) = 2r(D) = 0$ . Adding and then dividing yields a function with  $r(A) = 0$ ,  $r(B) = r(C) = 8$  and  $r(D) = 4$ . Applying this retiming to the SDFG in Figure 8(a) produces the retimed graph in Figure 8(b) which executes within our desired time interval.

## 4 Conclusion

In this paper, we have established a notation for expressing and studying synchronous data-flow graphs. We have presented the difficulties involved with retiming SDFGs, and then constructed a polynomial-time algorithm for retiming a synchronous graph so that it achieves a sufficiently small clock period. Finally, we have demonstrated the effectiveness of our algorithm on examples. Note that, in all cases we have studied in this paper, we have been able to achieve minimal execution times.

## 5 Acknowledgement

This work is partially supported by NSF grants MIP95-01006 and MIP97-04276, and by the A.J. Schmitt Foundation.

## References

- [1] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. Signal Process.*, 44:397–408, 1996.
- [2] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, Inc., 1991.
- [3] E.A. Lee. *A Coupled Hardware and Software Architecture for Programmable Digital Signal Processors*. PhD thesis, Dept. of EECS, Univ. of California at Berkeley, 1986.
- [4] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data-flow programs for digital signal processing. *IEEE Trans. Comput.*, 36:24–35, 1987.
- [5] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
- [6] T.W. O'Neil and E. H.-M. Sha. Retiming synchronous data-flow graphs to minimize execution time. Technical Report 00-08, Univ. of Notre Dame, 2000. Available online at [www.cse.nd.edu/tech\\_reports/](http://www.cse.nd.edu/tech_reports/).
- [7] K.K. Parhi. Algorithm transformation techniques for concurrent processors. *Proc. IEEE*, 77:1879–1895, 1989.
- [8] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed. *Trans. Circuits & Sampling*, CAS-28:196–202, 1981.
- [9] G.C. Sih. *Multiprocessor Scheduling to Account For Interprocessor Communication*. PhD thesis, Dept. of EECS, Univ. of California at Berkeley, 1991.
- [10] V. Zivojnovic, S. Ritz, and H. Meyr. Optimizing DSP programs under the multirate retiming transformation. In *Proc. 7th European Signal Process. Conf.*, volume 3, pages 1597–1600, 1994.
- [11] V. Zivojnovic and R. Schoenen. On retiming of multirate DSP algorithms. In *Proc. ICASSP-96*, volume 6, pages 3310–3313, 1996.