

# PARALLELIZING SYNCHRONOUS DATA-FLOW GRAPHS VIA RETIMING

TIMOTHY W. O'NEIL

*Dept. of Comp. Sci. & Eng., University of Notre Dame, Notre Dame, IN 46556, USA  
E-mail: toneil@cse.nd.edu*

EDWIN H.-M. SHA

*Dept. of Comp. Science, Erik Jonsson School of Eng. & C.S., Box 830688, MS EC 31, Univ. of  
Texas at Dallas, Richardson, TX 75083-0688, USA  
E-mail: esha@cse.nd.edu*

SISSADES TONGSIMA

*High Perf. Comp. Lab., Nat'l Electronics & Comp. Tech. Center, 11th Floor Bangkok Thai  
Tower Bldg., 108 Rangnam Road, Phyathai, Rachathewee, Bangkok 10400, THAILAND  
E-mail: stongsim@hpcc.nectec.or.th*

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. In this paper, we do just this. We develop the basic definitions and results necessary for expressing and studying SDFGs. We review the problems faced when attempting to retime a SDFG in order to minimize clock period, then present an algorithm for doing this. Finally, we demonstrate the effectiveness of our method on several examples.

## 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 [1]. 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 [2] with its use leading to numerous important results regarding the scheduling [3], hierarchization [4], vectorization [5] and multiprocessor allocation [1] of DSP programs.

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* [6, 7], 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 sched-

uled for execution on multiprocessors [8]. It was later extended to the more general SDFG model in order to extend vectorization capabilities [9] 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.

In addition to its ability to reduce clock period, scheduling alone typically yields a schedule requiring more resources than the schedule produced by retiming first. (Both of these ideas are elaborated upon in [11].) While the benefits are clear, reworking our retiming methods so that they may be applied to synchronous graphs is not easy. The difference between the single-rate and multi-rate models lies in the specification of production and consumption rates on each edge; in single-rate graphs all such rates are assumed to be the same, whereas different rates for different edges are typically specified when constructing SDFGs. Two pitfalls were noted in [12]. First of all, a retiming may be derived for a single-rate DFG by solving a linear programming problem [7]. The introduction of rates on the edges potentially changes this to a more complicated integer linear programming problem. Second, the introduction of rates invalidates the traditional results regarding the delay counts of paths and cycles, depriving us of many useful results derived for the single-rate case. Specifically, in the single-rate case, we seek to remove zero-delay path with excessive total computation times. It isn't clear what we want to avoid in the multi-rate case; a specific delay count on one path may or may not be adequate, depending on what rates have been specified.

Finally, the most popular method for retiming SDFGs has been to translate the SDFG to its single-rate equivalent, retime this new graph, then translate back [10,13]. There are two problems with this idea. First, as we will demonstrate, it may be impossible to translate a retimed single-rate graph back to a retimed SDFG. Second, even if this method works, the costs in performing the necessary translations and dramatically increasing our problem size may be prohibitive. It 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 several examples.

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 [1], but was not rigorously defined until the work of Zivojnovic *et al* [10, 14]. In this section, we review their definitions and ideas in order to formalize these concepts.

## 2.1 Basic Definitions

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.

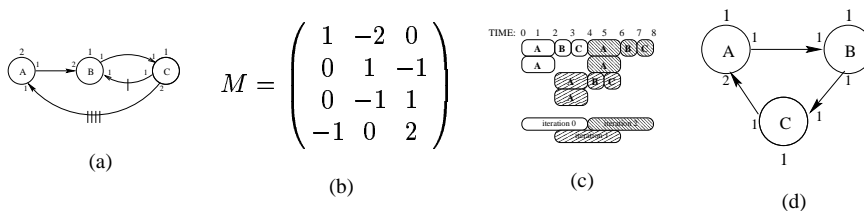


Figure 1: (a) A SDFG; (b) Its topology matrix; (c) Its repeating schedule; (d) An inconsistent SDFG

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 [1] 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  $G$  [15]. 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; see Figure 1(c). Finally, a SDFG is *consistent* if it has a BRV. An example of an inconsistent SDFG with rank 3 topology matrix appears as Figure 1(d). It is clear that, if we attempt to execute this circuit, each node will fire once before node  $A$  deadlocks the system waiting for its second token.

## 2.2 Constructing an Equivalent HDFG

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 [11]. It is adapted from the method of [15] for constructing the EHG of *cyclostatic* DFGs, which not only permit multiple tokens to pass along edges but also specifies the pattern of their production or consumption. The algorithm 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).

There are two significant differences between our algorithm and that of [15]. First, the original algorithm was more concerned with making sure that the amount of data produced and consumed on an edge matched. This yields a simpler but more confusing graph. For purposes of clarity, we do not combine edges between nodes. If multiple tokens are to be sent between nodes in the EHG, each travels along its own edge. One benefit is that the delay counts between the original SDFG and the EHG match in our model. More significantly, the original algorithm also inserted control dependencies into the EHG, insuring that all copies of a node execute serially. Since we are concerned with maximizing parallelism, we concern ourselves only with the necessary data dependencies.

Finally, as derived in [15], we will say that a SDFG is *live* if its EHG has no zero-weight cycles. Otherwise the graph is *deadlocked*. An example of a consistent deadlocked graph appears as Figure 2(b), with its EHG in Figure 2(c). As we can see, the loop between nodes  $A$  and  $B_2$  contains no delays, and so it is impossible to schedule them since each must precede the other. It should be clear that a SDFG must be both live and consistent in order for it to have a repeating static schedule.

## 3 Retiming

A great deal of research has been done attempting to optimize the schedule of an application's tasks after applying various graph transformation techniques to the application's HDFG. One of the more effective of these techniques is *retiming* [6, 7], where delays are redistributed among the edges so that the application's function remains the same while the execution time decreases.

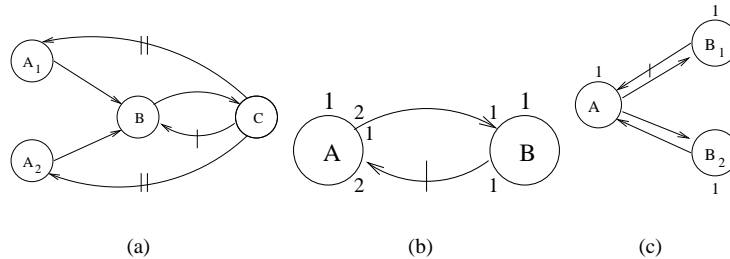


Figure 2: (a) Figure 1(a)'s EHG; (b) A deadlocked SDFG; (c) Its EHG

Despite its usefulness when applied to HDFGs, the application of retiming to SDFGs was explored only marginally prior to 1994 [13, 16] before being studied by Zivojnovic *et al* primarily as a way to minimize the delay count of a SDFG [10, 12]. In this section we intend to review the basics of retiming, explore some of the pitfalls which arise when studying retiming of SDFGs, demonstrate the effectiveness of retiming, and propose two algorithms for retiming SDFGs.

### 3.1 Basic Definitions

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 [8]. This definition is problematic on two counts. First, it is not clear what the delay count of a path in a SDFG really is in light of the inconsistencies in the results from [10]. Second, suppose that we attempt to apply our definition directly to SDFGs, as demonstrated by Figure 2(b). We would conclude that the clock period equals 2, but in reality the graph must have an infinite clock period because of the problems scheduling nodes  $A$  and  $B_2$ . Thus, we are forced to define the clock period of a SDFG to be equal to the clock period of its EHG. 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* [17], which is defined to be the maximum time-to-delay ratio of all cycles in the EHG. For example, the EHG in Figure 2(a) 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. This can be clearly seen from the schedule in Figure 1(c), where overlapped iterations create higher throughput. (The iteration period of an SDFG can be overestimated using the ideas from [14] without constructing the EHG, but our method yields a tighter bound, which is important as we attempt to minimize the iteration period of an SDFG next.)

A *retiming*  $r : V \rightarrow \mathbb{N} \cup \{0\}$  is a function which specifies a transformation of a graph  $G$ . It

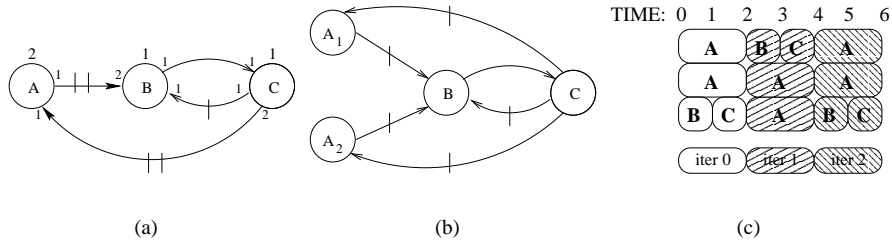


Figure 3: (a) Figure 1(a) retimed; (b) Its EHG; (c) Its repeating schedule

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, 12]. 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).

### 3.2 Problems Retiming 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, as was done by Lee originally [13]. 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.

## 4 Retiming a SDFG

Since we cannot retime a SDFG by working with its EHG, we must develop methods for retiming the SDFG directly. In this section we refine the methods of [7] to deal with this situation.

### 4.1 Initial Problems

Unfortunately, the retiming algorithms we will propose will either be pessimistic or expensive. The reason for this is that the original methods we are using as a basis were themselves built on one result from [7]:

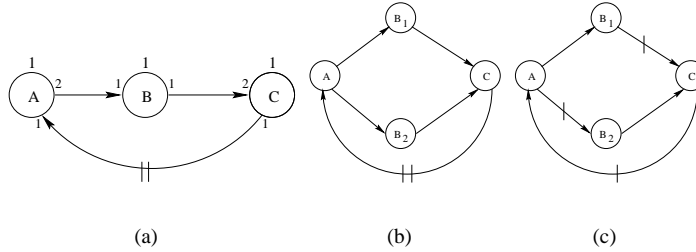


Figure 4: (a) A unit-time SDFG; (b) Its EHG; (c) Its retimed EHG

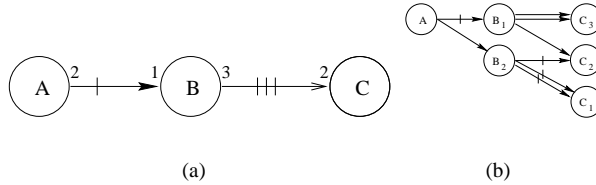


Figure 5: (a) A path in a SDFG; (b) Its homogeneous equivalent

**Thm 4.1** Let  $G$  be a HDFG and  $c$  a potential clock period.  $cl(G) \leq c$  iff every path in  $G$  with total computation time larger than  $c$  has delay count larger than 1.

The problem now is that insufficient delays along a path in a SDFG do not necessarily translate into a zero-delay path in the EHG. As an example, consider the unit-time SDFG in Figure 5(a) below, with its EHG given in Figure 5(b). For  $c = 2$ , examining only the original SDFG would lead us to retime this path even though such an exercise is unnecessary. To avoid such false paths, we may need to construct intermediate EHG's for study, a very costly process.

In a similar vein, the nature of an EHG raises the question of what a path actually is. The traditional definition says that a path is a sequence of nodes and edges. Since we now have multiple edges between nodes, we must be very careful to consider all paths resulting from such multiple copies. To illustrate, the traditional definition would dictate that there is one path from  $A$  to  $C_3$  in Figure 5(b). Because of the pair of edges between  $B_1$  and  $C_3$ , we will abuse our definition slightly and say that there are in fact two paths between  $A$  and  $C_3$  in the EHG when we do our calculations below. While this makes sense, it is somewhat different from what has always been done and so must be noted.

Another additional cost that the problem of insufficient delays forces us to pay comes in the form of additional checks for legality. In the original algorithms from [7], only one delay at a time was moved, a stipulation which did not cause the proposed retiming to become illegal at any intermediate step (as proven in [7]). Because we are now pulling groups of delays through nodes, this situation no longer exists, and so we will have to check for legality at every stage of

an algorithm.

The question now is to determine exactly how many delays to view as sufficient. Let  $e : u \rightarrow v$  be an edge in a SDFG. Each copy of  $u$  in the EHG creates  $p(u)$  tokens. By our construction each of these is to travel along a separate edge. Since there are  $q_u$  copies of node  $u$  in the EHG, there must then be a total of  $q_u \cdot p(e)$  edges to carry all of the data, each of which we expect to require a delay when we retime the graph. Similarly,  $q_v$  copies of  $v$  are each receiving  $c(e)$  tokens, and so there must be  $q_v \cdot c(e)$  edges for these data. We will use either of these figures as the number of tokens required by an edge in the SDFG during retiming.

#### 4.2 Retiming Algorithm

We will seek our retiming via *relaxation* on the edges of our graph. We do this by sorting our vertices and then sweeping along the sorted list. When we get to a point where the current path is too long, we insert enough delays so as to break the path up into sufficiently small pieces. We then verify that we are allowed to do this. If we cannot then there is no retiming and we return with an error; otherwise we sweep further. Once our prospective retiming has been found, we test the retimed graph to make sure that the clock period is within our requirements. If it is we have found a way to retime the SDFG; otherwise there is no such retiming.

We begin our construction by considering Algorithm 1 below, the  $O(|E|)$ -time algorithm from [7] for finding the length of the longest zero-delay path into each vertex of a HDFG. This procedure first sorts the vertices so that those occurring early in the list are connected to vertices later in the list by zero-delay edges. It then traces through the list, associating each vertex with the length of its longest zero-delay path. If a vertex is not connected to a previous one, its path length must equal its own computation time; otherwise its path length equals its own time, plus the sum of the times of all the other vertices found along the path to this point. We require this algorithm not just for constructing our retiming, but also for verifying that our final retimed graph executes within the required time frame.

With this in hand we can now proceed to our primary method, given as Algorithm 2. We begin by retiming our SDFG with the result to date and constructing its EHG. The EHG is then handed to Algorithm 1 to find the lengths of the maximum paths to all vertices. At this point, if the longest path length is sufficiently small, we return our current retiming function as the final answer. Otherwise the vertices in our SDFG fall into one of two groups. If all copies of a vertex

---

#### **Algorithm 1** Find computation time of most expensive zero-delay path to all vertices

---

**Input:** A HDFG  $G = \langle V, E, d, t \rangle$

**Output:** The  $|V|$ -length vector  $\tau$

Topologically sort the vertices of  $G$ , with  $u$  preceding  $v$  if there is a zero-delay edge from  $u$  to  $v$  in  $G$

**for all**  $v$  in order from the sorted list **do**

**if**  $v$  has no zero-delay incoming edge in  $G$  **then**

$\tau(v) \leftarrow t(v)$

**else**

$\tau(v) \leftarrow t(v) + \max\{\tau(u) \mid \exists e : u \rightarrow v \text{ in } G \text{ with } d(e) = 0\}$

**end if**

**end for**

**return**  $\tau$

---

---

**Algorithm 2** Retime a SDFG via Relaxation

---

**Input:** A SDFG  $G = \langle V, E, d, t, p, c \rangle$ , a potential clock period  $c$

**Output:** A retiming  $r$  such that  $cl(G_r) \leq c$  if one exists

```
for all  $v \in V$  do
   $r(v) \leftarrow 0$ 
end for
for  $i \leftarrow 1$  to  $|V|$  do
  Construct the retimed graph  $G_r$  given the current  $r$ 
  Construct the EHG  $H = \langle V', E', d', t' \rangle$  for  $G_r$ 
   $\tau \leftarrow \text{MaxPath}(H)$  /* Apply Algorithm 1 */
  if  $\max\{\tau(v) | v \in V'\} \leq c$  then
    return  $r$  /* All path lengths small enough; stop. */
  end if
  for all  $v$  in  $V$  do
    if no copy of  $v$  in  $H$  is incident on a zero-delay edge in  $H$  then
       $\Upsilon(v) \leftarrow \infty$ 
    else
       $\Upsilon(v) \leftarrow \max\{\tau(v_i) | v_i \text{ is a copy of } v \text{ in } H\}$ 
    end if
  end for
  for all  $v$  with  $\Upsilon(v) \leq c$  do
     $r(v) \leftarrow r(v) + \max\left\{\left\lceil \frac{q_v \cdot p(e) - d_r(e)}{p(e)} \right\rceil \mid e : v \rightarrow u \text{ in } G_r \text{ for some } u\right\}$ 
  end for
  for all  $e : u \rightarrow v$  in  $E$  do
    if  $d(e) + p(e)r(u) - c(e)r(v) < 0$  then
      return FALSE /* Retiming illegal; return with error */
    end if
  end for
end for
Construct the retimed graph  $G_r$  given the current  $r$  /* Determine clock period */
Construct the EHG  $H$  for  $G_r$ 
 $\Upsilon \leftarrow \text{MaxPath}(H)$  /* Apply Algorithm 1 again */
if  $\max\{\Upsilon(v) | v \in V'\} > c$  then
  return FALSE /* No feasible retiming */
else
  return  $r$  /* Otherwise  $r$  is the retiming */
end if
```

---

in the EHG are isolated (i.e., connected to the rest of the graph only by edges containing delays), we do not wish to retime the node and remove it from consideration. Otherwise the node lies along some zero-delay path and we may have to retime it. In this case we assign it a longest path length equal to the longest path length of any of its copies in the EHG.

We now consider nodes for further retiming. Since we want to push delays forward along our paths (rather than pulling them backward as was done in [7]), we retime those nodes which occur early in a path. This process is complicated by the different rates of production and consumption on each node. For example, for each delay drawn into node  $A$  in Figure 6(a), three delays are pushed onto the edge from  $A$  to  $B$  and two delays onto the edge from  $A$  to  $C$ . Therefore, for each outgoing edge from such a node, we calculate the number of delays needed to retime all

copies of the edge in the EHG, subtract the number of delays currently on the edge, adjust for the different rates of production and consumption, and retime by the maximum of these needs. Once all nodes are retimed, we test the prospective retiming for legality, i.e. we check that retiming by our function doesn't result in some edge containing a negative number of delays. If we pass this test, we look further along our path for other nodes in need of retiming.

Once we have checked all nodes at least once and have derived a legal retiming function, it is time to test our answer. We repeat our earlier steps to find the lengths of the maximum zero-delay paths to each node one last time. Since the length of the largest zero-delay path in the EHG equals our clock period, this value is tested against our requested clock period. If it is still too large we cannot retime this SDFG to execute in the time we wish and must return with an error. Otherwise we have found our retiming.

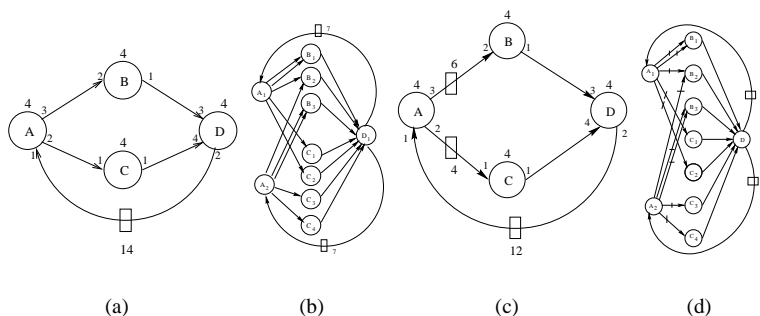


Figure 6: (a) An example SDFG; (b) Its EHG; (c) The retimed SDFG after first pass; (d) Its EHG

We now demonstrate our method by executing it on the SDFG of Figure 6(a) with  $c = 4$ . Sorting the vertices of Figure 6(b), computing longest path lengths and taking the maxima reveals that  $\Upsilon(A) = 4$ ,  $\Upsilon(B) = \Upsilon(C) = 8$  and  $\Upsilon(D) = 12$  in this case. We thus only retime node  $A$  at this step. Since there are no delays on the edge  $(A, B)$ , our initial retiming has  $r(A) = q_A = 2$  and  $r(v) = 0$  for any other node  $v$ . Pulling 2 delays through  $A$  pushes 6 delays onto  $(A, B)$  and 4 onto  $(A, C)$ , as seen in the retimed graph in Figure 6(c), with its EHG appearing in Figure 6(d).

Looping back around, we see from Figure 6(d) that only node  $A$  is cut off; all other nodes lie along some zero-delay path. Thus  $\Upsilon(A) = \infty$ ,  $\Upsilon(B) = \Upsilon(C) = 4$  and  $\Upsilon(D) = 8$  now, calling for us to retime nodes  $B$  and  $C$ . Since neither of the edges  $(B, D)$  nor  $(C, D)$  currently contains delays, our retiming now has  $r(A) = 2$ ,  $r(B) = q_B = 3$ ,  $r(C) = q_C = 4$  and  $r(D) = 0$ . The new retimed graph is given below as Figure 7(a), with its EHG in Figure 7(b). Note that, due to node  $B$ 's consumption rate of 2, a retiming of 3 applied to  $B$  requires us to pull  $2 \times 3 = 6$  delays in so that the proper number of delays is pushed back out.

Studying the new EHG shows us that node  $D$  is now cut off, but node  $A$  requires further retiming. We have  $\Upsilon(A) = 4$ ,  $\Upsilon(B) = \Upsilon(C) = 8$  and  $\Upsilon(D) = \infty$  now, so only node  $A$  must be retimed. Since both of the edges  $(A, B)$  and  $(A, C)$  are devoid of delays, we must add  $q_A = 2$  onto the retiming for  $A$ , giving us the function  $r(A) = 4$ ,  $r(B) = 3$ ,  $r(C) = 4$  and  $r(D) = 0$ .

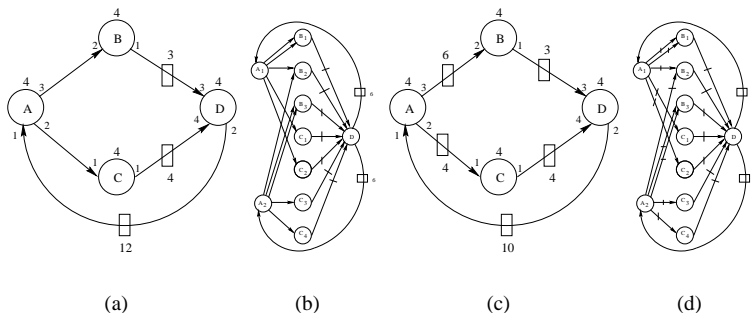


Figure 7: (a) The retimed SDFG after two passes; (b) Its EHG; (c) Figure 6(a) retimed; (d) Its EHG

The application of this retiming to the original SDFG results in the graph of Figure 7(c) and we have found our answer.

However we have a final pass of the algorithm to perform. This time, all nodes in the SDFG have been isolated, so  $\Upsilon(v) = \infty$  for all nodes  $v$ , insuring that no further retiming takes place. We now study the EHG of Figure 7(d), find that the maximum zero-delay path is an individual node with computation time 4 and conclude that we have found our retiming.

Let  $G = \langle V, E, d, t, p, c \rangle$  be our SDFG with EHG  $H = \langle V', E', d', t' \rangle$ . Constructing the EHG and executing Algorithm 1 each execute in  $O(|E'|)$  time; thus Algorithm 2 only requires  $O(|V|^2 + |V||E'|)$  time. However, while we suspect that its success is both a necessary and sufficient condition for a SDFG to be retimable to a given clock period, it is unknown whether or not this is the case. In our defense, the algorithm from [7] upon which this method is based was also never proven both necessary and sufficient, but has been extremely useful in practice. We suspect that the algorithm we've described here will prove just as valuable despite this logical gap. Let  $G = \langle V, E, d, t, p, c \rangle$  be our SDFG with EHG  $H = \langle V', E', d', t' \rangle$ . Since constructing the EHG and executing Algorithm 1 each require  $O(|E'|)$  time, Algorithm 2 only requires  $O(|V||V'| + |V||E'|)$  time to complete. However, while we suspect that its success is both a necessary and sufficient condition for a SDFG to be retimable to a given clock period, it is unknown whether or not this is the case. In our defense, the algorithm from [7] upon which this method is based was also never proven both necessary and sufficient, but has been extremely useful in practice. We suspect that the algorithm we have described here will prove just as valuable despite this logical gap.

## 5 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 an example. (The interested reader may find further examples in [11].)

Regardless of how good our algorithm may be, it is still not proven to represent both a necessary and sufficient condition for retiming. This proof, or the construction of an alternate method which is necessary and sufficient, remain interesting open problems. Correcting the errors in [10] will definitely lead to greater understanding of our model and may open the door to removing this logical gap. It may also lead to a study of retiming applied to even more complicated models, such as cyclo-static or dynamic DFGs [15].

### Acknowledgements

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

### References

1. 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.
2. J.L. Pino, S. Ha, E.A. Lee, and J.T. Buck. Software synthesis for DSP using Ptolemy. *J. VLSI Signal Process.*, 9:7–21, 1995.
3. G. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow programs for DSP computation. In *Proc. ICASSP-92*, volume 5, pages 561–564, 1992.
4. S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proc. Int. Conf. Application-Specific Array Processors*, pages 285–296, 1993.
5. S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr. High-level software synthesis for the design of communication systems. *IEEE J. Selected Areas in Communications, Special Issue on Comput.-Aided Modeling, Analysis & Design of Communication Links*, 11:348–358, 1993.
6. C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. *J. VLSI & Comput. Syst.*, 1(1):41–67, 1983.
7. C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
8. L.-F. Chao. *Scheduling and Behavioral Transformations for Parallel Systems*. PhD thesis, Dept. of Comp. Science, Princeton University, 1993.
9. V. Zivojnovic, S. Ritz, and H. Meyr. Retiming of DSP programs for optimum vectorization. In *Proc. ICASSP-94*, volume 1, pages 465–468, 1994.
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. 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/).
12. V. Zivojnovic and R. Schoenen. On retiming of multirate DSP algorithms. In *Proc. ICASSP-96*, volume 6, pages 3310–3313, 1996.
13. 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.
14. R. Schoenen, V. Zivojnovic, and H. Meyr. An upper bound of the throughput of multirate multiprocessor schedules. In *Proc. ICASSP-97*, volume 1, pages 655–658, 1997.
15. G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. Signal Process.*, 44:397–408, 1996.
16. K.K. Parhi. Algorithm transformation techniques for concurrent processors. *Proc. IEEE*, 77:1879–1895, 1989.
17. M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed. *Trans. Circuits & Sampling*, CAS-28:196–202, 1981.