

# Retiming Synchronous Data-Flow Graphs to Reduce Execution Time\*

\*Timothy W. O'Neil  
Dept. of Comp. Science & Eng.  
University of Notre Dame  
Notre Dame, IN 46556

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

## 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. 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 algorithms for doing this. Finally, we demonstrate the effectiveness of our methods on several examples.

**EDICS Class:** 5-PROG

---

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

# 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 [15]. 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 [11, 13, 20, 26] with its use leading to numerous important results regarding the scheduling [9], hierarchization [23], vectorization [22] and multiprocessor allocation [10, 15] 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* [17, 18], 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 [4–6]. It was later extended to the more general SDFG model in order to extend vectorization capabilities [28] or minimize the total delay count of a SDFG [27]. 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.

To illustrate the benefits of the retiming transformation, consider the single-rate DFG in Figure 1(a). The numbers above the nodes represent computation times of individual tasks, while the short bar-lines cutting the edges are the delays. We can see that our clock period in this case is 4 due to the zero-delay edge from  $A$  to  $B$ , hereafter referred to as  $(A, B)$ . Retiming allows us to remove a delay from  $(C, A)$  and place it on  $(A, B)$  to create the retimed graph in Figure 1(b) with clock period 3. The function of the two graphs is the same, with the only complication being that we will have to provide the first value of  $A$  when we begin execution. The costs of doing this are miniscule when we consider that we will be saving a clock cycle each time we execute the loop.

An additional benefit is that scheduling alone may yield a schedule requiring more resources than the schedule produced by retiming first. To illustrate this, consider the single-rate data-flow graph in Figure 2(a). If we apply the DFG Scheduling Algorithm from [5], we derive the schedule in Figure 2(b). Since the repeating part of this schedule of tasks repeats every 4 clock cycles, we say that this is a static schedule with *cycle period* 4. Note that this schedule requires a minimum of 5 processing units to execute because of the work called for at any time-step greater than zero which is a multiple of 4.

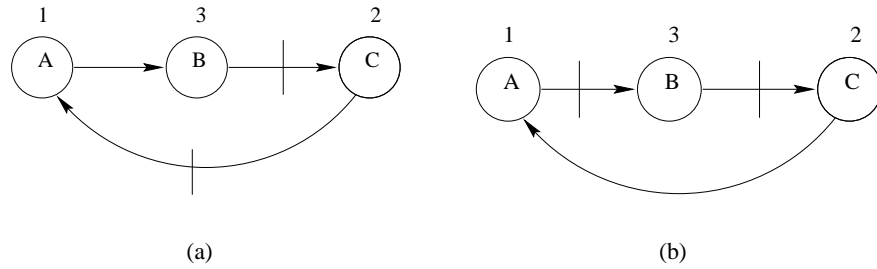


Figure 1: (a) A single-rate data-flow graph; (b) This DFG retimed to have clock period 3

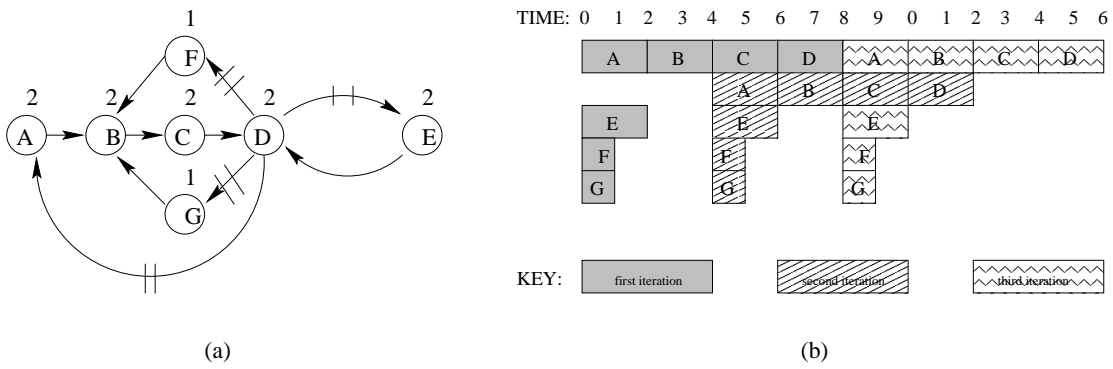


Figure 2: (a) A sample DFG; (b) Its schedule with cycle period 4

On the other hand, suppose that we retime our graph to become the DFG of Figure 3(a) and apply the DFG Scheduling Algorithm to the resulting graph. The resulting schedule in Figure 3(b) is more compact and requires only 3 processors (a 40% reduction in resources required for execution) due to the flexibility when scheduling node *E* uncovered by retiming.

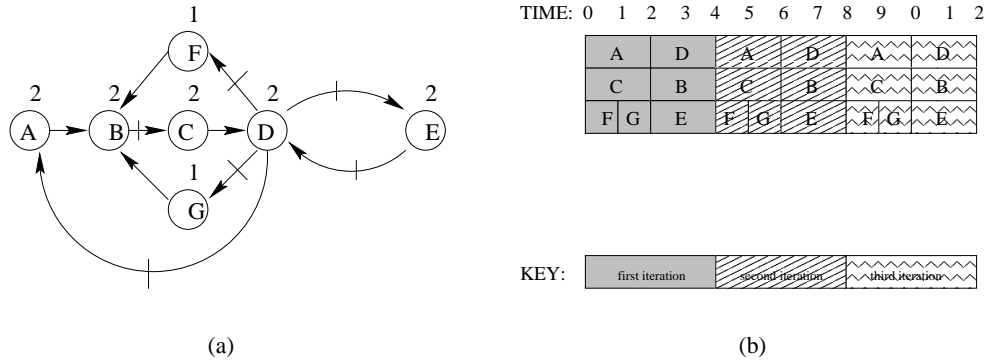


Figure 3: (a) Figure 2(a) retimed; (b) The retimed DFG’s schedule with cycle period 4

The benefits are clear, but reworking our retiming methods so that they may be applied to multirate DFGs 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 [29]. First of all, a retiming may be derived for a single-rate DFG by solving a linear programming problem [18]. The introduction of rates on the edges potentially changes this to a more complicated integer linear programming problem. We later show that this particular ILP system exhibits special properties which permits an efficient solution. 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 paths 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.

The most popular method to date for retiming SDFGs was to avoid the problem entirely by translating the SDFG to its single-rate equivalent and retiming this new graph [12]. The possibility of then translating this new graph back to an equivalent retimed SDFG was mentioned in [27]. Unfortunately, as we will demonstrate, it may be impossible to translate a retimed single-rate graph back to a retimed SDFG. The original idea is flawed as well, in that performing retiming only once replaces an SDFG with a dramatically larger single-rate graph, complicating future stages in the design phase. Clearly it would be preferable to deal with the smaller SDFG as much as possible during this future work once retiming is completed, and while the retiming algorithms we will propose do rely heavily on the much larger single-rate equivalent at intermediate phases, the final product is a still a SDFG.

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 polynomial-time algorithms (in the size of the SDFG's single-rate equivalent) which retime a given SDFG to have a specified clock period. Finally, we will demonstrate the effectiveness of our algorithms by applying them to several examples, in all cases achieving a provably minimal clock period.

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 are our retiming algorithms, followed by detailed examples. 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 [14–16] and later by Zivojnovic *et al* [24, 27, 29]. 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:

1.  $V$  is the vertex set of nodes or *actors*, which transform input data streams into output streams;
2.  $E \subseteq V \times V$  is the edge set, representing channels which carry data streams;
3.  $d : E \rightarrow \mathbf{N} \cup \{0\}$  is a function with  $d(e)$  the number of initial tokens (*delays*) on edge  $e$ ;
4.  $t : V \rightarrow \mathbf{N}$  is a function with  $t(v)$  the execution time of node  $v$ ;
5.  $p : E \rightarrow \mathbf{N}$  is a function with  $p(e)$  the number of data tokens produced at  $e$ 's source node to be carried by  $e$ ;
6.  $c : E \rightarrow \mathbf{N}$  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 4(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.

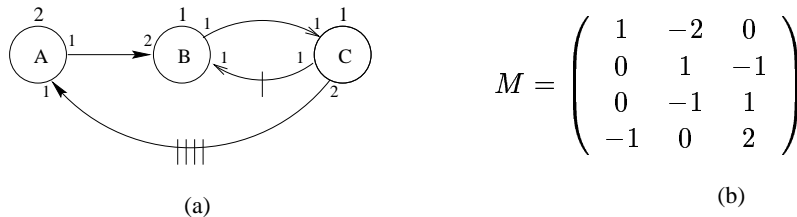


Figure 4: (a) A sample SDFG; (b) Its topology matrix  $M$

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 4(a) is given in Figure 4(b).

In [15] 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. The vector with the smallest norm from this nullspace is called the *repetitions vector (RV)* (or *basic repetition vector* in [2]) for  $G$ . For example, the RV for the SDFG in Figure 4(a) is  $q = [ 2 \ 1 \ 1 ]^T$ . The elements of a RV  $q$  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 5(a). Finally, a SDFG is *consistent* if it has a RV. An example of an inconsistent SDFG from [15] appears as Figure 5(b), with its rank 3 topology matrix in Figure 5(c). It is clear that any attempt to execute this circuit will end in either deadlock or overflowed buffers.

## 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.

Algorithms for creating EHG appear in [2] and [25]. In general, they first create enough copies of each

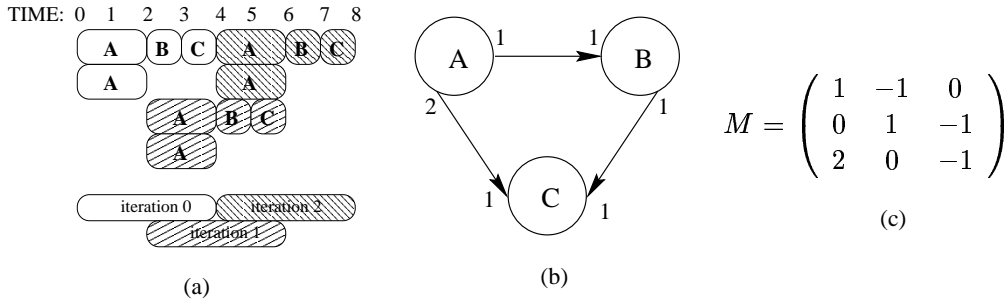


Figure 5: (a) The repeating schedule for Figure 4(a); (b) An inconsistent SDFG; (c) Its topology matrix

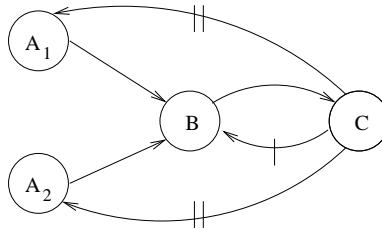


Figure 6: Figure 4(a)'s EHG

node to satisfy the specifications of the RV. They then insert 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. Such algorithms determine which copies of source and sink to map to one another based on how much data has been created and used. As an example, the EHG of Figure 4(a) appears in Figure 6. Note that, for purposes of clarity, we do not combine edges between nodes as is typically done. If multiple tokens are to be sent between nodes in the EHG, each travels along its own edge.

Finally, as derived in [2] and [7], we will say that a SDFG is *live* if its EHG has no zero-delay cycles. Otherwise the graph is *deadlocked*. An example of a consistent deadlocked graph appears as Figure 7(a), with its EHG in Figure 7(b). As we can see, the loop between nodes *A* and *B*<sub>2</sub> 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.

### 2.3 The Delay Count of a Path

In [27], Zivojnovic *et al* briefly discussed computing the *cumulative delay count* of a path in a SDFG, the sum of the delay counts of all copies of the path in the EHG. They omitted many details which would have aided understanding of their ideas, and then added to the confusion with typographical errors. Because of the

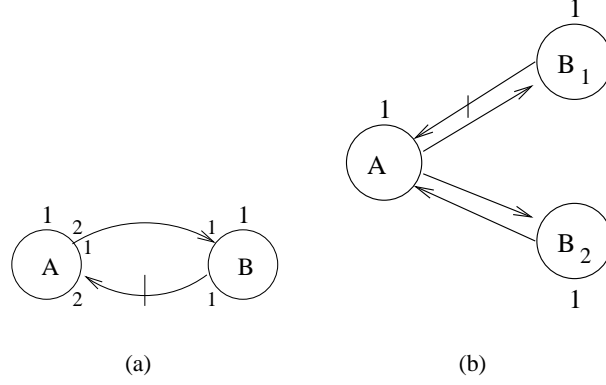


Figure 7: (a) A deadlocked SDFG; (b) Its EHG

necessity of these results for complete understanding of the SDFG model, we will now review, clarify and expand their line of reasoning.

A *path*  $p$  in a SDFG is an ordered sequence of nodes and edges. Given a node  $v$  on path  $p$  with incoming edge  $e_{in}$  and outgoing edge  $e_{out}$ , we will define the *rate gain* of  $v$  in  $p$  as  $g_p(v) = \frac{p(e_{out})}{c(e_{in})}$ . This figure provides some measure of a node's production. If the rate gain is larger than 1, the node is producing more than it is consuming; otherwise we are taking more than we are giving. For example, in the path  $p = (A, B, C)$  in Figure 4(a),  $g_p(B) = \frac{1}{2}$ , indicating that every group of two delays pushed through  $B$  along  $p$  results in one delay being produced as output from  $B$ .

With this concept in hand we can now demonstrate the following:

**Theorem 2.1** *Let  $G$  be a SDFG. Let  $p$  be a path in  $G$  consisting of the nodes  $v_1, v_2, \dots, v_n$  connected by the edges  $e_i : v_i \rightarrow v_{i+1}$  for  $i = 1, 2, \dots, n - 1$ . Then the cumulative delay count of  $p$  is  $D(p) = \sum_{i=1}^{n-1} g_p(v_{i+1}) \dots g_p(v_n) d(e_i)$ .*

*Proof:* The idea is to push all delays down the path and onto the final edge  $e_{n-1}$ . Moving the delays from  $e_1$  through  $v_2$  and onto  $e_2$  produces  $g_p(v_2)d(e_1) + d(e_2)$  delays on  $e_2$ . Moving these further down the path and onto  $e_3$  yields a count of  $g_p(v_3)(g_p(v_2)d(e_1) + d(e_2)) + d(e_3)$  delays on  $e_3$ . Continuing in this fashion, we eventually arrive at  $e_{n-1}$  with

$$g_p(v_2)g_p(v_3)\dots g_p(v_{n-1})d(e_1) + g_p(v_3)\dots g_p(v_{n-1})d(e_2) + \dots + d(e_{n-1})$$

total delays. □

The chief issue with this result is the consistency of the delay count for a cycle. As an example, consider the cycle  $A \rightarrow B \rightarrow C \rightarrow A$  in Figure 4(a). When the nodes are visited in this order, the cycle contains 4 total delays. However, when visited in the order  $C \rightarrow A \rightarrow B \rightarrow C$ , the delay count becomes 2. There is still some gap in our understanding which requires further investigation.

### 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* [17, 18], where delays are redistributed among the edges so that the application’s function remains the same while the execution time decreases. Despite its usefulness when applied to HDFGs, the application of retiming to SDFGs was explored only marginally prior to 1994 [12, 19] before being studied by Zivojnovic *et al* primarily as a way to minimize the delay count of a SDFG [27, 29]. 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

As we’ve said, 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 [3]. This definition is problematic if we attempt to apply it directly to SDFGs, as we can see if we do so to Figure 7(a). 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 4(a) is 4 by this 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 4(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* [21], which is defined to be the maximum time-to-delay ratio of all cycles in the EHG. For example, the EHG in Figure 6 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 4(a) is 2. This can be clearly seen from the schedule in Figure 5(a), where overlapped iterations create higher throughput. (The iteration period of an SDFG can be overestimated using the ideas from [24] 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*  $r : V \rightarrow \mathbf{N} \cup \{0\}$  is a function which specifies a transformation of a graph  $G$ . It labels each vertex with a factor by which production and consumption rates are multiplied when computing the delay counts of the edges in the transformed graph. The effect is to change  $G$  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  $E$  [27, 29]. A retiming is *legal* if  $d_r(e) \geq 0$  for all edges  $e \in E$ . As an example, a legal retiming with  $r(A) = 2$  and  $r(B) = r(C) = 0$  transforms the SDFG of Figure 4(a) into that of Figure 8(a). Examining the EHG in Figure 8(b), we see that we have now achieved



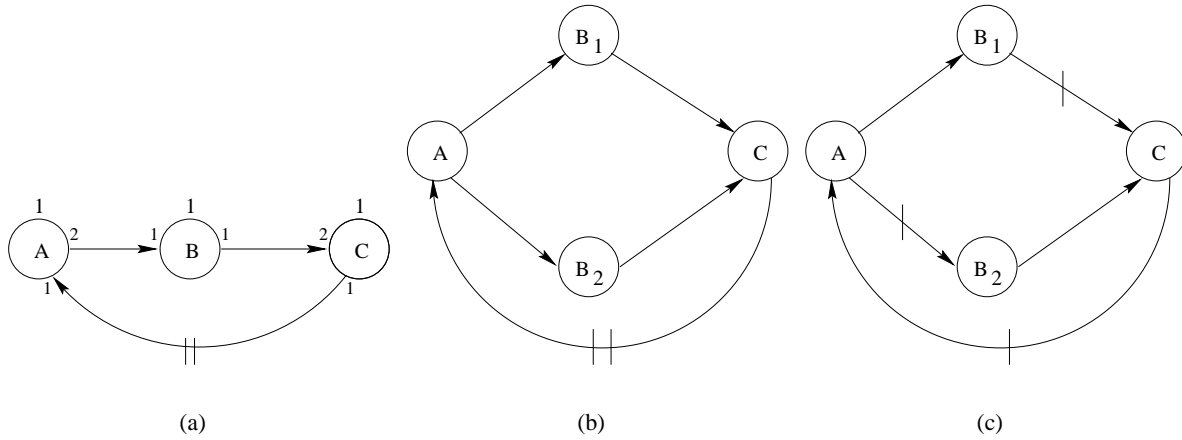


Figure 9: (a) A unit-time SDFG; (b) Its EHG; (c) Its retimed EHG

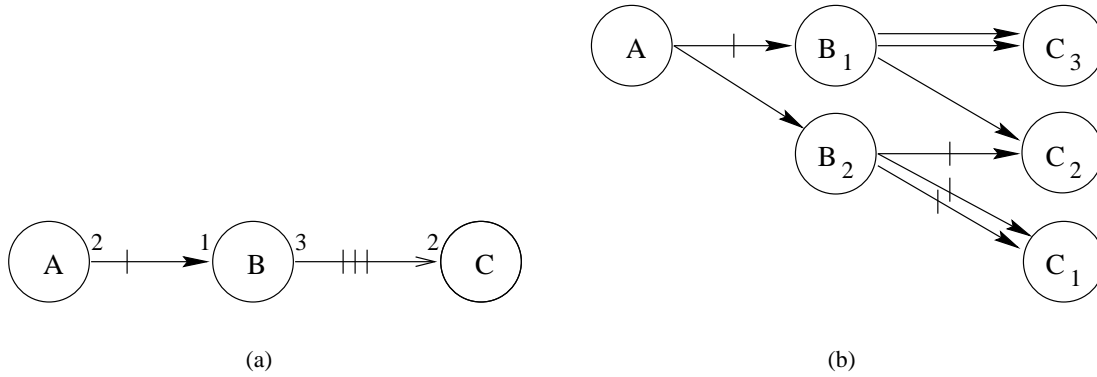


Figure 10: (a) A path in a SDFG; (b) Its homogeneous equivalent

**Theorem 4.1** Let  $G$  be a HDFG and  $c$  a potential clock period.  $cl(G) \leq c$  if and only if 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 10(a) below, with its EHG given in Figure 10(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 must construct intermediate EHG's for study, a very costly process.

In a similar vein, as in [1], 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 10(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 [18], 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 [18]). 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 the definition of the EHG, 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 First Method: Linear Programming

We begin with a mathematical solution based entirely on the following idea.

**Theorem 4.2** *Let  $G = \langle V, E, d, t, p, c \rangle$  be a SDFG with RV  $q$ . 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  $w$ .

*Proof:* For (1), a retiming  $r$  is legal if and only if  $d_r(e) \geq 0$  for all edges  $e$ , which happens if and only if  $d(e) + p(e)r(u) - c(e)r(v) \geq 0$  for all  $e$ , both results derived from definitions. Simple algebraic manipulations now yield the desired result.

The remaining criteria ensure that there is no zero-delay path in the EHG with computation time too large. If  $T(u, v) - \min\{t(u), t(v)\} > c$  then the path from  $u$  to  $v$  contains a subpath which must have positive delay count, so it suffices to consider only the cases we've specified. The conditions for (2) dictate that the path from  $w$  to  $v$  has small enough computation time; we only have a problem when we add an initial edge from  $u$  to  $w$

to the path. Therefore, if we make sure that the retimed delay count for this initial edge is large enough for every copy of the edge in the EHG to have a non-zero delay count, each copy of the path in the EHG will have non-zero delay count and the conditions of Theorem 4.1 will be satisfied. Since there are  $q_u \cdot p(e)$  copies of this edge  $e$  in the EHG, we want  $d_r(e)$  to exceed this figure, leading to the stated criterion. Condition (3) is derived in the same manner, dealing with the final edge in the path rather than the first edge.  $\square$

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 [8]. Furthermore, because we are working with values derived from the EHG, we will avoid the false path problem.

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. Algorithm 1 below is based on the method of [18], constructing a matrix  $M$  and manipulating it via the Floyd-Warshall all-pairs shortest-path algorithm [8] to compute these values for an EHG. Once we have these figures, we compute  $T$  and  $D$  for the original SDFG by setting

$$D(u, v) = \min \{D(u_i, v_j) | u_i \text{ is a copy of } u \text{ and } v_j \text{ a copy of } v \text{ in the EHG}\}$$

and

$$T(u, v) = \max \{T(u_i, v_j) | u_i \text{ is a copy of } u \text{ and } v_j \text{ a copy of } v \text{ in the EHG, } D(u_i, v_j) = D(u, v)\}.$$

We are forced to work with the EHG due to the problems we noted earlier regarding the calculation of a path's delay count.

As an example of the potential wastefulness, let us demonstrate our ideas on the path in Figure 10(a) first with  $c = 2$ . Recall that this example does not need to be retimed given these conditions. The values of  $T$  and  $D$  based on this graph's EHG are given in Table 1. (The blank spaces in the table indicate infinite values representing unconnected nodes.) There are only two edges we are considering, so the first condition of Theorem 4.2 gives us two inequalities:

$$\begin{aligned} r(B) - 2r(A) &\leq 1 \\ 2r(C) - 3r(B) &\leq 3 \end{aligned}$$

There is only one path satisfying the computation time requirements of the second and third conditions, that from  $A$  to  $C$ . Denoting the edge from  $A$  to  $B$  in Figure 10(a) as  $e_1$  and that from  $B$  to  $C$  as  $e_2$ , we see from Figures 10(a) and 10(b) that  $q_A = 1$ ,  $q_C = 3$  and  $p(e_1) = c(e_2) = 2$ , so that the second and third conditions of Theorem 4.2 give us the inequalities

$$\begin{aligned} r(B) - 2r(A) &\leq -1 \\ 2r(C) - 3r(B) &\leq -3 \end{aligned}$$

---

**Algorithm 1** Computing max. comp. time and min. delay count along critical paths

---

**Input:** A HDFG  $G = \langle V, E, d, t \rangle$ **Output:** The values of  $T(u, v)$  and  $D(u, v)$  for all connected nodes  $u$  and  $v$ 

```
for  $i \leftarrow 0$  to  $|V| - 1$  do
  for  $j \leftarrow 0$  to  $|V| - 1$  do
    if  $i = j$  then
       $M[i, j] \leftarrow (0, 0)$ 
    else
       $M[i, j] \leftarrow (\infty, \infty)$ 
    end if
  end for
end for
for all edges  $e : u \rightarrow v$  in  $G$  do
   $M[u, v] \leftarrow (d(e), -t(u))$ 
end for
for  $k \leftarrow 0$  to  $|V| - 1$  do
  for  $i \leftarrow 0$  to  $|V| - 1$  do
    for  $j \leftarrow 0$  to  $|V| - 1$  do
      if  $M[i, j] > M[i, k] + M[k, j]$  then
         $M[i, j] \leftarrow M[i, k] + M[k, j]$ 
      end if
    end for
  end for
end for
for  $i \leftarrow 0$  to  $|V| - 1$  do
  for  $j \leftarrow 0$  to  $|V| - 1$  do
     $D[i, j] \leftarrow M[i, j].x$ 
     $T[i, j] \leftarrow t(j) - M[i, j].y$ 
  end for
end for
```

---

Since the second system supersedes the first, it is sufficient to solve only the second and derive a solution with  $r(A) = r(B) = 1$  and  $r(C) = 0$ . Applying this function to Figure 10(a) yields a retimed graph with  $d_r(e_1) = 2$  and  $d_r(e_2) = 6$ , leading to an EHG where all edges in any copy of this path contain a delay. Considering the observation that the graph was already optimal, we see that our algorithm costs us a great deal in this case.

For another interesting example, consider Figure 11(a) 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.) It's EHG is given in Figure 11(b), with the  $T$  and  $D$  values in Table 2. Let us begin by attempting to satisfy the second and third conditions of Theorem 4.2. Since all nodes have computation time 4 in this example, we need only consider paths with total computation time 8 whose  $T$  and  $D$  values appear in boldfaced type in Table 2. 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  $A$ :

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

	<i>T</i>						<i>D</i>					
	<i>A</i>	<i>B</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>C</i> <sub>1</sub>	<i>C</i> <sub>2</sub>	<i>C</i> <sub>3</sub>	<i>A</i>	<i>B</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>C</i> <sub>1</sub>	<i>C</i> <sub>2</sub>	<i>C</i> <sub>3</sub>
<i>A</i>	1	2	2	3	3	3	0	1	0	1	1	1
<i>B</i> <sub>1</sub>		1		2	2			0		0	0	
<i>B</i> <sub>2</sub>			1		2	2			0		1	1
<i>C</i> <sub>1</sub>				1						0		
<i>C</i> <sub>2</sub>					1						0	
<i>C</i> <sub>3</sub>						1						0

Table 1: *T* and *D* values for the EHG of Figure 10(a)

	<i>T</i>										<i>D</i>									
	<i>A</i> <sub>1</sub>	<i>A</i> <sub>2</sub>	<i>B</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>B</i> <sub>3</sub>	<i>C</i> <sub>1</sub>	<i>C</i> <sub>2</sub>	<i>C</i> <sub>3</sub>	<i>C</i> <sub>4</sub>	<i>D</i> <sub>1</sub>	<i>A</i> <sub>1</sub>	<i>A</i> <sub>2</sub>	<i>B</i> <sub>1</sub>	<i>B</i> <sub>2</sub>	<i>B</i> <sub>3</sub>	<i>C</i> <sub>1</sub>	<i>C</i> <sub>2</sub>	<i>C</i> <sub>3</sub>	<i>C</i> <sub>4</sub>	<i>D</i> <sub>1</sub>
<i>A</i> <sub>1</sub>	4	16	<b>8</b>	<b>8</b>	20	<b>8</b>	<b>8</b>	20	20	12	0	7	<b>0</b>	<b>0</b>	7	<b>0</b>	<b>0</b>	7	7	0
<i>A</i> <sub>2</sub>	16	4	20	<b>8</b>	<b>8</b>	20	20	<b>8</b>	<b>8</b>	12	7	0	7	<b>0</b>	<b>0</b>	7	7	<b>0</b>	<b>0</b>	0
<i>B</i> <sub>1</sub>	12	12	4	16	16	16	16	16	16	<b>8</b>	7	7	0	7	7	7	7	7	7	<b>0</b>
<i>B</i> <sub>2</sub>	12	12	16	4	16	16	16	16	16	<b>8</b>	7	7	7	0	7	7	7	7	7	<b>0</b>
<i>B</i> <sub>3</sub>	12	12	16	16	4	16	16	16	16	<b>8</b>	7	7	7	7	0	7	7	7	7	<b>0</b>
<i>C</i> <sub>1</sub>	12	12	16	16	16	4	16	16	16	<b>8</b>	7	7	7	7	7	0	7	7	7	<b>0</b>
<i>C</i> <sub>2</sub>	12	12	16	16	16	16	4	16	16	<b>8</b>	7	7	7	7	7	7	0	7	7	<b>0</b>
<i>C</i> <sub>3</sub>	12	12	16	16	16	16	16	4	16	<b>8</b>	7	7	7	7	7	7	7	0	7	<b>0</b>
<i>C</i> <sub>4</sub>	12	12	16	16	16	16	16	16	4	<b>8</b>	7	7	7	7	7	7	7	7	0	<b>0</b>
<i>D</i> <sub>1</sub>	<b>8</b>	<b>8</b>	12	12	12	12	12	12	12	4	<b>7</b>	<b>7</b>	7	7	7	7	7	7	7	0

Table 2: *T* and *D* values for the EHG of Figure 11(b)

$$r(C) - 2r(A) \leq -4$$

$$3r(D) - r(B) \leq -3$$

$$4r(D) - r(C) \leq -4$$

$$r(A) - 2r(D) \leq 12$$

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

$$4r(B) - 6r(A) \leq -12$$

$$3r(C) - 6r(A) \leq -12$$

$$12r(D) - 4r(B) \leq -12$$

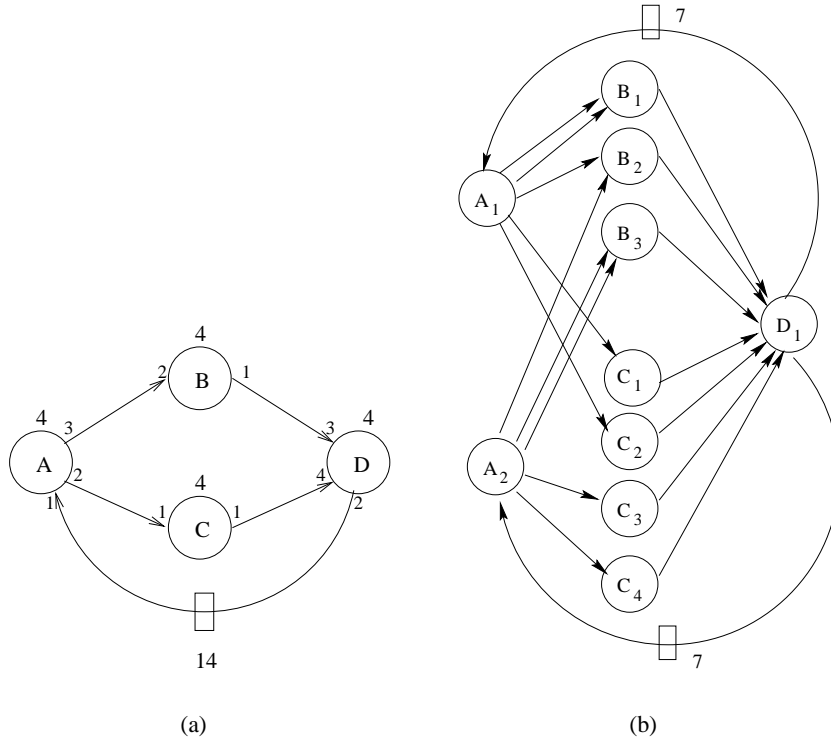


Figure 11: (a) An example SDFG; (b) Its EHG

$$12r(D) - 3r(C) \leq -12$$

$$6r(A) - 12r(D) \leq 72$$

As in [8], we can now model our set of inequalities by the constraint graph of Figure 12(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 before dividing 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 11(a) yields the retimed graph of Figure 12(b) whose EHG appears in Figure 12(c). An examination of this EHG reveals that we have indeed found a retiming which achieves our desired clock period. We also see that, due to the different rates of production and consumption by each of the nodes, the delay counts in the cycles no longer appear to match.

### 4.3 Second Method: Relaxation

Alternately we can more efficiently seek our retiming via *relaxation* on the edges of our graph. We do this by topologically sorting our vertices (so that  $u$  precedes  $v$  if there is a zero-delay edge  $(u, v)$  in the EHG [8]) and then sweeping along the sorted list. When we get to a point where the current path is too long, we insert



---

**Algorithm 2** 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**  $\tau$ 

---

assign it a longest path length  $\Upsilon(u)$  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 [18]), 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 11(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.

We now demonstrate our method by executing it on the SDFG of Figure 11(a) with  $c = 4$ . Sorting the vertices of Figure 11(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 13(a), with its EHG appearing in Figure 13(b).

Looping back around, we see from Figure 13(b) 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  $r(A) = 2$ ,

---

**Algorithm 3** 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 2 */
  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 2 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
```

---

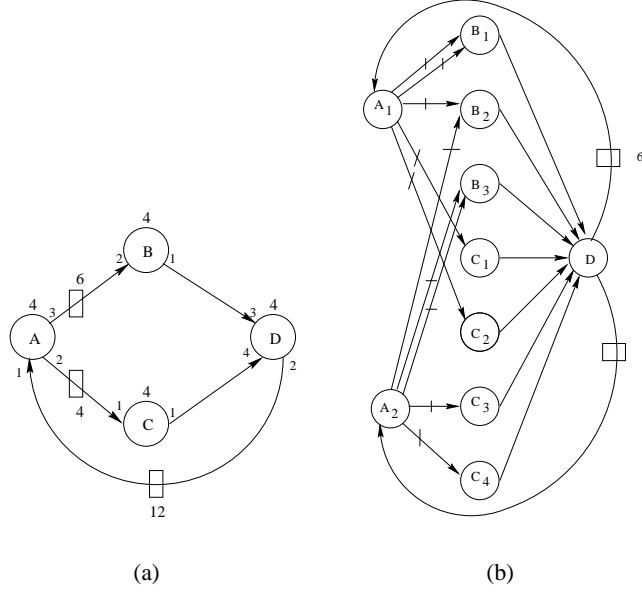


Figure 13: (a) The retimed SDFG after one pass of algorithm; (b) Its EHG

$r(B) = q_B = 3$ ,  $r(C) = q_C = 4$  and  $r(D) = 0$ . The new retimed graph is given below as Figure 14(a), with its EHG in Figure 14(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  $r(D) = 0$ . The application of this retiming to the original SDFG results in the graph of Figure 12(b) and we have found our answer.

However we have a final pass of the algorithm to perform. We construct and study the EHG of Figure 12(c), find that the maximum zero-delay path is an individual node with computation time 4, conclude that we have found our retiming and return it as our answer before proceeding to the inner nested loops.

#### 4.4 Discussion

Our first method is a straight-forward problem in linear programming but constitutes a very expensive solution. If  $G = \langle V, E, d, t, p, c \rangle$  is our SDFG with EHG  $H = \langle V', E', d', t' \rangle$ , then it takes  $O(|V'|^3)$  time to derive the values of  $D$  and  $T$  via Algorithm 1, plus another  $O(|V|^3)$  time to execute the Bellman-Ford algorithm to find the shortest path lengths in our constraint graph. There is no guarantee that this method will yield an answer, and even if it finds a solution, it does so at a great price.

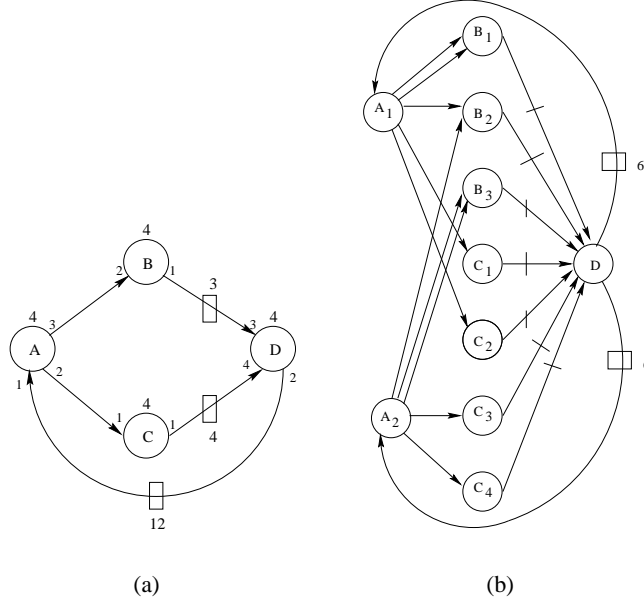


Figure 14: (a) The retimed SDFG after two passes of algorithm; (b) Its EHG

On the other hand, our second procedure is more efficient than our first method and more intuitive. Since the construction of an EHG and Algorithm 2 each require  $O(|E'|)$  time, Algorithm 3 executes in only  $O(|V||V'| + |V||E'|)$  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 [18] 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 Examples

In this section, we illustrate our methods further by applying them to various SDFGs found in the literature.

### 5.1 First Example

Consider the SDFG in Figure 15(a), a variation on the example from [19] with a RV of  $q = [2 \ 1 \ 2]^T$ . In our example, nodes  $A$  and  $B$  take 1 time unit to execute and  $C$  takes 2; thus we will attempt to retime it to have an optimal clock period of 2. Our relaxation algorithm requires two passes to complete. At the outset, we compute the longest path lengths and find that  $\Upsilon(A) = 1$ ,  $\Upsilon(B) = 2$  and  $\Upsilon(C) = 4$ . Hence nodes  $A$  and  $B$  require retiming. Node  $A$  is source node for only one edge with production rate 1; thus  $r(A) = q_A = 2$ . On the other hand, two edges emanate from  $B$ :  $(B, A)$  with delay count 4 and production rate 2, and  $(B, C)$

contains no delays with production rate 4. Taking the maximum yields a value of  $r(B) = q_B = 1$ . Applying this retiming produces the graph of Figure 15(b). Looping back around, we find that the path lengths in the EHG are acceptably small and return our current function as the desired retiming.

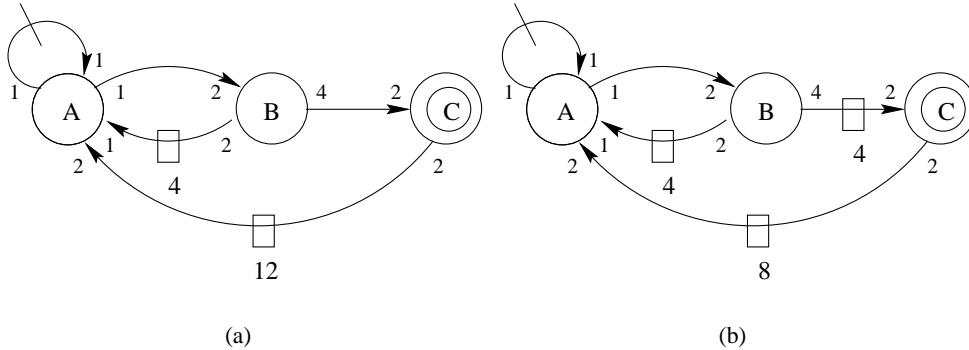


Figure 15: (a) A sample SDFG; (b) Figure 15(a) retimed

## 5.2 Second Example

Our next example is a slowed version of the example from [29], given in Figure 16(a). The RV is  $q = [1 \ 8 \ 8 \ 4]^T$ . We will attempt to achieve a clock period of 3, which equals the execution time of node A and hence is minimal. If we apply our relaxation algorithm, we complete execution in two passes. The first time through computes longest zero-delay path lengths of  $\Upsilon(A) = 6$ ,  $\Upsilon(B) = \Upsilon(C) = 3$  and  $\Upsilon(D) = 2$ , leading to the retiming of all nodes except A. Since  $p(e) = 1$  and  $d(e) = 0$  for all edges  $e$  which emanate from either B or C, we have  $r(v) = q_v = 8$  for  $v = B$  or  $C$ . On the other hand, two edges have D as source node, both with production rate 2. Since  $q_D = 4$ ,  $r(D) = \frac{8-0}{2} = 4$ . This is a legal retiming, and when we begin the next pass of our loop, we find that it is adequate for retiming Figure 16(a) to have clock period 3, thus terminating execution of our algorithm with the retimed graph in Figure 16(b) which executes within our desired time interval.

## 5.3 A Simplified Spectrum Analyzer

Finally, let us apply our algorithms to a variation of the simplified spectrum analyzer from [27] which appears in Figure 17(a), with node descriptions in Figure 17(b). This graph has a RV of  $q = [16 \ 1 \ 1 \ 1 \ 4 \ 1]^T$ , so in the interests of space we will not display the EHG at each step. Instead, we shall describe the pertinent information. It can be shown that the lower bound on the clock period for this SDFG is 3, and so we will attempt to retime it to be optimal.

When we apply our relaxation algorithm, we first see the zero-delay paths  $(F, B, C)$  and  $(A, B, C)$  which give us  $\Upsilon(A) = \Upsilon(D) = 1$ ,  $\Upsilon(B) = 3$ ,  $\Upsilon(C) = 5$  and  $\Upsilon(E) = \Upsilon(F) = 2$ . Thus all nodes except C need

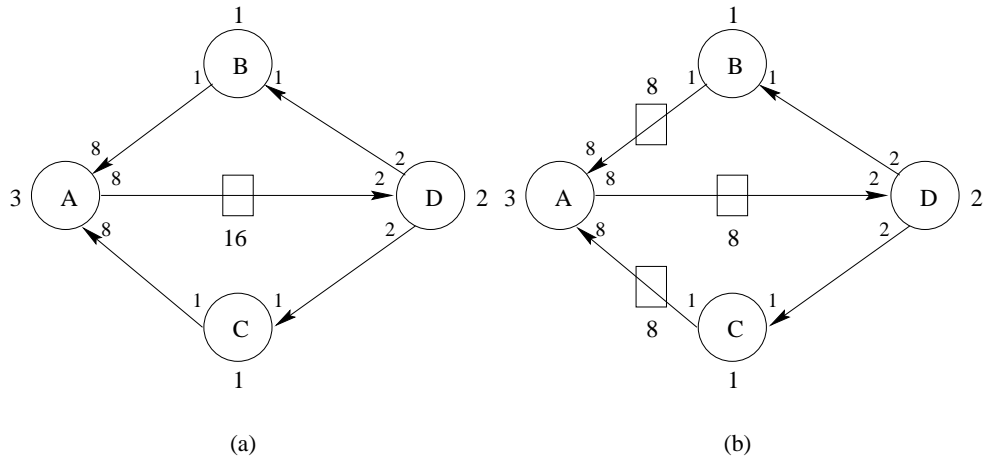
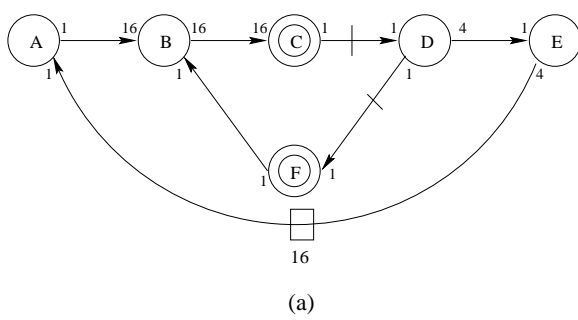


Figure 16: (a) Another sample SDFG; (b) Figure 16(a) retimed



Node	Description	Time
<i>A</i>	Adaptive Low-Pass	1
<i>B</i>	FFT Zoom	1
<i>C</i>	Peak Detector	2
<i>D</i>	Interpolator	1
<i>E</i>	Decision	1
<i>F</i>	Zoom Control	2

(b)

Figure 17: A simplified spectrum analyzer

retiming, and our formula gives an initial retiming of  $r(A) = p((A, B)) = 16$ ,  $r(B) = \frac{p((B,C))}{p((B,C))} = 1$ ,  $r(C) = 0$ ,  $r(D) = \max \left\{ \frac{p((D,E))}{p((D,E))}, p((D, F)) - 1 \right\} = 1$ ,  $r(E) = \frac{q_E \cdot p((E,A)) - 16}{p((E,A))} = 0$  and  $r(F) = p((F, B)) = 1$ . (The value for  $r(E)$  reveals an interesting pattern; if there are already sufficient delays on an edge, the value of the retiming for the edge's source node will not change.) Applying this retiming to the graph in Figure 17(a) yields the graph of Figure 18. In the next pass of the algorithm, we first check the clock period of this retimed graph, find that we have achieved an optimal retimed graph, and return the current value of the retiming as our final answer.

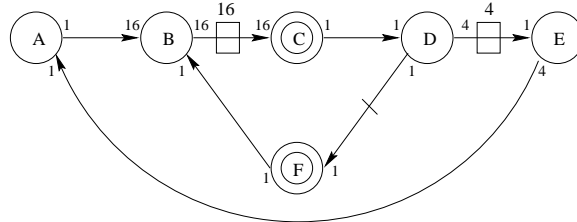


Figure 18: The retimed analyzer

## 6 Conclusion

In this paper, we have established a notation for expressing and studying retimings of synchronous data-flow graphs. We have presented the difficulties involved with retiming SDFGs, and then constructed polynomial-time algorithms (in the size of the SDFG's homogeneous equivalent) for retiming a synchronous graph so that it achieves a sufficiently small clock period. Finally, we have demonstrated the effectiveness of our algorithms on several examples. In all cases we have studied, we have been able to achieve minimal execution times, indicating the strength of our second algorithm.

Regardless of how good our algorithms may be, they are still not proven to represent both necessary and sufficient conditions for retiming. These proofs, or the construction of an alternate method which is necessary and sufficient, remain interesting open problems. A better grasp of the results regarding delay counts in [27] 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 [2] or dynamic DFGs.

## References

- [1] S.S. Bhattacharya, P.K. Murthy, and E.A. Lee. *Software Synthesis from Dataflow Graphs*. Kluwer Academic Publishers, 1996.

- [2] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44:397–408, 1996.
- [3] L.-F. Chao. *Scheduling and Behavioral Transformations for Parallel Systems*. PhD thesis, Dept. of Computer Science, Princeton University, 1993.
- [4] L.-F. Chao and E. H.-M. Sha. Retiming and unfolding data-flow graphs. In *Proceedings of the International Conference on Parallel Processing*, pages II 33–40, 1992.
- [5] L.-F. Chao and E. H.-M. Sha. Static scheduling for synthesis of DSP algorithms on various models. *Journal of VLSI Signal Processing*, 10:207–223, 1995.
- [6] L.-F. Chao and E. H.-M. Sha. Scheduling data-flow graphs via retiming and unfolding. *IEEE Transactions on Parallel and Distributed Systems*, 8:1259–1267, 1997.
- [7] F. Commoner, A.W. Holt, S. Even, and A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5:511–523, 1971.
- [8] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, Inc., 1991.
- [9] G. Gao, R. Govindarajan, and P. Panangaden. Well-behaved dataflow programs for DSP computation. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 5, pages 561–564, 1992.
- [10] P.D. Hoang and J.M. Rabaey. Scheduling of DSP programs onto multiprocessors for maximum throughput. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 41:2225–2235, 1993.
- [11] R. Lauwereins, M. Engels, J.A. Peperstraete, E. Steegmans, and J. Van Ginderdeuren. GRAPE: A CASE tool for digital signal parallel processing. *IEEE ASSP Magazine*, 7:32–43, 1990.
- [12] 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.
- [13] E.A. Lee, W.H. Ho, E. Goei, J. Bier, and S.S. Bhattacharya. Gabriel: A design environment for DSP. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 37:1751–1762, 1989.
- [14] E.A. Lee and D.G. Messerschmitt. Pipeline interleaved programmable DSP's: Synchronous data flow programming. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-35:1334–1345, 1987.
- [15] E.A. Lee and D.G. Messerschmitt. Static scheduling of synchronous data-flow programs for digital signal processing. *IEEE Transactions on Computers*, 36:24–35, 1987.

- [16] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75:1235–1245, 1987.
- [17] C.E. Leiserson and J.B. Saxe. Optimizing synchronous systems. *Journal of VLSI and Computer Systems*, 1(1):41–67, 1983.
- [18] C.E. Leiserson and J.B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
- [19] K.K. Parhi. Algorithm transformation techniques for concurrent processors. *Proceedings of the IEEE*, 77:1879–1895, 1989.
- [20] J.L. Pino, S. Ha, E.A. Lee, and J.T. Buck. Software synthesis for DSP using Ptolemy. *Journal of VLSI Signal Processing*, 9:7–21, 1995.
- [21] M. Renfors and Y. Neuvo. The maximum sampling rate of digital filters under hardware speed. *Transactions on Circuits and Sampling*, CAS-28:196–202, 1981.
- [22] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr. High-level software synthesis for the design of communication systems. *IEEE Journal on Selected Areas in Communications, Special Issue on Computer-Aided Modeling, Analysis and Design of Communication Links*, 11:348–358, 1993.
- [23] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr. Optimum vectorization of scalable synchronous dataflow graphs. In *Proceedings of the International Conference on Application-Specific Array Processors*, pages 285–296, 1993.
- [24] R. Schoenen, V. Zivojnovic, and H. Meyr. An upper bound of the throughput of multirate multiprocessor schedules. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 655–658, 1997.
- [25] G.C. Sih. *Multiprocessor Scheduling to Account For Interprocessor Communication*. PhD thesis, Dept. of EECS, Univ. of California at Berkeley, 1991.
- [26] M. Veiga, J. Parera, and J. Santos. Programming DSP systems on multiprocessor architectures. In *Proceedings of the International Conference on Acoustics, Speech and Signal Processing*, volume 2, pages 965–968, 1990.
- [27] V. Zivojnovic, S. Ritz, and H. Meyr. Optimizing DSP programs under the multirate retiming transformation. In *Proceedings of the 7th European Signal Processing Conference*, volume 3, pages 1597–1600, 1994.
- [28] V. Zivojnovic, S. Ritz, and H. Meyr. Retiming of DSP programs for optimum vectorization. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 1, pages 465–468, 1994.

- [29] V. Zivojnovic and R. Schoenen. On retiming of multirate DSP algorithms. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 6, pages 3310–3313, 1996.