Retiming Synchronous Data-Flow Graphs to Reduce Execution Time

Timothy W. O'Neil, *Student Member, IEEE*, and Edwin H.-M. Sha, *Member, IEEE*

***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 to expres and study SDFGs. We review the problems faced when attempting to retime an SDFG in order to minimize clock period and then present algorithms for doing this. Finally, we demonstrate the effectiveness of our methods on several examples.**

I. 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 an SDFG represent functional elements, whereas 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], [21], [27] with its use leading to numerous important results regarding the scheduling [9], hierarchization [24], vectorization [23], 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 [29] or minimize the total delay count of an SDFG [28]. However, the problem of using retiming to minimize the clock period of a multirate DFG has remained unexplored. In this paper, we will discuss this problem and propose a method for accomplishing this task.

The benefits of retiming single-rate data-flow graphs are widely reported in the literature. (See several cited in [19].) However, reworking our retiming methods so that they may be applied to multirate DFGs is not easy. The difference between the single-rate and multirate 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 [30]. 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 that permit 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 is not clear what we want to avoid in the multirate 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 [28]. 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 an SDFG.

In this paper, we will develop the basic definitions and results necessary for specifying and manipulating an SDFG and

Manuscript received August 8, 2000; revised June 14, 2001. This work was supported in part by the National Science Foundation under Grants MIP95-01006 and MIP97-04276 and by the A. J. Schmitt Foundation. The associate editor coordinating the review of this paper and approving it for publication was Prof. Chaitali Chakrabarti.

T. W. O'Neil is with the Department of Computer Science and Engineering, The University of Notre Dame, Notre Dame, IN 46556 USA (e-mail: toneil@cse.nd.edu).

E. H.-M. Sha is with the Department of Computer Science, Erik Jonsson School of Engineering and Computer Science, The University of Texas at Dallas, Richardson, TX 75083-0266, USA (e-mail: edsha@utdallas.edu).

Publisher Item Identifier S 1053-587X(01)08247-2.

its single-rate equivalent. We will review retiming and point out the problems that arise when it is applied to SDFGs. We will propose polynomial-time algorithms (in the size of the SDFG's single-rate equivalent) that 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.

II. 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.* [25], [28], [30]. In this section, we review their definitions and ideas in order to formalize these concepts.

A. Basic Definitions

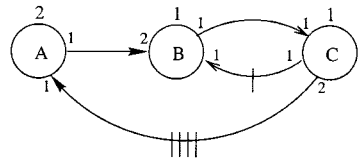
A *synchronous data-flow graph* (*SDFG*) (sometimes called a *multirate* or *regular* data-flow graph) is a finite, directed, weighted graph $G = \langle V, E, d, t, p, c\rangle$ where we have the following.

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 where $d(e)$ is 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 $p(e)$, which is 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 $c(e)$, which is the number of data tokens consumed from e by e's sink node.

(In this definition, $\mathbf{N}$ is the set of natural numbers $\{1, 2, 3, \ldots\}$.) 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 Fig. 1(a). The numbers above the nodes represent the execution times for the individual tasks, whereas the smaller numbers at either end of an edge denote tokens produced or consumed. As an example, $t(A) = 2$, whereas $t(B) = t(C) = 1$ 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, whereas node B consumes two tokens from this edge each time it fires.

It is sometimes useful to characterize an SDFG by its *topology matrix*, which is an $|E| \times |V|$ matrix similar to an incidence matrix. Each row corresponds to one edge in the graph, whereas each column corresponds to a node. A positive



(a)

$$\begin{pmatrix} 1 & -2 & 0 \\ 0 & 1 & -1 \\ 0 & -1 & 1 \\ -1 & 0 & 2 \end{pmatrix}$$

(b)

Fig. 1. (a) Sample SDFG. (b) Its topology matrix M.

(i, j)th entry in the topology matrix indicates the number of tokens produced by the jth node on the ith edge, whereas 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 Fig. 1(a) is given in Fig. 1(b).

In [15], it was demonstrated that a repeating sequential schedule can be constructed for an 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 Fig. 1(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 Fig. 2(a). Finally, an SDFG is *consistent* if it has an RV. An example of an inconsistent SDFG from [15] appears as Fig. 2(b), with its rank-3 topology matrix in Fig. 2(c). It is clear that any attempt to execute this circuit will end in either deadlock or overflowed buffers.

B. 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 EHGs appear in [2] and [26]. In general, they first create enough copies of each node to satisfy the specifications of the RV. They then insert edges. If nodes in an 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 Fig. 1(a) appears in Fig. 3(a). 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 an SDFG is *live* if its EHG has no zero-delay cycles. Otherwise, the graph is *deadlocked*. An example of a consistent deadlocked graph

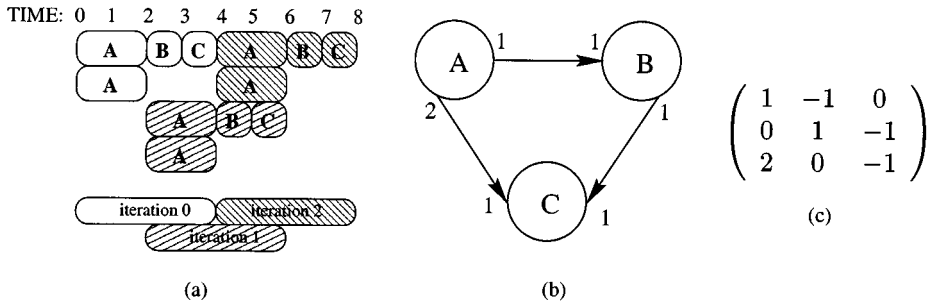


Fig. 2. (a) Repeating schedule for Fig. 1(a). (b) Inconsistent SDFG. (c) Its topology matrix M.

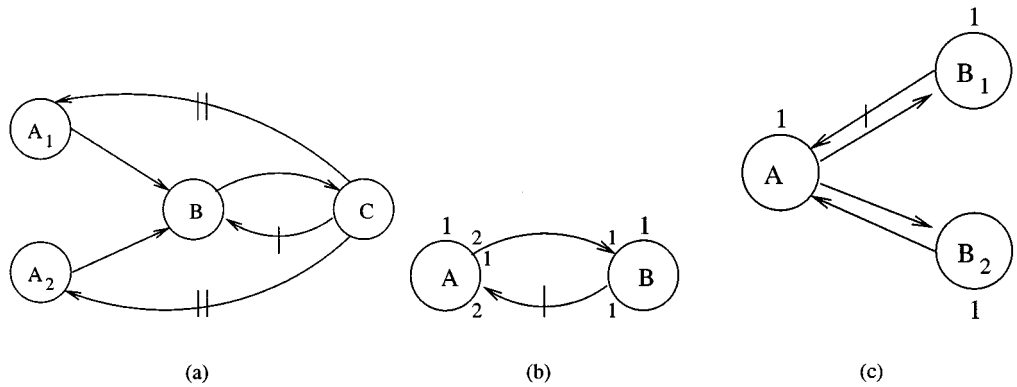


Fig. 3. (a) EHG of Fig. 1(a). (b) Deadlocked SDFG. (c) Its EHG.

appears as Fig. 3(b), with its EHG in Fig. 3(c). As we can see, the loop between nodes A and B_2 contain no delays, and therefore, it is impossible to schedule them since each must precede the other. It should be clear that an SDFG must be both live and consistent in order for it to have a repeating static schedule.

C. Delay Count of a Path

In [28], Zivojnovic *et al.* briefly discussed computing the *cumulative delay count* of a path in an SDFG, which is the sum of the delay counts of all copies of the path in the EHG. They omitted many details that would have aided understanding of their ideas and then added to the confusion with typographical errors. Because of the 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 an 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 $g_p(v) = (p(e_{\text{out}}))/(c(e_{\text{in}}))$. 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 Fig. 1(a), $g_p(B) = 1/2$, 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 II.1: Let G be an SDFG. Let p be a path in G consisting of the nodes $v_1, v_2, \ldots, v_n$ connected by the edges $e_i : v_i \rightarrow v_{i+1}$ for $i = 1, 2, \ldots, n-1$. Then, the cumulative delay count of p is $D(p) = \sum_{i=1}^{n-1} g_p(v_{i+1}) \ldots g_p(v_{n-1}) d(e_i)$.

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

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

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 Fig. 1(a). When the nodes are visited in this order, the cycle contains four 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 that requires further investigation.

III. 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], [20] before being studied by Zivojnovic *et al.*primarily as a way to minimize the delay count

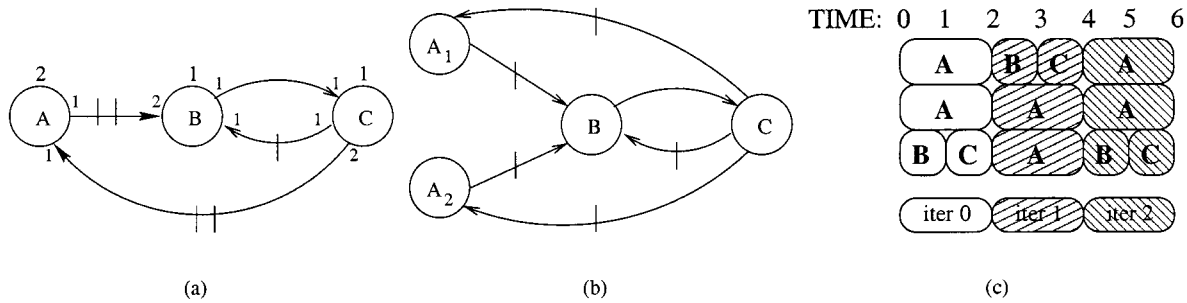


Fig. 4. (a) Fig. 1(a) retimed. (b) Its EHG. (c) Its repeating schedule.

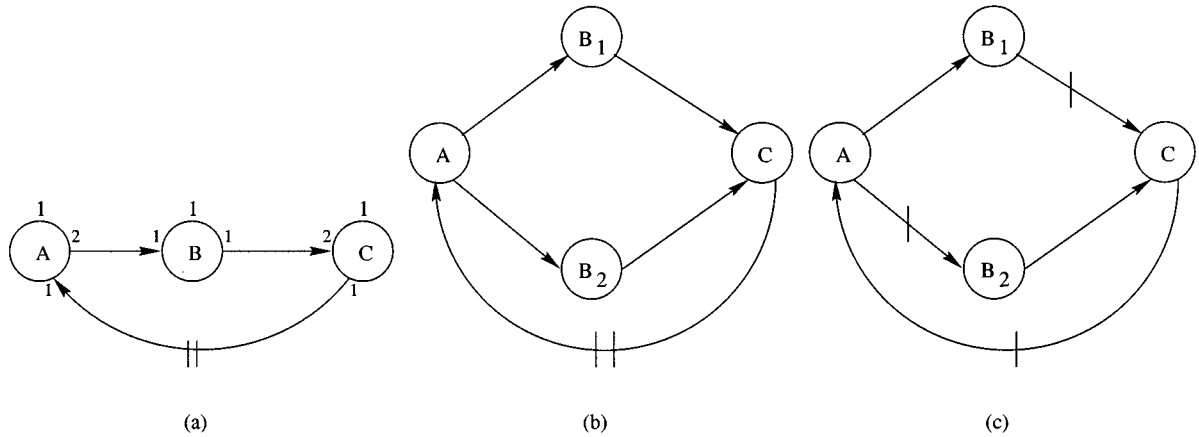


Fig. 5. (a) Unit-time SDFG. (b) Its EHG. (c) Its retimed EHG.

of an SDFG [28], [30]. In this section, we intend to review the basics of retiming, explore some of the pitfalls that arise when studying retiming of SDFGs, demonstrate the effectiveness of retiming, and propose two algorithms for retiming SDFGs.

A. Basic Definitions

As we have said, a *path* in either an SDFG or an HDFG is any sequence of nodes and edges. The *clock period* $\mathrm{cl}(G)$ of an 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 Fig. 3(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 an SDFG to be equal to the clock period of its EHG. As an example, the clock period of the SDFG in Fig. 1(a) is 4 by this definition.

Similar problems arise when we attempt to minimize the clock period. We will say that an *iteration* of an 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 Fig. 1(a), the iteration period is also 4.) If an SDFG contains a loop, then the iteration period is bounded from below by the *iteration bound* [22], which is defined to be the maximum time-to-delay ratio of all cycles in the EHG. For example, the EHG in Fig. 3(a) contains three loops: (A_1, B, C) and (A_2, B, C), each with a 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 Fig. 1(a) is 2. This can be clearly seen from the schedule in Fig. 2(a), where overlapped iterations create higher throughput. (The iteration period of an SDFG can be overestimated using the ideas from [25] 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 that 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 [28], [30]. 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 Fig. 1(a) into that of Fig. 4(a). Examining the EHG in Fig. 4(b), we see that we have now achieved an optimal clock period of 2, which translates into the more compact schedule of Fig. 4(c). Finally, as in [28], the retimed delay count of a path p is $D_r(p) = D(p) + g_p(v_2) \ldots g_p(v_{n-1})p(e_1)r(v_1) - c(e_{n-1})r(v_n)$.

B. Problems Retiming EHGs

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 proposed by Zivojnovic [28]. Unfortunately, the initial translation from SDFG to EHG is too complex to permit this. As an example, consider the unit-time SDFG given in Fig. 5(a), with its EHG appearing in Fig. 5(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 Fig. 5(c) with clock

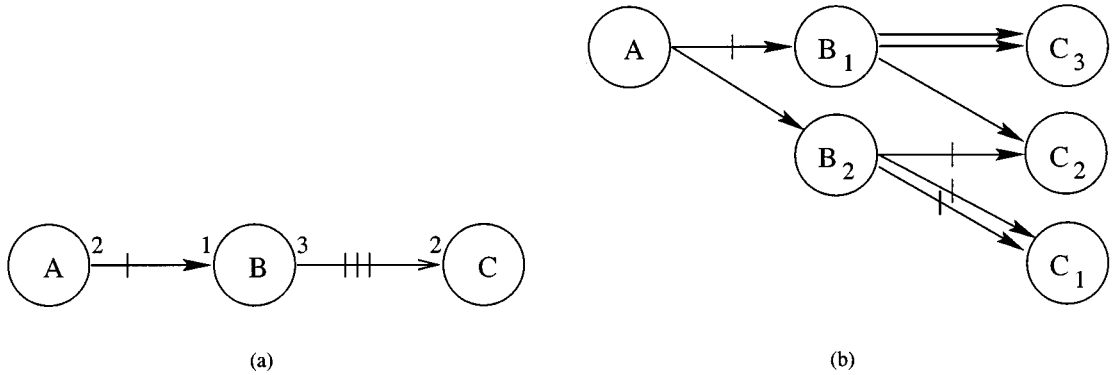


Fig. 6. (a) Path in an SDFG. (b) Its homogeneous equivalent.

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), which is 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 nonzero 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.

IV. Retiming an SDFG

Since we cannot retime an SDFG by working with its EHG, we must develop methods for retiming the SDFG directly. In this section, we refine the methods of [18] to deal with this situation.

A. 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 [18]:

Theorem IV.1: Let G be an HDFG and c a potential clock period. $\mathrm{cl}(G) \leq c$ 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 an SDFG do not necessarily translate into a zero-delay path in the EHG. As an example, consider the unit-time SDFG in Fig. 6(a), with its EHG given in Fig. 6(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, which is 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 Fig. 6(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 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, which was a stipulation that 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 therefore, 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 $e : u \to v$ be an edge in an 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 therefore, 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.

B. First Method—Linear Programming

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

Theorem IV.2: Let $G = \langle V, E, d, t, p, c\rangle$ be an 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 define $T(u, v)$ 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 $\mathrm{cl}(G_r) \leq c$ if we have the following.

1) $c(e)r(v) - p(e)r(u) \leq d(e)$ for all edges $e : u \to v$ 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 $e : u \to w$ is the edge in π 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 $e : w \to v$ is the edge in π 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, where both results are 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 that must have positive delay count; therefore, it suffices to consider only the cases we have 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 nonzero delay count, each copy of the path in the EHG will have nonzero delay count, and the conditions of Theorem IV.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. ■

This result leads to an algorithm that 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 that 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, which are 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

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

and

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

where u_i is a copy of u, and v_j is a copy of v in the EHG. We are forced to work with the EHG due to the problems we noted 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 Fig. 6(a) first with $c = 2$. Recall that this example does not need to be retimed given these conditions. There are only two edges we are considering; therefore, the first condition of Theorem IV.2 gives us two inequalities:

$$r(B) - 2r(A) \leq 1$$
$$2r(C) - 3r(B) \leq 3.$$

There is only one path satisfying the computation time requirements of the second and third conditions—that from A to C. (Indeed, one can see that $T(A, C_i) = 3$ and $D(A, C_i) = 1$ for $i = 1, 2, 3$; a complete table of T and D values for this example may be found in [19].) Denoting the edge from A to B in Fig. 6(a) as e_1 and that from B to C as e_2, we see from Figs. 6(a) and (b) that $q_A = 1$, $q_C = 3$, and $p(e_1) = c(e_2) = 2$; therefore, the second and third conditions of Theorem IV.2 give us the inequalities

$$r(B) - 2r(A) \leq -1$$
$$2r(C) - 3r(B) \leq -3.$$

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 Fig. 6(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.

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

Input: An 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 ← 0 to |V| − 1 do
  for j ← 0 to |V| − 1 do
    if i = j then
      M[i, j] ← (0, 0)
    else
      M[i, j] ← (∞, ∞)
    end if
  end for
end for
for all edges e : u → v in G do
  M[u, v] ← (d(e), −t(u))
end for
for k ← 0 to |V| − 1 do
  for i ← 0 to |V| − 1 do
    for j ← 0 to |V| − 1 do
      if M[i, j] > M[i, k] + M[k, j] then
        M[i, j] ← M[i, k] + M[k, j]
      end if
    end for
  end for
end for
for i ← 0 to |V| − 1 do
  for j ← 0 to |V| − 1 do
    D[i, j] ← M[i, j].x
    T[i, j] ← t(j) − M[i, j].y
  end for
end for
```

For another interesting example, consider Fig. 7(a), 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.) Its EHG is given in Fig. 7(b). The T and D values derived from this graph are given in [19]; suffice it to say that for each pair of vertices u and v connected by an edge in the SDFG, $T(u,v) = 8$ and $D(u,v) = 0$, except that $D(D, A) = 7$. Let us begin by attempting to satisfy the second and third conditions of Theorem IV.2. Since all nodes have computation time 4 in this example, we need only consider paths with total computation time 8. We

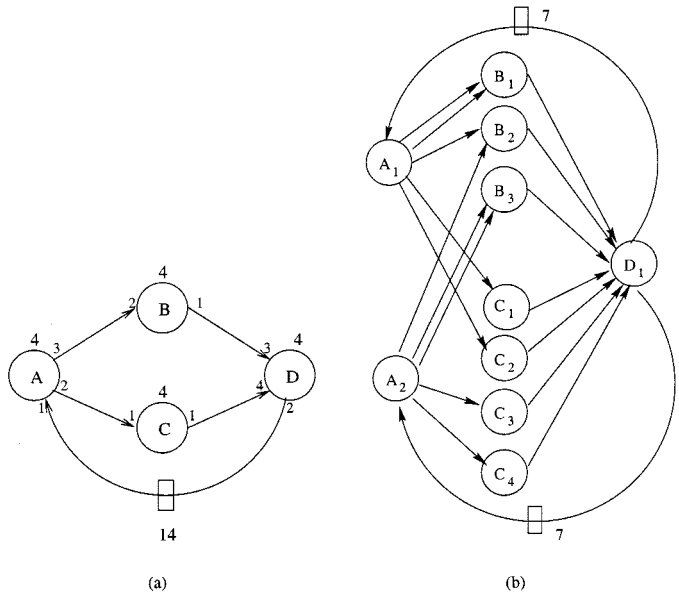


Fig. 7. (a) Example SDFG. (b) Its EHG.

derive five 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

$$\begin{aligned} 2r(B) - 3r(A) &\leq -6 \\ r(C) - 2r(A) &\leq -4 \\ 3r(D) - r(B) &\leq -3 \\ 4r(D) - r(C) &\leq -4 \\ r(A) - 2r(D) &\leq 12. \end{aligned}$$

Condition 1 also gives us five 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; therefore, it suffices to consider only the constraints derived above.

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

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

As in [8], we can now model our set of inequalities by the constraint graph of Fig. 8(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; we can verify that dividing and then normalizing yields an incorrect answer.) Applying this function to the SDFG of Fig. 7(a) yields the retimed graph of Fig. 8(b), whose EHG appears in Fig. 8(c). An examination of this EHG reveals that we have indeed found a retiming that 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.

C. 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 enough delays 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 2: the $O(|E|)$-time algorithm from [18] 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.

Algorithm 2 Find computation time of most expensive zero-delay path to all vertices

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

Output: The $|V|$-length vector τ

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 τ

With this in hand, we can now proceed to our primary method, given as Algorithm 3. We begin by retiming our SDFG with the result to date and constructing its EHG. The EHG is then handed to Algorithm 2 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 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

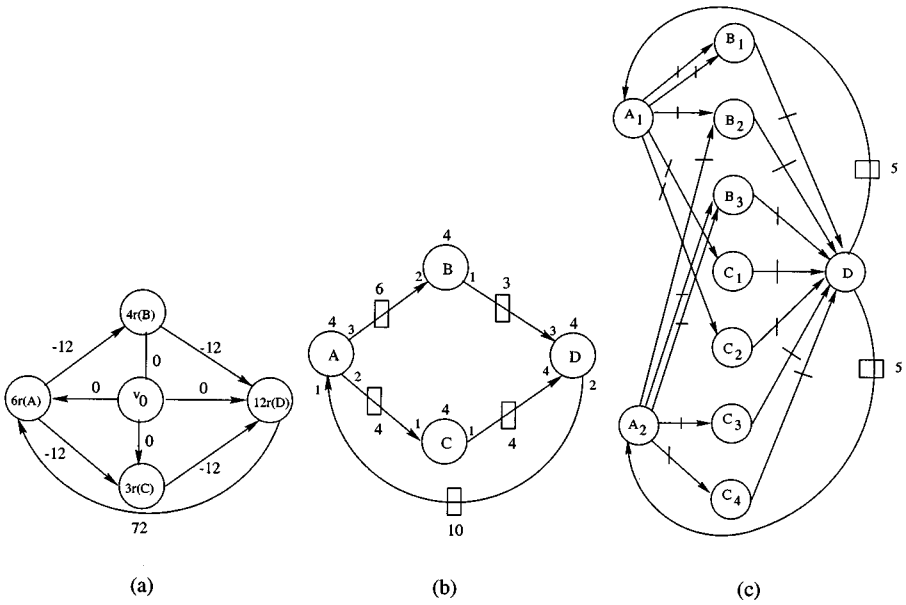


Fig. 8. (a) Constraint graph for inequality system. (b) Fig. 7(a) retimed. (c) Its EHG.

consideration. Otherwise, a copy of the node u lies along some zero-delay path in the EHG, and we may have to retime it. In this case, we 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 that 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 Fig. 7(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 does not 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.

Algorithm 3 Retime an SDFG via Relaxation

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

Output: A retiming r such that $\mathrm{cl}(G_r) \le c$ if one exists

```
for all v ∈ V do
  r(v) ← 0
end for
for i ← 1 to |V| do
  Construct the retimed graph G_r given the current r
  Construct the EHG H = ⟨V', E', d', t'⟩ for G_r
  τ ← MaxPath(H)      /* Apply Algorithm 2 */
  if max{τ(v) | v ∈ V'} ≤ 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
      Υ(v) ← ∞
    else
      Υ(v) ← max{τ(v_i) | v_i is a copy of v in H}
    end if
  end for
  for all v with Υ(v) ≤ c do
    r(v) ← r(v) + max{⌈(q_v · p(e) − d_r(e))/p(e)⌉ | e : v → u in G_r for some u}
  end for
  for all e : u → 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
Υ ← MaxPath(H)      /* Apply Algorithm 2 again */
if max{Υ(v) | v ∈ V'} > c then
  return FALSE      /* No feasible retiming */
else
  return r      /* Otherwise r is the retiming */
end if
```

We now demonstrate our method by executing it on the SDFG of Fig. 7(a) with $c = 4$. Sorting the vertices of Fig. 7(b), com-

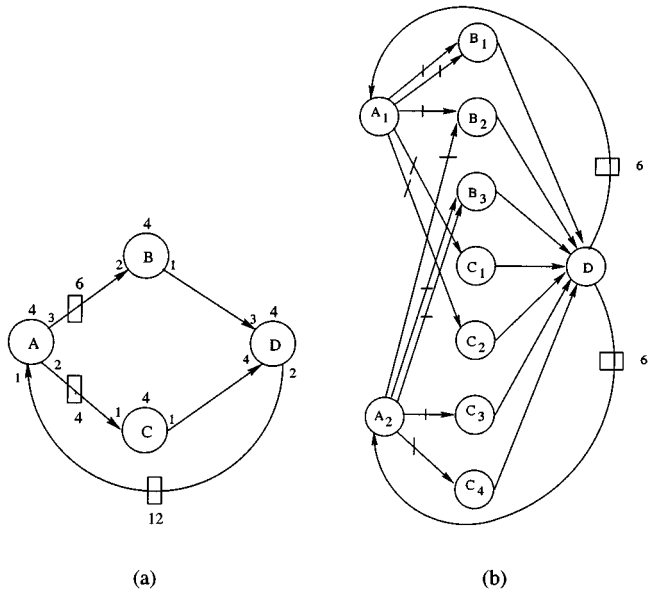


Fig. 9. (a) Retimed SDFG after one pass of algorithm. (b) Its EHG.

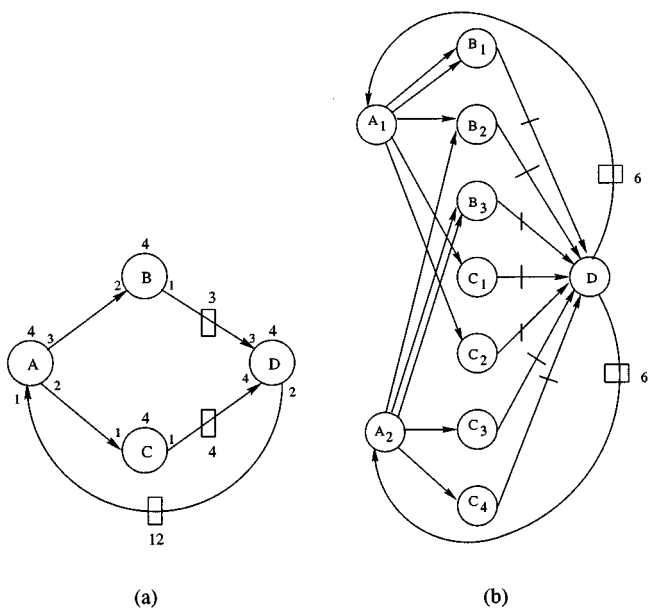


Fig. 10. (a) Retimed SDFG after two passes of algorithm. (b) Its EHG.

puting 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 six delays onto (A, B) and four onto (A, C), as seen in the retimed graph in Fig. 9(a), with its EHG appearing in Fig. 9(b).

Looping back around, we see from Fig. 9(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 $r(A) = 2$, $r(B) = q_B = 3$, $r(C) = q_C = 4$, and $r(D) = 0$. The new retimed graph is given below as Fig. 10(a), with its EHG in Fig. 10(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; therefore, 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 Fig. 8(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 Fig. 8(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.

D. 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 $O\left(|V|^3\right)$ 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.

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 an 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.

V. Examples

In this section, we illustrate our methods further by applying them to various SDFGs found in the literature. Additional examples may be found in [19].

A. First Example

Let us begin with a slowed version of the example from [30] given in Fig. 11(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 that emanate from either B or C, we have $r(v) = q_v =$

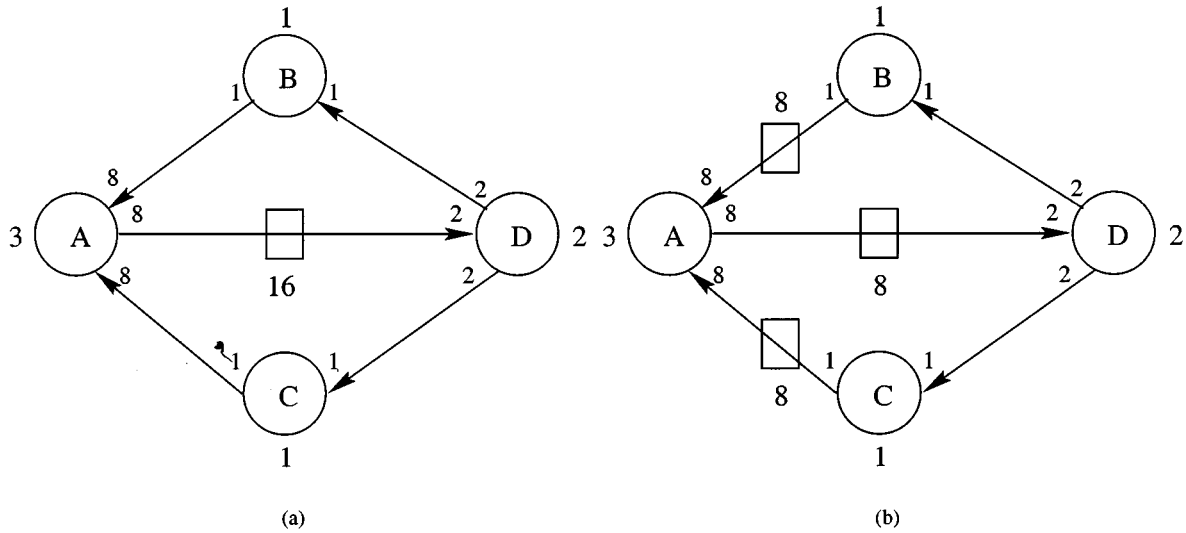
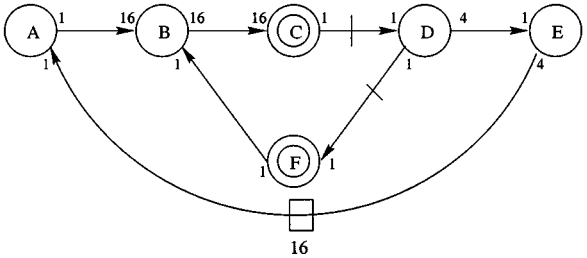


Fig. 11. (a) Another sample SDFG. (b) Fig. 11(a) retimed.



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

(b)

Fig. 12. Simplified spectrum analyzer.

8 for $v = B$ or C. On the other hand, two edges have D as source node, both with production rate 2. Since $q_D = 4, r(D) = (8-0)/2 = 4$. This is a legal retiming, and when we begin the next pass of our loop, we find that it is adequate for retiming Fig. 11(a) to have clock period 3, thus terminating execution of our algorithm with the retimed graph in Fig. 11(b), which executes within our desired time interval.

B. Simplified Spectrum Analyzer

Finally, let us apply our algorithms to a variation of the simplified spectrum analyzer from [28], which appears in Fig. 12(a), with node descriptions in Fig. 12(b). This graph has an RV of $q = [16\ 1\ 1\ 1\ 4\ 1]^T$; therefore, in the interests of space, we will not display the EHG at each step. Instead, we will describe the pertinent information. It can be shown that the lower bound on the clock period for this SDFG is 3, and therefore, 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) that 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 retiming, and our formula gives an initial retiming of $r(A) = p((A,B)) = 16$, $r(B) = p((B,C))/p((B,C)) = 1$, $r(C) = 0, r(D) = \max\{(p((D,E))/p((D,E))), p((D,F)) - 1\} = 1$, $r(E) = (q_E \cdot p((E,A)) - 16)/(p((E,A))) = 0$, 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 Fig. 12(a) yields the graph of Fig. 13. 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.

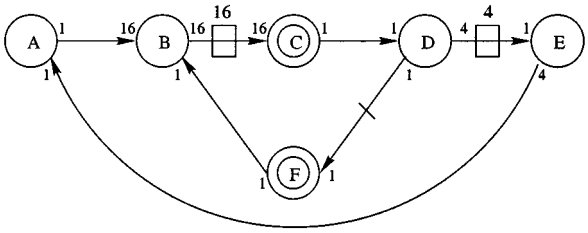


Fig. 13. Retimed analyzer.

VI. 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 a polynomial-time algorithm (in the size of the SDFGs homogeneous equivalent) for retiming a synchronous graph so that it achieves a sufficiently small clock period. Finally, we have demonstrated the effectiveness of our

algorithm 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 algorithm may be, it is still not proven to represent both necessary and sufficient conditions for retiming. This proof, or the construction of an alternate method that is necessary and sufficient, remains an interesting open problem. A better grasp of the results regarding delay counts in [28] 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*. Boston, MA: Kluwer, 1996.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cyclo-static dataflow," *IEEE Trans. Signal Processing*, vol. 44, pp. 397–408, Feb. 1996.

[3] L.-F. Chao, "Scheduling and behavioral transformations for parallel systems," Ph.D. Thesis, Dept. Comput. Sci., Princeton Univ., Princeton, NJ, 1993.

[4] L.-F. Chao and E. H.-M. Sha, "Retiming and unfolding data-flow graphs," in *Proc. Int. Conf. Parallel Process.*, vol. II, 1992, pp. 33–40.

[5] ——, "Static scheduling for synthesis of DSP algorithms on various models," *J. VLSI Signal Process.*, vol. 10, pp. 207–223, 1995.

[6] ——, "Scheduling data-flow graphs via retiming and unfolding," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, pp. 1259–1267, Dec. 1997.

[7] F. Commoner, A. W. Holt, S. Even, and A. Pnueli, "Marked directed graphs," *J.Comput. Syst. Sci.*, vol. 5, pp. 511–523, 1971.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill, 1991.

[9] G. Gao, R. Govindarajan, and P. Panangaden, "Well-behaved dataflow programs for DSP computation," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, vol. 5, 1992, pp. 561–564.

[10] P. D. Hoang and J. M. Rabaey, "Scheduling of DSP programs onto multiprocessors for maximum throughput," *IEEE Trans. Signal Processing*, vol. 41, pp. 2225–2235, June 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 Acoust., Speech, Signal Processing Mag.*, vol. 7, pp. 32–43, 1990.

[12] E. A. Lee, "A coupled hardware and software architecture for programmable digital signal processors," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. Calif., Berkeley, CA, 1986.

[13] E. A. Lee, W. H. Ho, E. Goei, J. Bier, and S. S. Bhattacharya, "Gabriel: A design environment for DSP," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. 37, pp. 1751–1762, Nov. 1989.

[14] E. A. Lee and D. G. Messerschmitt, "Pipeline interleaved programmable DSP's: Synchronous data flow programming," *IEEE Trans. Acoust., Speech, Signal Processing*, vol. ASSP-35, pp. 1334–1345, July 1987.

[15] ——, "Static scheduling of synchronous data-flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. C–36, pp. 24–35, Jan. 1987.

[16] ——, "Synchronous data flow," *Proc. IEEE*, vol. 75, pp. 1235–1245, 1987.

[17] C. E. Leiserson and J. B. Saxe, "Optimizing synchronous systems," *J. VLSI Comput. Syst.*, vol. 1, no. 1, pp. 41–67, 1983.

[18] ——, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[19] T. W. O'Neil and E. H.-M. Sha, "Retiming synchronous data-flow graphs to minimize execution time," Comput. Sci. Eng. Dept., Univ. Notre Dame, Notre Dame, IN, Tech. Rep. TR–00-08, 2000.

[20] K. K. Parhi, "Algorithm transformation techniques for concurrent processors," *Proc. IEEE*, vol. 77, pp. 1879–1895, Sept. 1989.

[21] J. L Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for DSP using Ptolemy," *J. VLSI Signal Process.*, vol. 9, pp. 7–21, 1995.

[22] M. Renfors and Y. Neuvo, "The maximum sampling rate of digital filters under hardware speed," *IEEE Trans. Circuits Syst.*, vol. CAS-28, pp. 196–202, 1981.

[23] S. Ritz, M. Pankert, V. Zivojnovic, and H. Meyr, "High-level software synthesis for the design of communication systems," *IEEE J. Select. Areas Commun., Special Issue on Computer-Aided Modeling, Analysis and Design of Communication Links*, vol. 11, pp. 348–358, Apr. 1993.

[24] ——, "Optimum vectorization of scalable synchronous dataflow graphs," *Proc. Int. Conf. Appl.-Specific Array Processors*, pp. 285–296, 1993.

[25] R. Schoenen, V. Zivojnovic, and H. Meyr, "An upper bound of the throughput of multirate multiprocessor schedules," *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process.*, vol. 1, pp. 655–658, 1997.

[26] G. C. Sih, "Multiprocessor scheduling to account for interprocessor communication," Ph.D. dissertation, Dept. Elect. Eng. Comput. Sci., Univ. Calif., Berkeley, 1991.

[27] M. Veiga, J. Parera, and J. Santos, "Programming DSP systems on multiprocessor architectures," in *Proc. Int. Conf. Acoust., Speech, Signal Process.*, vol. 2, 1990, pp. 965–968.

[28] V. Zivojnovic, S. Ritz, and H. Meyr, "Optimizing DSP programs under the multirate retiming transformation," in *Proc. 7th Euro. Signal Process. Conf.*, vol. 3, 1994, pp. 1597–1600.

[29] ——, "Retiming of DSP programs for optimum vectorization," *Proc. Int. Conf. Acoust., Speech, Signal Process.*, vol. 1, pp. 465–468, 1994.

[30] V. Zivojnovic and R. Schoenen, "On retiming of multirate DSP algorithms," *Proc. Int. Conf. Acoust., Speech, Signal Process.*, vol. 6, pp. 3310–3313, 1996.

Timothy W. O'Neil (S'00) received the B.S. degrees in secondary education-mathematics and computer science from Clarion University of Pennsylvania in 1989. He received M.S. degrees in mathematics and computer and information sciences in 1991 and 1993, respectively, both from The Ohio State University, Columbus. He is currently pursuing the Ph.D. degree in computer science at the University of Notre Dame, Notre Dame, IN.

His current research interests include loop transformations and data scheduling.

Edwin H.-M. Sha (S'88–M'92) received the B.S.E. degree in computer science and information engineering from National Taiwan University, Taipei, Taiwan, R.O.C., in 1986 and the M.A. and Ph.D. degrees from the Department of Computer Science, Princeton University, Princeton, NJ, in 1991 and 1992, respectively.

From August 1992 to August 2000, he was with the University of Notre Dame, Notre Dame, IN. He served as Associate Chairman of the Department of Computer Science and Engineering since 1995. He is now a Tenured Full Professor and the Head of the Computer Science Division with the Department of Computer Science, University of Texas at Dallas, Richardson. His research interests include high-performance computer and network architectures, real-time systems and communications, parallel processing, embedded systems for DSP and communications, hardware/software co-designs, high-level synthesis, application-specific VLSI systems, Java and multimedia networking systems, and parallel and distributed systems. He has published more than 130 research papers in refereed conferences and journals.

Dr. Sha has been serving as an editor for several journals, such as IEEE TRANSACTIONS ON SIGNAL PROCESSING and *Journal of VLSI Signal Processing*. In 1994, He served as the Program Committee Chair for the Fourth IEEE Great Lakes Symposium on VLSI. He also served as program committee members in numerous conferences such as the International Symposium on System Synthesis, the International Conference on Parallel and Distributed Computing and Systems, and the International Symposium on the Frontiers of Massively Parallel Computation. He received Oak Ridge Association Junior Faculty Enhancement Award in 1994 and the National Science Foundation CAREER Award. He was a Guest Editor for the Special Issue on Low Power Design of the IEEE TRANSACTIONS ON VLSI SYSTEMS in 1997. He is also serving as the Program Chairs for the International Conference on Parallel and Distributed Computing Systems in 2000 and 2001. He received the CSE Undergraduate Teaching award in 1998.