Efficient Retiming of Multi-rate DSP Algorithms

Xue-Yang Zhu, Twan Basten, Marc Geilen, and Sander Stuijk

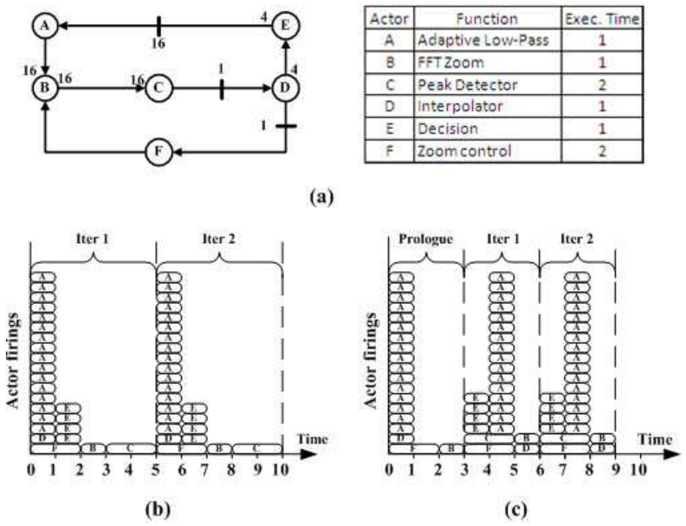
***Abstract*—Multi-rate digital signal processing (DSP) algorithms are often modeled with synchronous dataflow graphs (SDFGs). A lower iteration period implies a faster execution of a DSP algorithm. Retiming is a simple but efficient graph transformation technique for performance optimization, which can decrease the iteration period without affecting functionality. In this paper, we deal with two problems: feasible retiming — retiming an SDFG to meet a given iteration period constraint; and optimal retiming — retiming an SDFG to achieve the smallest iteration period. We present a novel algorithm for feasible retiming and based on that one, a new algorithm for optimal retiming, and prove their correctness. Both methods work directly on SDFGs, without explicitly converting them to their equivalent homogeneous SDFGs (HSDFGs). Experimental results show that our methods give a significant improvement compared to the earlier methods.**

***Index Terms*—retiming, multi-rate digital signal processing, synchronous dataflow graphs, iteration period, optimization**

I. Introduction and related work

Data-driven execution and real-time requirements are two important features of digital signal processing (DSP) systems. Dataflow models of computation are widely used to represent DSP applications. Each node (also called actor) in such a model represents a computation or function and each edge models a FIFO channel. One of the most useful dataflow models for designing multi-rate DSP algorithms are *synchronous dataflow graphs* (SDFGs) [1], also called *multi-rate dataflow graphs*. The sample rates of actors of an SDFG may differ. The graph *SSA* in Fig. 1(a), for example, is an SDFG model of a simplified spectrum analyzer [2], [3].

DSP algorithms are often repetitive. Execution of all the computations for the required number of times is referred to as an *iteration*. A DSP algorithm repeats iterations periodically. An iteration of *SSA* in Fig. 1(a), for example, includes 16 executions, often called *firings* in dataflow, of actor A, 4 firings of E and 1 firing of all other actors. Actor firings synchronize through *delays* or *tokens*, shown with annotated bars in Fig. 1(a). Actor B, for example, needs 16 tokens from edge $\langle A, B\rangle$ and 1 from $\langle F, B\rangle$ to fire, and produces 16 tokens on $\langle B, C\rangle$. The *iteration period* of an SDFG is the least time



Actor	Function	Exec. Time
A	Adaptive Low-Pass	1
B	FFT Zoom	1
C	Peak Detector	2
D	Interpolator	1
E	Decision	1
F	Zoom control	2

Fig. 1. (a) The SDFG model *SSA* of a simplified spectrum analyzer, where the sample rates are omitted when they are 1; (b) the periodic schedule of *SSA*; (c) the schedule with prologue.

required for executing one iteration of a DSP algorithm [4]. The iteration period of *SSA*, for example, is 5 as shown by the periodic schedule of Fig. 1(b). The data dependencies among actors F, B and C imply that the iteration period of *SSA* cannot be lower than 5 no matter which periodic schedule is used. That is because the iteration period of a dataflow graph is limited by its topology, the computation time of its actors, and the delay distribution. The *delay distribution* indicates the numbers of initial tokens on edges of the SDFG. To decrease the iteration period and speed up a DSP algorithm, we need therefore to change any of these three factors.

Restructuring an SDFG or rewriting functions of actors to adjust the computation time involves redesign of the whole model, which may need much effort, e.g. to verify the correctness of each actor and the functional correctness of the restructured model. If the new model still has a high iteration period, designers have to repeat the procedure again. An optimization method that preserves the functionality of the original SDFG is a preferred choice. Redistributing delays of an SDFG provides an opportunity to do so.

Consider the SDFG *SSA* in Fig. 1(a) again. A schedule of *SSA* with prologue, shown in Fig. 1(c), consists of some initial actor firings followed by iterations whose execution time is shorter than the iteration period of *SSA*. Executing the prologue of *SSA*, that is, firing actor A 16 times, and firing D, F and B once, respectively, leads to the graph in Fig. 2(a), whose delays are redistributed and whose iteration period is reduced to 3, as its periodic schedule, Fig. 2(b), shows.

Fig. 2(a) is in fact a graph obtained by *retiming SSA* in Fig. 1(a), and it is called a retimed graph of *SSA*. Retiming is

This work was supported in part by the National Natural Science Foundation of China, under grant No. 60833001.

X. Y. Zhu is with State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China (e-mail: zxy@ios.ac.cn).

T. Basten is with the Department of Electrical Engineering, Eindhoven University of Technology, and the Embedded Systems Institute, Eindhoven, the Netherlands (e-mail: a.a.basten@tue.nl).

M. Geilen and S. Stuijk are with the Department of Electrical Engineering, Eindhoven University of Technology, Eindhoven, the Netherlands (e-mail: m.c.w.geilen@tue.nl; s.stuijk@tue.nl).

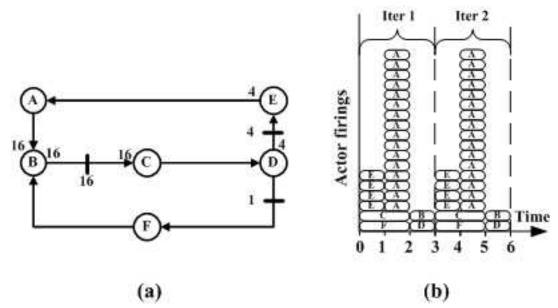


Fig. 2. (a) A retimed equivalent graph of *SSA*; (b) the periodic schedule of the retimed *SSA*.

a graph transformation technique that only changes the delay distribution of a graph, while it has no effect on its functionality [5]. A *retiming function* specifies the firings required to redistribute the delays. Retiming is originally applied in [5] to reduce the iteration period of *homogeneous synchronous dataflow graphs* (HSDFGs), which is a special type of SDFGs. Retiming can also be used to optimize algorithms according to other criteria, such as minimizing the memory usage [2], extending the vectorization capabilities [6], and decreasing power consumption [7]. A great deal of research has been done on retiming in the context of HSDFGs [5], [8], [9], [10], [11].

Different from an HSDFG, however, an iteration of an SDFG may execute actors more than once and a different number of times. This invalidates many results derived for HSDFGs and complicates the analysis of retiming properties of SDFGs. Nevertheless, some efforts have been made. Some important properties of retiming on SDFGs, such as functional equivalence and reachability, were proven in [12] and [2].

When retiming is used to reduce the iteration period of an SDFG, we can distinguish two problems: the first one is a decision problem, checking whether a *feasible retiming* of the SDFG exists so that the retimed graph has an iteration period of at most a given value; the second one is an optimization problem, finding an *optimal retiming* of the SDFG so that the retimed graph has an iteration period as small as possible.

The traditional way to solve these problems is to first convert the SDFG to its equivalent HSDFG and then to use the available methods for HSDFGs [13]. Theoretically, this is always possible. However, converting an SDFG to an HSDFG may increase the problem size tremendously and it is very time-consuming when SDFGs scale up. The size of the HSDFG can be exponentially larger than the original SDFG in extreme cases [12].

We focus in this paper on methods that directly work on SDFGs without converting them to HSDFGs. Issues that need to be considered for a correct and efficient feasible/optimal retiming method include:

1) Whether the iteration period is computed directly on SDFGs.
2) Whether the retiming function is computed directly on SDFGs.
3) Whether a feasible retiming method is sufficient and necessary. That is, when an algorithm returns a retiming, whether it is a feasible retiming for the input, and if there exist feasible retimings for the input, whether it is guaranteed to find one.
4) Whether the termination condition of the method is sharp. Sharpness of the termination condition means that the algorithm terminates once a conclusion can be drawn, so that no redundant work is performed.

O'Neil et al. [3] propose a feasible retiming method that computes the retiming function directly on SDFGs, but the computation for the iteration period is still on HSDFGs. Thus, the method still needs to convert an SDFG to an HSDFG. The method is sufficient but not necessary. Our experimental results show that the method fails in some cases. The method in [14] for feasible retiming computes both the iteration period and retiming function on SDFGs and it is sufficient. But the paper says nothing about necessity. Liveris et al. [15] present an optimal retiming method that computes both the iteration period and retiming function on SDFGs but it does not deal with the feasible retiming problem, although of course it can be adapted straightforwardly. They use a complex termination condition and prove that it is sufficient for an optimal retiming to be found, but no evidence is given that it is sharp.

In this paper, we deal with both the feasible retiming and optimal retiming problems directly on SDFGs and address all above-mentioned issues. Our contributions are as follows.

1) We present a new method to compute the iteration period of an SDFG.
2) We present a feasible retiming algorithm and prove that it is a sufficient and necessary solution and with a sharp termination condition.
3) We present an optimal retiming algorithm based on the feasible retiming algorithm, with a sharp termination condition.

For evaluating the efficiency of our new algorithms and other related methods, we implemented them in the open source tool SDF3 [16]. The methods for feasible retiming in our comparison include the traditional method [13], 'byHSDF', O'Neil's algorithm [3], 'O'Neil01', the method in [14], 'Zhu10', and our new method, 'sdfFEAS'. The methods for optimal retiming include Liveris's algorithm [15] , 'Liveris07', and our new method, 'sdfOPT'. The experimental results, based on hundreds of synthetic SDFGs and several realistic SDFG models, show that our new methods lead to a significant improvement compared to earlier methods.

The remainder is organized as follows. We first describe the main relevant concepts in Section II. The basic ideas of our new methods are introduced in Section III and the details are illustrated in Sections IV to VI. Section VII provides an experimental evaluation. Finally, Section VIII concludes.

II. Preliminaries

In this section, we introduce the main concepts and notations used in the paper, mainly including synchronous dataflow graphs (the model we use), iteration period (the goal we try to optimize), and retiming (the technique we use).

A. Synchronous Dataflow Graphs

Definition 1. A *synchronous dataflow graph* (SDFG) is a finite directed multigraph $G = \langle V, E, t, d, prd, cns \rangle$, in which

- V is the set of actors, modeling the functional elements of the system. Each actor $v \in V$ is weighted with its computation time $t(v)$, a nonnegative integer;
- E is the set of directed edges, modeling interconnections between functional elements. Each edge $e \in E$ is weighted with three properties: $d(e)$, the number of initial tokens associated with e; $prd(e)$, a positive integer that represents the number of tokens produced onto e by each execution of the source actor of e; $cns(e)$, a positive integer that represents the number of tokens consumed from e by each execution of the sink actor of e. These numbers are also called the *delay*, *production rate* and *consumption rate*, respectively. The source actor and sink actor of $e \in E$ are denoted as $src(e)$ and $snk(e)$, respectively.

We represent the edge e with source actor u and sink actor v mostly by $e = \langle u, v\rangle$, and by $e = \langle u, v, k\rangle$, where k numbers the edges connecting u to v, only when distinguishing different edges between two actors is necessary. The set of incoming edges to $v \in V$ is denoted by $InE(v)$, and the set of outgoing edges from $v \in V$ by $OutE(v)$. We use $v \in G$ to represent that v is an actor of G and $e \in G$ to represent that e is an edge of G. Note that for technical reasons explained later, we allow $d(e)$ to be negative. If $prd(e) = cns(e) = 1$ for each $e \in E$, then we say that G is a *homogeneous synchronous dataflow graph* (HSDFG), also called a *single-rate dataflow graph*. We represent an HSDFG as $G_h = \langle V, E, t, d\rangle$.

A *delay distribution* of an SDFG is a vector containing delays on all edges of the SDFG G, denoted as $d(G)$. For example, the delay distribution of *SSA* in Fig. 1(a) is $d(SSA) = [0, 0, 1, 0, 16, 1, 0]$ corresponding to the edges $\langle A, B\rangle, \langle B, C\rangle, \langle C, D\rangle, \langle D, E\rangle, \langle E, A\rangle, \langle D, F\rangle$ and $\langle F, A\rangle$.

Applications for signal processing are usually nonterminating. Memory used must be bounded no matter how many times they are executed. In order that an SDFG G has a well-defined meaning, according to [1], we place restrictions on it:

VS1. $d(e) \geq 0$ for each $e \in G$.

VS2. G is bounded. *Boundedness* means that, if infinite execution sequences exist, then there are some for which the number of tokens on every edge is always finite.

VS3. G is live. *Liveness* means that there is no execution sequence leading to a deadlock.

An SDFG that satisfies conditions VS1, VS2, and VS3 is a *valid SDFG*.

An SDFG G is *sample-rate consistent* if there exists a positive integer vector $q(V)$ such that for each edge $e \in G$,

$$q(src(e)) \times prd(e) = q(snk(e)) \times cns(e), \tag{1}$$

where (1) is called a *balance equation*. The smallest q is called the *repetition vector* [1]. We use q to represent the repetition vector directly. One *iteration* of an SDFG G is an execution sequence in which each actor v in G occurs exactly $q(v)$ times.

Take the SDFG *SSA* in Fig. 1(a) for example. For each edge e, its $prd(e)$, $cns(e)$ that are not equal to 1 and its $d(e)$ are labeled on e. The computation time vector $t = [1, 1, 2, 1, 1, 2]$. A balance equation can be constructed for each edge. By solving these balance equations, we get *SSA*'s repetition vector $q = [16, 1, 1, 1, 4, 1]$.

An SDFG is sample-rate inconsistent if there is no nonzero solution for its balance equations. Any execution of an inconsistent SDFG will result in deadlock or unbounded memory.

Property 2. [1] A live SDFG is bounded *if and only if* it is sample-rate consistent.

Property 2 gives a necessary and sufficient condition for boundedness. We consider a necessary and sufficient condition for liveness in the next subsection.

B. Equivalent HSDFG

A sample-rate consistent SDFG can always be converted to an equivalent HSDFG, which captures the data dependencies among firings of actors in the original SDFG in an iteration.

In an iteration of an SDFG G, each actor v fires $q(v)$ times. We can map the i^{th} firing of v to an actor (v, i) in its equivalent HSDFG. Each edge $e = \langle u, v\rangle$ in G has $q(v)cns(e)$ instances in this HSDFG. For each actor (u, i) in the equivalent HSDFG, the (v, j)s to which it is connected are determined by the production and consumption rates and delays on e. If there are delays on e, they are always consumed by initial firings of v. For example, there is one delay on the edge $\langle A, B\rangle$ of G_1 in Fig. 3(a). According to the consumption rate of $\langle A, B\rangle$, the first firing of B will consume this initial token and 2 other tokens produced by the first firing of A. These data dependencies map to its equivalent HSDFG in Fig. 3(b) as two zero-delay edges connecting $(A, 1)$ and $(B, 1)$ and one edge connecting $(A, 3)$ and $(B, 1)$ with one delay.

Algorithms for transforming an SDFG to its equivalent HSDFG appear in the literature [17], [18]. For developing our ideas, we now formalize the transformation as a map.

Definition 3. [17] Let $G = \langle V, E, t, d, prd, cns\rangle$ be a sample-rate consistent SDFG and q its repetition vector. H maps G to its equivalent HSDFG $H(G) = \langle V', E', t', d'\rangle$ as follows:

3-1. For each $v \in V$ and $i \in [1, q(v)]$, there is an actor $(v, i) \in V'$ with $t'(v, i) = t(v)$;

3-2. For each $e = \langle u, v\rangle \in E$, $i \in [1, q(u)]$, $k \in [1, prd(e)]$, and

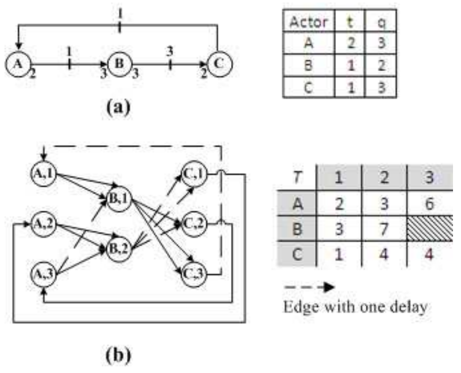
$$j = \left\lfloor \frac{((i-1)prd(e) + (k-1) + d(e)) \bmod cns(e)q(v)}{cns(e)} \right\rfloor + 1,$$

there is an edge $e' = \langle (u, i), (v, j), k\rangle \in E'$ with

$$d'(e') = \left\lfloor \frac{(i-1)prd(e) + (k-1) + d(e)}{cns(e)q(v)} \right\rfloor.$$

For $(v, i) \in H(G)$, we call it the i^{th} copy or the i^{th} firing of $v \in G$; we call i the *label* of (v, i). As described later, only the edges with zero delays in an HSDFG are of interest in our methods. According to Definition 3-2, d' is nondecreasing with k. Therefore, for an equivalent HSDFG of a valid SDFG, if there exist edges between the actors (u, i) and (v, j) with zero delays, then the edge $\langle (u, i), (v, j), k_0\rangle$, where k_0 is the lowest value among the edges between (u, i) and (v, j), has zero delays. For conciseness, we use $\langle (u, i), (v, j)\rangle$ to represent $\langle (u, i), (v, j), k_0\rangle$ in the following description.

A *path* in a graph is a sequence of actors and edges, containing no actor twice except, possibly, the first and last



Actor	t	q
A	2	3
B	1	2
C	1	3

T	1	2	3
A	2	3	6
B	3	7	▨
C	1	4	4

Fig. 3. (a) The SDFG G_1; (b) its equivalent HSDFG $H(G_1)$ and T.

actors. A path is *trivial* if it includes only a single actor without edges. A *cycle* is a nontrivial path that begins and ends at the same actor. We extend the delay function d of an HSDFG from single edges to arbitrary nontrivial paths. For any path

$$p : v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \ldots \xrightarrow{e_n} v_n$$

in the HSDFG, its *path delay* is the sum of the delays of the edges in the path:

$$d(p) = \sum_{i=1}^{n} d(e_i).$$

If $d(p) = 0$, then we say p is a *zero-delay path*, e.g., the path in Fig. 3(b):

$$P_1 : (A, 1) \to (B, 1) \to (C, 2) \to (A, 3) \to (B, 2).$$

Similarly, we define the *path computation time* of p as the sum of the computation times of the actors in the path p:

$$t(p) = \sum_{i=0}^{n} t(v_i).$$

To check if a sample-rate consistent SDFG is live, two methods exist. One is to construct a single-processor schedule for one iteration of the SDFG; if such a schedule exists, then the SDFG is live [1]. Another way is to check if a zero-delay cycle in its equivalent HSDFG exists; if not, then it is live [19].

Property 4. [19] A sample-rate consistent SDFG G is live *if and only if* there is no zero-delay cycle in $H(G)$.

By inserting precedence constraints with a finite number of delays between the source and sink actors of an SDFG, or by dividing the SDFG into strongly connected components, any SDFG can be converted to a strongly connected graph [5], [2]. We therefore only consider strongly connected graphs when developing our ideas. Our methods can also be applied for weakly connected graphs as we show in Section VII.

C. Iteration Period

One iteration of an HSDFG is an execution in which each actor occurs exactly *once*. The iteration period of an HSDFG essentially corresponds to the clock period of a circuit [5].

A *maximal zero-delay path* to an actor v of an HSDFG is a zero-delay path whose first actor has no zero-delay incoming edges and whose last actor is v. We use $maxZDPto(v)$ to represent the set of maximal zero-delay paths to an actor v. The *earliest completion time* of each $v \in G_h$ is the earliest possible finish time of v, denoted by the entry $T(v)$ of vector T, determined by the maximal computation time of all maximal zero-delay paths to v:

$$T(v) = \max_{p \in maxZDPto(v)} t(p).$$

For example, there are two maximal zero-delay paths to $(B, 2)$ in Fig. 3(b), P_1 already given before, and

$$P_2 : (C, 1) \to (A, 2) \to (B, 2).$$

Then $T(B, 2) = \max\{t(P_1), t(P_2)\} = t(P_1) = 7$.

The *iteration period* (IP) of G_h, $IP(G_h)$, is the maximum earliest completion time of all actors [20]:

$$IP(G_h) = \max_{v \in G_h} T(v).$$

For Fig. 3(b), for example, the maximal earliest completion time is $T(B, 2) = 7$, which gives $IP(H(G_1)) = 7$.

An SDFG allows each actor to be executed more than once per iteration, and two actors are not required to execute the same number of times in an iteration. Therefore, its iteration period cannot be defined as straightforwardly as that of an HSDFG. However, since an SDFG G and $H(G)$ express the identical behavior, we can use the definition for the HSDFG. For an actor (v, i) in $H(G)$, $T(v, i)$ is the earliest completion time of the i^{th} firing of v in G, then the maximum of $T(v, i)$s is the IP of G:

$$IP(G) = IP(H(G)) = \max_{v \in G, i \in [1, q(v)]} T(v, i).$$

For example, according to this definition, the iteration period of G_1 in Fig. 3(a) is also 7. For conciseness, we use $T(v, i)$ rather than $T((v, i))$ as the entry of T.

Intuitively, for an actor v in an SDFG G, $T(v, i)$ is the completion time of the i^{th} firing of v in a self-timed execution [21] [22] of one iteration of G. It is easy to see that a later firing of v cannot start before an earlier firing, and therefore cannot complete before it. That is,

$$T(v, i) \leq T(v, j) \text{ for } i < j. \tag{2}$$

D. Retiming

Retiming is a graph transformation that redistributes the graph's delays while its functionality remains unchanged. Retiming can be defined either in a forward fashion, by which retiming an actor once means firing this actor once [2] [3], or in a backward fashion, by which retiming an actor once means reversed firing this actor once [5] [15] [14]. The two approaches to retiming are equivalent, in the sense that any retimed graph can be obtained through both a forward retiming and a backward retiming. We try to shorten zero-delay paths by retiming with a backward strategy, like [5], so the latter definition is used in this paper.

Given an SDFG $G = \langle V, E, t, d, prd, cns \rangle$, a *retiming* of G is a function $r : V \to \mathbb{Z}$, specifying a transformation r of G into a new SDFG $r(G) = \langle V, E, t, d_r, prd, cns \rangle$, where the delay-function d_r is defined for each edge $u \xrightarrow{e} v$ by the equation:

$$d_r(e) = d(e) + cns(e)r(v) - prd(e)r(u). \tag{3}$$

A retiming r of a valid SDFG G is *legal* if the retimed graph $r(G)$ is a valid SDFG. It is sufficient to check if $r(G)$ satisfies **VS1** to ensure that a retiming is legal [14].

Given a valid SDFG G and a desired iteration period — a nonnegative integer dip, a retiming r of G is a *feasible retiming* if $r(G)$ is valid and $IP(r(G)) \leq dip$. We say that dip is a *feasible IP* of G or dip is feasible for G if a feasible retiming exists for G with dip. Fig. 2(a), for example, is the retimed graph of Fig. 1(a) obtained by the feasible retiming $R_2 = [0, 0, 4, 0, 1, 0]$, or equivalently $R_3 = [-16, -1, 0, -1, 0, -1]$, with $dip = 3$. The first retiming, R_2, corresponds to the reversed firings of the C and E actor, whereas R_3 captures the 'forward' prologue given in Fig. 1(c), leading to negative entries to the function.

An *optimal retiming* r of G is a feasible retiming to make $IP(r(G))$ as small as possible. We call such an $IP(r(G))$ the *optimal iteration period* (optIP) of G.

III. Basic Idea of Our Methods

The underlying idea of our study is to make use of the inter-relation between an SDFG and its equivalent HSDFG. Liveris et al. [15] were the first to explore the relation between an SDFG and its equivalent HSDFG to achieve high efficiency in retiming SDFGs. We further elaborate on this inter-relation. Our methods attempt to find certain walks through the SDFG that do not contain sufficient delays to allow the required actor firings for one iteration. A *walk* is a sequence of actors and edges. In contrast to paths, there may be repeated actors or edges in a walk. In the SDFG, these walks may or may not have delays, but in the equivalent HSDFG, these walks correspond to paths without delays. We refer to these walks through the SDFG as *critical walks*. Such critical walks ultimately determine the IP.

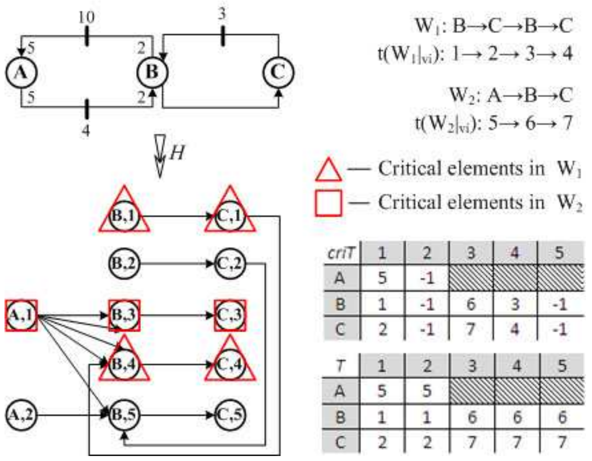
We find the zero-delay paths in the equivalent HSDFG that are longer than the desired IP by going through critical walks in the SDFG. If we manage to insert a proper number of delays to shorten these critical walks, and therefore shorten those corresponding zero-delay paths, without introducing new, longer ones, we will have reduced the IP. This basic procedure is repeated until a feasible retiming has been found or no improvements are possible anymore. Unlike any earlier work, we do not use backtracking to get retiming functions to determine the change of delays in the procedure; instead we compute retiming functions according to part of the computation times of critical walks and their subwalks.

The solution for finding an optimal retiming is a repetitive procedure that checks whether a potentially optimal IP is a feasible IP. Following [15], potentially optimal IPs are checked decreasingly until we find one that is not feasible.

IV. Computing the Iteration Period

According to our definitions in Section II.C, the IP of an SDFG G is derived from the earliest completion time vector T. The T is derived from the maximal zero-delay paths to actors in $H(G)$. That is, for each (v, i) in $H(G)$, $T(v, i)$ is determined by the set $maxZDPto(v, i)$.

One way to compute the IP is to convert G to $H(G)$ and then use Algorithm CP from [5], which is an algorithm computing the complete T and the IP for HSDFGs. This is exactly the method that is used by byHSDF and O'Neil01. This method is time and space consuming due to the conversion procedure from an SDFG to an HSDFG. Another way is to compute the IP directly on the SDFG G without converting it to $H(G)$.



criT	1	2	3	4	5
A	5	-1	▨	▨	▨
B	1	-1	6	3	-1
C	2	-1	7	4	-1

T	1	2	3	4	5
A	5	5	▨	▨	▨
B	1	1	6	6	6
C	2	2	7	7	7

Fig. 4. Example to show properties of critical walks, where $t(A) = 5, t(B) = 1$ and $t(C) = 1$; edges with delays in HSDFGs are omited.

Because of the equivalence of G and $H(G)$, $IP(G)$ can be computed with only a part of T. Liveris07 and Zhu10 use this method. Taking into account the relationship between G and $H(G)$, only a part of T that includes the maximum among the $T(v, i)$s for each v in G is computed to get the IP.

We show in this section that, without converting to $H(G)$, by searching certain walks in the SDFG G, we can get $T(v, i)$s of some actors in $H(G)$. We can get the IP and other information of T that we use in the next section, from those $T(v, i)$s.

A. Critical Walks

A zero-delay path in $H(G)$ always corresponds to a walk in G. For example, in Fig. 4, the path

$$P_{11} : (B, 1) \rightarrow (C, 1) \rightarrow (B, 4) \rightarrow (C, 4)$$

corresponds to walk W_1.

A walk in G may map to more than one zero-delay path in $H(G)$. For example, the walk W_2, shown in Fig. 4, corresponds to zero-delay paths

$$P_{21} :(A, 1) \rightarrow (B, 3) \rightarrow (C, 3), \text{ and}$$
$$P_{22} :(A, 1) \rightarrow (B, 4) \rightarrow (C, 4), \text{ and}$$
$$P_{23} :(A, 1) \rightarrow (B, 5) \rightarrow (C, 5), \text{ and}$$
$$P_{24} :(A, 2) \rightarrow (B, 5) \rightarrow (C, 5).$$

Similar to paths, we define the *walk computation time* of w, $t(w)$, as the sum of the computation times of the actors in w.

It is easy to see that a walk has the same computation time as all its corresponding paths. Then the computation time of a walk may cover a set of paths. For example, $t(P_{2i}) = t(W_2)$ for $i = 1, 2, 3, 4$, in Fig. 4. On the other hand, a walk may be found by only searching according to one of these corresponding zero-delay paths, as we show below.

Definition 5. Let G be a valid SDFG and the actors $u, v \in G$. If there exists a zero-delay path $p \in H(G)$:

$$p : (u, i) = (v_0, l_0) \rightarrow (v_1, l_1) \rightarrow \cdots \rightarrow (v_n, l_n) = (v, j)$$

for some $i \in [1, q(u)]$ and $j \in [1, q(v)]$, then

$$u = v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_n = v$$

is called a *zero-delay reachable* (ZR) walk and v is *zero-delay reachable* from u.

In p in Definition 5, if $v_i = v_j$ with $i \neq j$, then $l_i \neq l_j$, because there is no zero-delay cycle in $H(G)$. This is guaranteed by Property 4.

For example, in Fig. 4, W_1 is a ZR walk because there is a zero-delay path P_{11} in its equivalent HSDFG. And also all the subwalks of W_1 are ZR walks. This example also indicates that, unlike for a zero-delay path in an HSDFG, even if there are delays on the edges of a walk, it still may be a ZR walk.

A ZR walk w of G corresponds to a set of zero-delay paths of $H(G)$, denoted as $ZDPof(w)$. Then for each path p in $ZDPof(w)$, $t(p) = t(w)$.

For an edge $e = \langle u, v\rangle$ in an SDFG G, if $l_{i-1} - 1$ firings of u have produced sufficient tokens for $l_i - 1$ firings of v but not sufficient for the l_i^{th} firing of v, then the l_i^{th} firing of v is the first firing that directly depends on the l_{i-1}^{th} firing of u. Therefore there is a zero-delay edge between (u, l_{i-1}) and (v, l_i) in $H(G)$. When $l_{i-1}-1$ firings of u produce tokens more than the need of $q(v)$ firings of v, that means only some firings of v in the next iteration directly depends on the l_{i-1}^{th} firing of u. So then there is no zero-delay edge from (u, l_{i-1}) to any (v, j). According to Definition 3-2 and always choosing $k = 1$, we get (4) below to model the relation between l_{i-1} and l_i; l_i is the least label of (v, j)s that have zero-delay edges from (u, l_{i-1}).

Theorem 6. [14] Let G be a valid SDFG, and $u, v \in G$. Then v is zero-delay reachable from u *if and only if* there is a path

$$p : (v_0, l_0) \to (v_1, l_1) \to \cdots \to (v_n, l_n)$$

in $H(G)$, where $v_0 = u, v_n = v$, such that for each $i \in [1, n]$:

$$l_i = \left\lfloor \frac{(l_{i-1}-1)prd(e_i)+d(e_i)}{cns(e_i)} \right\rfloor + 1 \tag{4}$$

and $l_i \leq q(v_i)$, where $e_i = \langle v_{i-1}, v_i\rangle \in G$.

This theorem indicates that for each $v_0 \in G$, we can find a ZR walk according to the above-defined p, which is a zero-delay path in $H(G)$. Since the SDFG is valid and finite, there is no zero-delay cycle in its equivalent HSDFG. Therefore the value of l_i is guaranteed to exceed its corresponding $q(snk(e))$ in a finite number of steps.

By Theorem 6, we can find a ZR walk w within a subset of $ZDPof(w)$. For example, see Fig. 4, if we search from actor A and $l_0 = 1$ to its next actor B and get $l_1 = 3 < q(B)$ by (4), going on from B and $l_1 = 3$ to C and get $l_2 = 3 < q(C)$ by (4), we get ZR walk W_2. Searching from A with $l_0 = 2$ leads to path P_{24}, which also belongs to $ZDPof(W_2)$. Fortunately, the redundancy in the search can be avoided. Only searching from $l_0 = 1$ guarantees to find all ZR walks, as shown by the following corollary.

Corollary 7. Let G be a valid SDFG. Walk

$$w : v_0 \xrightarrow{e_1} v_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} v_n$$

is a ZR walk in G *if and only if* there exists a path

$$p : (v_0, l_0) \to (v_1, l_1) \to \cdots \to (v_n, l_n)$$

in $H(G)$, such that in particular $l_0 = 1$ and l_i is defined as in (4) and $l_i \leq q(v_i)$ for each $i \in [1, n]$.

Proof: According to Theorem 6, the sufficiency is obvious. So we have to prove only the necessity. Suppose w is a ZR walk. Then there exists a zero-delay path p of $H(G)$ as described in Theorem 6. We construct a path:

$$(v_0, l'_0) \to (v_1, l'_1) \to \cdots \to (v_n, l'_n)$$

in $H(G)$ with $l'_0 = 1$ and for each $i \in [1, n]$, l'_i is computed by (4). According to (4), l_i is nondecreasing with l_{i-1}. Since $l'_0 \leq l_0$, $l'_i \leq l_i$ for each $i \in [1, n]$. And by $l_i \leq q(v_i)$, we get $l'_i \leq q(v_i)$. ■

By Corollary 7, we can find a ZR walk w according to one path of $ZDPof(w)$. The proof of Corollary 7 also implies that labels l_i of above p are the lowest values among labels of all paths in $ZDPof(w)$. That is, for another path p' in $ZDPof(w)$ with labels l'_i, we have $l_i \leq l'_i$.

Further, we only need those ZR walks corresponding to the *maximal* zero-delay paths of $H(G)$. Those walks can be found by searching only from some of the actors $(v, 1)$. If we can find actors in G that are enabled in the initial state of the graph, grouped in a set V_0, we can begin our search only from these actors. Actors that are not enabled need another actor to fire first, from which there is therefore a zero-delay edge and thus a longer zero-delay path. And by tracing back, we can eventually find a maximal zero-delay path corresponding to a ZR walk beginning with an actor in V_0.

Theorem 8. [14] Let $G = \langle V, E, t, d, prd, cns\rangle$ be a valid SDFG, and $V_0 = \{v \in V : \forall e \in InE(v), d(e) \geq cns(e))\}$. Then we have

8-1. $V_0 \neq \emptyset$; and

8-2. for each $v \in V$, there exists a $u \in V_0$, such that v is zero-delay reachable from u.

Theorem 8 further reduces the ZR walks we need to find. Combining Corollary 7 with Theorem 8, we can conclude that for any path $p \in maxZDPto(v, i)$ for any $(v, i) \in H(G)$, we can find a ZR walk w by searching from some $(v_0, 1)$ with $v_0 \in V_0$ according to (4) such that $p \in ZDPof(w)$, and therefore $t(w) = t(p)$. If we find all such walks, which are defined later as *critical walks*, we get the IP.

Definition 9. Let G be a valid SDFG and path $p \in H(G)$, with

$$p : (v_0, l_0) \to (v_1, l_1) \to \cdots \to (v_n, l_n).$$

Path p is a *critical path* if it satisfies the following conditions:

- $v_0 \in V_0$, as defined above;
- $l_0 = 1$ and for each $i \in [1, n]$, l_i is defined as in (4), i.e., l_i is the first firing of v_i that depends on firing l_{i-1} of v_{i-1}; and
- for each $e \in OutE(v_n)$, there is an $l_{n+1} > q(snk(e))$, where l_{n+1} is also defined as in (4), i.e., there are no firings of actors connected to v_n that depend on firing l_n of v_n in the same iteration.

Corollary 7 and Theorem 8 guarantee that a critical path of $H(G)$ can be found on G without an explicit conversion to $H(G)$. From now on, the label l_i will be used uniquely for critical paths.

Definition 10. A ZR walk that has a corresponding path that is a critical path is a *critical walk*.

We call (v, i) an *element* of a walk w if $v \in w$. It is in fact an actor in a path of the corresponding HSDFG. If (v, i) is an actor in a *critical* path p of the HSDFG, corresponding to a walk w, we call (v, i) a *critical element* in p, or in w. Elements that are not actors in any critical paths are *noncritical* elements. An element may belong to two critical walks and may be critical in one but not critical in another. For example, walks W_1 and W_2 are critical walks of Fig. 4. Actor $(B, 4)$ is an element of both walks. $(B, 4)$ is critical in W_1 but is not critical in W_2.

B. Computation Times of Critical Walks

We use a vector $criT$, with the same size as T, to contain the computation times of critical walks and their subwalks. Let $criW(v, l_i)$ represent the set of the critical walks of G that include (v, l_i) as a critical element. We denote a prefix subwalk of a walk w to v_i as $w|_{v_i}$ or $w|_{(v_i,l_i)} : v_0 \to v_1 \to \cdots \to v_i$, where (v_i, l_i) is critical in w. Then $criT$ is defined as follows.

$$criT(v, i) = \begin{cases} -1, & (v, i) \text{ is noncritical} \\ \max_{w \in criW(v,i)} t(w|_{(v,i)}), & (v, i) \text{ is critical.} \end{cases}$$

By Theorem 8-2, for any path $p \in maxZDPto(v, i)$, we can find a critical walk w that goes through v such that (v, l_j) is a critical element of w and $p \in ZDPof(w|_{(v,l_j)})$. So we have:

$$IP(G) = \max_{v \in G, i \in [1, q(v)]} criT(v, i).$$

Algorithm 1 sdfIP(G)

Input: A valid SDFG $G = \langle V, E, t, d, prd, cns \rangle$
Output: $criT$ and IP

```
1: $V_0 = \{v \in V : \forall e \in InE(v), d(e) \geq cns(e)\}$
2: $\forall v \in G, i \in [1, q(v)]$, let $criT(v, i) = -1$
3: for all $v \in V_0$ do
4:     $criT(v, 1) = t(v)$
5:     $getNextT(v, 1)$
6: end for
7: $IP = \max_{v \in V, i \in [1,q(v)]} criT(v, i)$
8: return  $IP$ and $criT$

   getNextT($u, l$)

9: for all $e \in OutE(u)$ do
10:    $l' = \lfloor \frac{(l-1)prd(e)+d(e)}{cns(e)} \rfloor + 1$ //  Equation (4)
11:    $uNext = snk(e)$
12:    if $l' \leq q(uNext)$ then
13:        if $criT(uNext, l') < criT(u, l) + t(uNext)$ then
14:            $criT(uNext, l') = criT(u, l) + t(uNext)$
15:            $getNextT(uNext, l')$
16:        end if
17:    end if
18: end for
```

Algorithm 1 computes $criT$ and the IP. It is a variation of a *depth-first search* (DFS) algorithm. It can also be implemented by a breadth-first search strategy. After initializing V_0 and $criT$, Algorithm 1 begins its search from each $(v, 1)$ with $v \in V_0$, using the recursive procedure $getNextT$ as a subroutine to explore critical walks and to compute the $criT$. In $getNextT$, line 10 is according to (4); lines 12- 17 show a DFS strategy for each critical walk; lines 13-14 guarantee that if some element $(uNext, l')$ belongs to more than one critical walk, $criT(uNext, l')$ holds the largest computation time.

C. Relationship between criT and T

When we calculate the retiming function in the next section, we need to count the numbers of (v, i)s with $T(v, i) > dip$ for each v, denoted as $nT_{dip}(v)$. For example, in Fig. 4, let $dip = 5$; then there are three copies of B, $(B, 3)$, $(B, 4)$ and $(B, 5)$, with $T(B, i) > 5$. So $nT_5(B) = 3$. We show below that we can get $nT_{dip}(v)$ directly from $criT$, although $criT$ includes only a part of T.

By the definitions of $criT$ and T, for each (v, i), we have $criT(v, i) \leq T(v, i)$. It is easy to see that if $criT(v, i) > dip$, then $T(v, i) > dip$.

Let $lCriT_{dip}(v) \in [1, q(v) + 1]$ be the lowest label i that has $criT(v, i) > dip$; if there is no (v, i) with $criT(v, i) > dip$ then $lCriT_{dip}(v) = q(v) + 1$. For example, in Fig. 4, let $dip = 5$; 3 is the lowest label with $criT(B, i) > 5$ and $lCriT_5(B) = 3$.

For each (v, j) with $j < lCriT_{dip}(v)$, $criT(v, j) \leq dip$. The longest maximal zero-delay path p to (v, j), which leads to $t(p) = T(v, j)$, corresponds to a subwalk $w|_{(v,l')}$ of a critical walk w that includes (v, l') with $l' \leq j$. Since $l' < lCriT_{dip}(v)$, $t(w|_{(v,l')}) = criT(v, l') \leq dip$. Therefore $T(v, j) \leq dip$. Then $lCriT_{dip}(v)$ is exactly the lowest label i that has $T(v, i) > dip$. Then $nT_{dip}(v)$ can be calculated via $lCriT_{dip}(v)$ according to the following equation.

$$nT_{dip}(v) = q(v) - lCriT_{dip}(v) + 1. \quad (5)$$

V. Feasible Retiming

This section presents our method for feasible retiming. Firstly, we show that our method, sdfFEAS, mimics the steps of the FEAS algorithm in [5], which is proven to be sufficient and necessary for feasible retiming of HSDFGs. An important observation is that FEAS is not directly applicable to SDFGs. It can only be applied after converting an SDFG to its equivalent HSDFG. Our approach works directly on SDFGs. It mimics the steps of FEAS but does not explicitly construct the HSDFG. Instead, it implicitly explores the relevant (critical) paths in the underlying HSDFG. Mimicking the steps of FEAS forms the basis of our correctness argument. Avoiding the explicit construction of the HSDFG is crucial for the efficiency of our approach. Secondly, in the final subsection of this section we provide a sharp termination condition. This further improves the efficiency of our approach.

A. FEAS Algorithm

FEAS, with a little adjustment, is shown in Algorithm 2 in terms of the notations defined in this paper. Procedure CP [5] is used to compute the T and IP for an HSDFG. The order of the lines invoking CP and checking whether $IP(G_r) \leq dip$ in Algorithm 2 is different from the order in FEAS in [5]. This doesn't affect the correctness of Algorithm 2 but speeds it up when a feasible retiming exists. Algorithm 2 works by relaxation. After initializing the retiming vector as the zero vector and an auxiliary variable G_r as G_h, it computes the T and IP of the original graph. If $IP(G_h) \leq dip$ already holds, then no further retiming is needed. Otherwise, at each iteration of the outer loop (lines 7-18), it increases each entry $r(v)$ of the retiming vector by 1 when $T(v)$ is larger than dip (the

Algorithm 2 FEAS(G_h, dip) [5]

```
Input: A valid HSDFG $G_h = \langle V, E, t, d \rangle$ and a nonnegative integer $dip$
Output: A retiming $r$ of $G_h$ such that $r(G_h)$ is a valid SDFG with $IP(r(G_h)) \leq dip$, if such a retiming exists
1: $\forall v \in V$, let $r(v) = 0$
2: $G_r = G_h$
3: get $T$ and $IP$ of $G_r$ from $CP(G_r)$
4: if $IP \leq dip$ then
5:     return $r$
6: end if
7: for $i = 1$ to $|V| - 1$ do
8:     for all $v \in V$ do
9:         if $(T(v) > dip)$ then
10:            $r(v) = r(v) + 1$
11:        end if
12:    end for
13:    $G_r = r(G_r)$;
14:    get $T$ and $IP$ of $G_r$ from $CP(G_r)$
15:    if $IP \leq dip$ then
16:        return $r$
17:    end if
18: end for
19: return false
```

Algorithm 3 sdfFEAS(G, dip)

```
Input: A valid SDFG $G = \langle V, E, t, d, prd, cns \rangle$ and a nonnegative integer $dip$
Output: A retiming $r$ of $G$ such that $r(G)$ is a valid SDFG with $IP(r(G)) \leq dip$, if such a retiming exists
1: $\forall v \in V$, let $r(v) = 0$
2: $G_r = G$
3: get $criT$ and $IP$ of $G_r$ from $sdfIP(G_r)$
4: if $IP \leq dip$ then
5:     return $r$
6: end if
7: for $i = 1$ to $\sum_{v \in V} q(v) - 1$ do
8:     for all $v \in V$ do
9:         get $lCriT_{dip}(v)$ from $criT$
10:        $nT_{dip}(v) = q(v) - lCriT_{dip}(v) + 1$ // Equation (5)
11:        $r(v) = r(v) + nT_{dip}(v)$
12:    end for
13:    $G_r = r(G)$
14:    get $criT$ and $IP$ of $G_r$ from $sdfIP(G_r)$
15:    if $IP \leq dip$ then
16:        return $r$
17:    end if
18: end for
19: return false
```

inner loop), trying to shorten those zero-delay paths, to form a *retiming step*; then it transforms the current HSDFG to a new graph according to this retiming step (line 13) and compute the T and IP for the new graph to check whether its IP is not larger than dip; if not, then the accumulation of the previous retiming steps forms a feasible retiming, otherwise, the algorithm goes on to the next iteration until the bound of the outer loop ($|V|-1$, where $|V|$ denotes the number of actors in V) is reached; if at that moment a G_r with $IP(G_r) \leq dip$ yet has not been found then it concludes that dip is not feasible for G_h.

The largest cycle in a graph includes at most $|V|$ edges and each retiming step increases entry $r(v)$ at most 1, so the worst case is that $|V|-1$ retiming steps cause delays to travel through the largest cycle and yet no feasible retiming is found. Any more retiming steps may only lead to a delay distribution that has already been tested. This bound (line 7) guarantees that all the delay distributions that may lead to different IPs are tested. We show later that this fixed bound in fact gives some redundant iterations except for the worst case.

B. sdfFEAS Algorithm

Converting an SDFG to its equivalent HSDFG first and then using Algorithm 2 can get a correct output when retiming an SDFG. This is the traditional method, which is called byHSDF. However, to the best of our knowledge, there are no proved sufficient and necessary algorithms in the literature that do not need to convert SDFGs to HSDFGs for finding a feasible retiming of an SDFG. Our feasible retiming algorithm, *sdfFEAS*, shown in Algorithm 3, is a sufficient and necessary solution working directly on SDFGs.

The structure of *sdfFEAS* is similar to that of *FEAS*. Only the inner loop (lines 8-12) of *sdfFEAS*, in which the retiming step is computed, differs. The retiming step may be assigned a positive integer larger than 1, whereas in *FEAS* the retiming step is always 1. If we consider these two algorithms on the same graph, say, $sdfFEAS(G, dip)$ and $FEAS(H(G), dip)$, however, since one actor in G may have more than one copy in $H(G)$, the retiming steps in these two algorithms may be in fact equal. As we show below, $sdfFEAS(G, dip)$ does give the same retiming as $FEAS(H(G), dip)$.

C. Equivalence of FEAS and sdfFEAS

It is obvious that the initial operations, lines 1-6, of Algorithm 2 are equivalent to those of Algorithm 3, lines 1-6; the bound of the outer loop in line 7 of Algorithm 2 is the same as the one in line 7 of Algorithm 3. It remains to be shown that iterations of the outer loop of these two algorithms are equivalent.

We formally decompose $FEAS(H(G), dip)$ into two sequences of retimings and a sequence of HSDFGs. After the i^{th} iteration ($i > 0$) of the outer loop, the retiming step is $\triangle r'_i$ and the retiming obtained is $r'_i = \sum_{j=1}^{i} \triangle r'_j$; the HSDFG sequence generated during the entire procedure is: $h_0, h_1, ..., h_N$, where $h_0 = H(G)$, and $h_{i+1} = \triangle r'_{i+1}(h_i)$ for $0 \leq i < N$.

Before the first iteration, all entries of the retiming vector are initialized to zero, so $r'_0 = \vec{0}$. According to the proof of Algorithm FEAS in [5], each $\triangle r'_i$ is legal and hence each r'_i is legal and each h_i is valid; if there are feasible retimings, then $IP(h_N) \leq dip$ and $IP(h_i) > dip$ for all $0 \leq i < N$; if not, $IP(h_i) > dip$ for all $0 \leq i \leq N$ and $N = |V(G_h)| - 1$. Fig. 5(a), for example, gives the sequences $\triangle r'_i$, r'_i and h_i for $FEAS(H(G_1), 4)$. At each iteration, the retiming step $\triangle r'_i$ is determined by the T in Fig. 5(b).

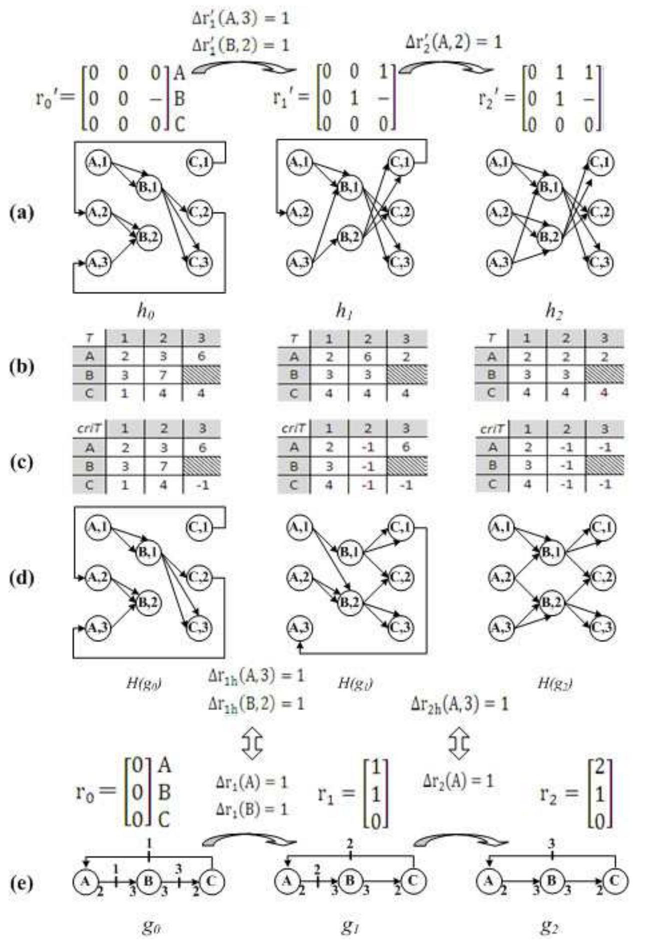


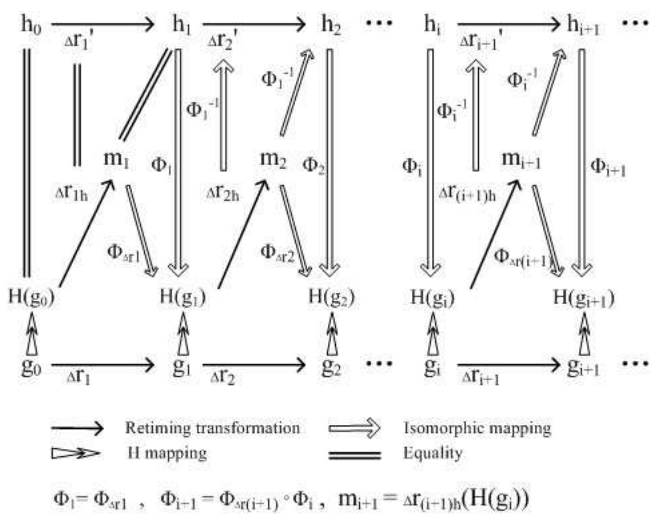
Fig. 5. Edges with delays in HSDFGs are omited. (a) The procedure of *FEAS*($H(G_1)$, 4); (b) *T* changes with *FEAS*($H(G_1)$, 4); (c) *criT* changes with *sdfFEAS*(G_1, 4); (d) $H(r(G_1))$ changes with *sdfFEAS*(G_1, 4); (e) the procedure of *sdfFEAS*(G_1, 4).

The retiming step for each iteration of *FEAS* sometimes seems unlikely to be legal. For example, in h_0 of Fig. 5(a), $\triangle r'_1(A, 3) = 1$ seems to lead to an illegal retiming because there are no delays on the outgoing edge of $(A, 3)$. This situation, however, implies the existence of a longer zero-delay path including the actor next to $(A, 3)$, $(B, 2)$, leading to $\triangle r'_1(B, 2) = 1$. And $(B, 2)$ has sufficient delays on its outgoing edge for one reversed firing. After $(B, 2)$ is retimed once, $(A, 3)$ gets enough delays for one reversed firing. Therefore, legality of $\triangle r'_1$ is guaranteed.

We also decompose the procedure of *sdfFEAS*(G, *dip*): after the i^{th} iteration of the outer loop, the retiming step is $\triangle r_i$ and the retiming obtained is $r_i = \sum_{j=1}^{i} \triangle r_j$; the SDFG sequence generated during the procedure is: $g_0, g_1, ..., g_M$, where $g_0 = G$, and $g_{i+1} = \triangle r_{i+1}(g_i)$ for $0 \le i < M$.

Before the first iteration, all entries of the retiming vector are initialized to zero, so $r_0 = \vec{0}$. Fig. 5(e), for example, gives the sequences $\triangle r_i$, r_i and g_i for *sdfFEAS*(G_1, 4). At each iteration, the retiming step $\triangle r_i$ is determined by the *criT* in Fig. 5(c).

In the next part, we show that each $\triangle r_i$ is legal and hence r_i is legal; that each $H(g_i)$ is isomorphic to h_i and hence $IP(g_i) = IP(h_i)$; and that $M = N$. Then we can conclude that the output of *sdfFEAS*(G, *dip*) is exactly the output of *FEAS*($H(G)$, *dip*),



Retiming transformation
H mapping
Isomorphic mapping
Equality
$\Phi_1 = \Phi_{\Delta r1}$, $\Phi_{i+1} = \Phi_{\Delta r(i+1)} \circ \Phi_i$, $m_{i+1} = \Delta r_{(i+1)h}(H(g_i))$

Fig. 6. The proof framework of the equivalence of *FEAS*($H(G)$, *dip*) and *sdfFEAS*(G, *dip*)

i.e., *sdfFEAS*(G, *dip*) and *FEAS*($H(G)$, *dip*) are equivalent.

We need to prove that each $\triangle r_i$ has the same effect on g_{i-1} as $\triangle r'_i$ on h_{i-1}. Since $\triangle r_i$ and $\triangle r'_i$ are defined on different graph structures, we cannot compare them directly. However, for each legal r of G there always is a corresponding r_h of $H(G)$, because the data dependencies of G can always be represented by $H(G)$; r changes them by redistributing delays of G. Therefore, when we say a retiming r' on $H(G)$ is equivalent to a retiming r on G, denoted as $r' \equiv r$, this means that $r' = r_h$ under some isomorphism we define later. For equivalent r and r', if r' is legal then r is legal, because they change the delay distribution identically.

First, let's see how r_h is defined. A retiming step $\triangle r(v)$ is in fact the result of the last copies of v in $H(G)$ that have $T(v, i) > dip$, according to *sdfFEAS*(G, *dip*). For example, in Fig. 5(e), $\triangle r_1(A) = 1$ and $\triangle r_1(B) = 1$ for g_0. This is caused by $criT(A, 3) > 4$ and $criT(B, 2) > 4$ (Fig. 5(c)). So the paths in $H(g_0)$ (Fig. 5(d)) that need to be shortened are those that reach $(A, 3)$ and $(B, 2)$. So the effect of $\triangle r_1(A) = 1$ and $\triangle r_1(B) = 1$ on G is exactly the effect of $\triangle r_{1h}(A, 3) = 1$ and $\triangle r_{1h}(B, 2) = 1$ on $H(G)$. We generalize the above discussion in Definition 11.

Definition 11. Given a valid SDFG G and its legal retiming r. r's corresponding retiming on $H(G)$, r_h, is defined as follows: for each $(v, i) \in H(G)$,

$$r_h(v, i) = \begin{cases} 0, & i \le q(v) - r(v) \\ 1, & i > q(v) - r(v). \end{cases}$$

Before the first iteration, $h_0 = H(g_0) = H(G)$, and $r_{0h} = \vec{0}$; therefore r_0 is legal. We show below that after each iteration i, $\triangle r'_i \equiv \triangle r_i$ holds and h_i is isomorphic to $H(g_i)$, denoted by $h_i \cong H(g_i)$. We illustrate the reasoning with the example in Fig. 5 and the proof framework outlined in Fig. 6.

At the first iteration, *sdfFEAS* works on g_0 to generate the first retiming step $\triangle r_1$, corresponding to $\triangle r_{1h}$ of $H(g_0)$. According to lines 8-12 of Algorithm 3, $\triangle r_1(v) = nT_{dip}(v)$, making $\triangle r_{1h}(v, i) = 1$ for $i > q(v) - nT_{dip}(v)$ and $\triangle r_{1h}(v, i) = 0$ for other i according to Definition 11.

FEAS works on h_0 ($= H(g_0)$) to generate the first retiming step $\triangle r'_1$. According to (2) and lines 8-12 of Algorithm 2, we have $\triangle r'_1(v,i) = 1$ for $i > q(v) - nT_{dip}(v)$ and $\triangle r'_1(v,i) = 0$ for other i.

So we have $\triangle r_{1h} = \triangle r'_1$ and hence $\triangle r'_1 \equiv \triangle r_1$. Carrying out the retiming transformation $\triangle r_1$ on g_0 and $\triangle r'_1$ on h_0, we get g_1 and h_1, respectively. That is, $g_1 = \triangle r_1(g_0)$ and $h_1 = \triangle r'_1(h_0)$.

We need to show that $h_1 \cong H(g_1)$. Because of $\triangle r_{1h} = \triangle r'_1$, we know $\triangle r_{1h}(H(g_0)) = \triangle r'_1(h_0) = h_1$. If $\triangle r_{1h}(H(g_0)) = H(\triangle r_1(g_0))$ then $h_1 = H(g_1)$. But this is not the case. For example, in Fig. 5, $(A,2)$ in $\triangle r_{1h}(H(g_0))$ ($= h_1$) has an incoming edge with zero delays while $(A,2)$ in $H(\triangle r_1(g_0))$ ($= H(g_1)$) has not; and a similar situation occurs for $(A,3)$.

However, examining $\triangle r_{1h}(H(g_0))$ and $\triangle H(r_1(g_0))$ closely, we find that the numbers of delays changed on edges between each actor pair $((u,i),(v,j))$ are the same because $\triangle r'_1 \equiv \triangle r_1$; disregarding labels of actor names, the graphs have the same structure. For example, rewriting $(A,1)$ to $(A,2)$, $(A,2)$ to $(A,3)$, $(A,3)$ to $(A,1)$, $(B,2)$ to $(B,1)$ and $(B,1)$ to $(B,2)$ of h_1 ($= \triangle r_{1h}(H(g_0))$) in Fig. 5(a), we get graph $H(g_1)$ ($= H(\triangle r_1(g_0))$) in Fig. 5(d).

Because a retiming r on G always increases the number of delays on the incoming edges of some later (v,i)s in $H(G)$, and because the mapping H always arranges actors with zero-delay incoming edges to the later (v,i)s, we can generalize the above-mentioned rewriting as a function $\Phi_r : r_h(H(G)) \to H(r(G))$, mapping each (v,i) to (v,j) in terms of

$$j = \begin{cases} i + r(v), & i \leq q(v) - r(v) \\ i + r(v) - q(v), & i > q(v) - r(v). \end{cases}$$

Φ_r is an isomorphism and doesn't change the IP because it only rewrites the names, in fact the labels, of actors of an HSDFG.

Lemma 12. Given a valid SDFG G and its legal retiming r. $r_h(H(G)) \cong H(r(G))$ under Φ_r.

Since $\Phi_{\triangle r_1}(\triangle r_{1h}(H(g_0))) = H(\triangle r_1(g_0)) = H(g_1)$ and $\triangle r_{1h}(H(g_0)) = h_1$, we have $h_1 \cong H(g_1)$ under $\Phi_{\triangle r_1}$.

At the second iteration, consider *sdfFEAS* working on g_1 to get $\triangle r_2$ and *FEAS* working on $H(g_1)$ (not h_1) to get $\triangle r''_2$; repeating the previous discussion, we have $\triangle r''_2 = \triangle r_{2h}$. In Fig. 5, for example, $\triangle r''_2(A,3) = \triangle r_{2h}(A,3) = 1$. Since $h_1 \cong H(g_1)$ under $\Phi_{\triangle r_1}$, using the inverse mapping of $\Phi_{\triangle r_1}$, $\Phi^{-1}_{\triangle r_1}$, on $\triangle r''_2$, we get exactly the retiming step $\triangle r'_2$ computed by *FEAS* on h_1. Using an inverse rewriting of the one we used for h_1 in Fig. 5(a), $\triangle r_{2h}$ is transformed to $\triangle r'_2$, for example. An inverse of an isomorphism is also an isomorphism, so we have $\triangle r_2 \equiv \triangle r'_2$ under $\Phi^{-1}_{\triangle r_1}$. Therefore the effect of $\triangle r'_2$ on h_1 is the same as the effect of $\triangle r_2$ on g_1 under $\Phi^{-1}_{\triangle r_1}$. That is,

$$h_2 = \Phi^{-1}_{\triangle r_1}(\triangle r_{2h}(H(g_1))).$$

As explained at the first iteration, according to Lemma 12, there is

$$\Phi_{\triangle r_2}(\triangle r_{2h}(H(g_1))) = H(\triangle r_2(g_1)) = H(g_2).$$

Then, we have $h_2 \cong H(g_2)$ under $\Phi_{\triangle r_2} \circ \Phi_{\triangle r_1}$, which is the composition of two isomorphisms and therefore is also an isomorphism.

The discussion repeats till *sdfFEAS* and *FEAS* reach an i^{th} iteration such that $IP(g_i) = IP(h_i) \leq dip$, if there are feasible retimings for G with *dip*; alternatively they pass through $\sum_{v \in V} q(v) - 1$ iterations and find no feasible retiming. In both cases, the numbers of iterations are the same, that is, $M = N$.

In summary, see Fig. 6, by means of one auxiliary sequence of retimings of $H(g_i)$, $\triangle r_{ih}$, and one auxiliary HSDFG sequence, $m_i = \triangle r_{ih}(H(g_{i-1}))$, which is isomorphic to both $H(g_i)$ and h_i, we prove that $H(g_i) \cong h_i$.

According to the above discussion, we have the following theorem.

Theorem 13. Given a valid SDFG G with *dip*, *sdfFEAS*(G, dip) is equivalent to *FEAS*$(H(G), dip)$.

We may conclude that *sdfFEAS* is a sufficient and necessary algorithm for the feasible retiming problem on SDFGs as *FEAS* is for HSDFGs.

D. *sdfFEAS with Sharp Termination Condition*

When *dip* is a feasible IP of G, *sdfFEAS*(G, dip) terminates at once when it finds a feasible retiming. As observed in [8], the IPs of the intermediate graphs generated during the procedure, the g_i (h_i) introduced above, converge rapidly. For example, *sdfFEAS*$(G_1, 4)$, shown in 5(e), terminates after only 2 iterations of the main loop.

When *dip* is not feasible for G, however, *sdfFEAS*(G, dip) has to do $\sum_{v \in G} q(v) - 1$ iterations to determine that there are no feasible retimings for this *dip*. For example, because 3 is not a feasible IP of G_1, *sdfFEAS*$(G_1, 3)$ will terminate after 7 iterations and report false. Examining the procedure of *sdfFEAS*$(G_1, 3)$, shown in Fig. 7, we find that after 5 iterations it reaches a graph that has been checked before. That is, the 6^{th} and 7^{th} iterations in fact repeat some previous iterations, the 3^{th} and 4^{th} iterations in this example.

Since a retiming only changes the delay distribution, $d(G)$, of a graph G, we can conclude that $g_i = g_j$ because $d(g_i) = d(g_j)$. Based on the above observation, we can improve *sdfFEAS* as follows.

1) Store each $d(g_i)$ into a list *DDs*;
2) When $IP(g_i) > dip$, check whether there is repetition in *DDs*, that is, there is a $j < i$ such that $d(g_j) = d(g_i)$;
3) If finding repetition in *DDs*, or when reaching the bound of the outer loop of Algorithm 3, terminate and report false; otherwise, continue as Algorithm 3 shows.

It is clear that this improved procedure is correct. The new termination condition finds repetition in *DDs* to stop the procedure as soon as possible. The new termination condition is sufficient to guarantee the procedure to stop before any redundant work is done, as we show below. That is, we do not need check the bound $\sum_{v \in V} q(v) - 1$. The algorithm can simply be run until a delay distribution re-occurs.

Recall that we have proven that *sdfFEAS* is necessary (and sufficient) by showing its equivalence to *FEAS*. That means, when it reports that a *dip* is not feasible for G, all the possibilities have been exhausted, as we explain at the end of Section V.A. Then the graphs generated during the procedure, g_i, $i = 0, ..., M$ with $M = \sum_{v \in G}(q(v)) - 1$, cover all the possibilities to be checked. This leads to two possible situations: there exist i and j with $i < j \leq M$ such that $g_j = g_i$

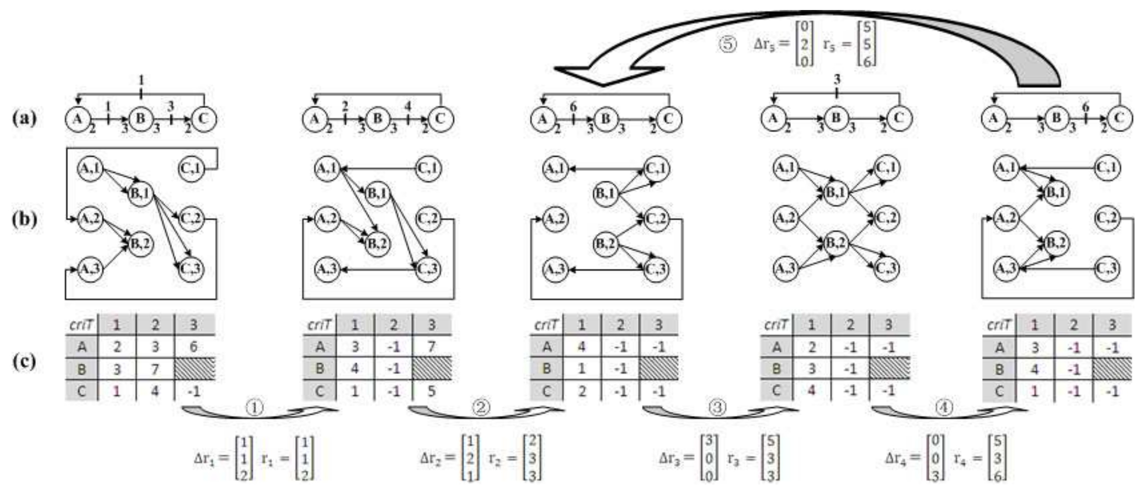


Fig. 7. Edges with delays in HSDFGs are omited. (a) The procedure of *sdfFEAS*$(G_1, 3)$; (b) $H(g_i)$ changes with *sdfFEAS*$(G_1, 3)$; (c) *criT* changes with *sdfFEAS*$(G_1, 3)$.

and therefore $g_{j+x} = g_{i+x}$ for $x \le M - j$; otherwise, there must be some $g_i = g_{M+1}$ with $i < M$ and $g_{M+1} = \triangle r_{M+1}(g_M)$, because if not, g_{M+1} is a graph that hasn't been checked by *sdfFEAS*, and there may be an $IP(g_{M+1}) \le dip$. This would mean that *sdfFEAS* is not necessary and cause a contradiction.

For the former situation, x redundant iterations are removed by the improved procedure; for the latter situation, the number of the iterations of the improved procedure is the same as that of *sdfFEAS*. Therefore the new termination condition, checking only the re-occurrence of the delay distribution after retiming, makes sure that only the necessary iterations are performed, improving the bottleneck of the feasible retiming problem [8]. It is in fact a sharp condition.

From now on, *sdfFEAS* represents the improved procedure. For example, the *DDs* of *sdfFEAS*$(G_1, 3)$ is as follows:

$$\begin{array}{c} \langle A,B\rangle \\ \langle B,C\rangle \\ \langle C,A\rangle \end{array} \begin{pmatrix} 1 \\ 3 \\ 1 \end{pmatrix} \mapsto \begin{pmatrix} 2 \\ 4 \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} \mathbf{6} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 0 \\ 3 \end{pmatrix} \mapsto \begin{pmatrix} 0 \\ 6 \\ 0 \end{pmatrix} \mapsto \begin{pmatrix} \mathbf{6} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}$$

When the second $(6, 0, 0)$ is reached, there is repetition in the list *DDs*, and *sdfFEAS*$(G_1, 3)$ terminates.

VI. Optimal Retiming

According to the definition of optimal retiming, when we say that r is an optimal retiming of G and *optIP* is the optimal iteration period of G, this means that r is a feasible retiming for G with $dip = optIP$, and that there is no feasible retiming for G with $dip = optIP - 1$. The typical procedure of optimal retiming is to find some potential *optIP*s as *dip* and test whether they are feasible until we find a smallest one.

We denote the IP of the original SDFG G as initial IP (initIP) and the maximal execution time of actors of G as *maxt*. It is obvious that no matter what retiming is used, G will not be transformed to a new graph with IP lower than its *maxt*. Therefore the range of potential *optIP*s for G is the integer interval $[maxt, initIP]$.

Recall from the previous section that dealing with an input for which there is no feasible retiming is usually a bottleneck to the solution for the feasible retiming problem. Although our method provides a sharp bound for this situation, the number of iterations when no feasible retiming exists seems to be more than the number of iterations when one exists; see Fig. 5 and Fig. 7 for example. Although a binary search has a lower theoretical complexity than a linear search, a binary search method for optimal retiming tends to test more than one potential *optIP* that is not feasible, while a linear search method, searching potential *optIP*s decreasingly, only tests one potential *optIP* that is not feasible. So, in line with the approach in [15], we use a linear search to find the optimal retiming, called *sdfOPT*, shown in Algorithm 4.

Algorithm 4 sdfOPT(G)

```
Input: A valid SDFG $G = \langle V, E, t, d, prd, cns\rangle$
Output: A retiming $r$ of $G$ such that $IP(r(G))$ is as small as possible
1:  $\forall v \in V$, let $r(v) = 0$;
2:  $G_r = G$;
3:  get $IP$ of $G_r$ from $sdfIP(G_r)$;
4:  $optIP = IP$
5:  while $optIP > \max_{v \in V} t(v)$ do
6:      run Algorithm sdfFEAS($G_r, optIP - 1$) to determine whether a feasible retiming exists
7:      if a feasible retiming $r'$ exists then
8:          $G_r = r'(G_r)$;
9:          $r += r'$;
10:         get $IP$ of $G_r$ from $sdfIP(G_r)$;
11:         $optIP = IP$
12:     else
13:         return $r$
14:     end if
15: end while
16: return $r$
```

Since *initIP* is a feasible IP, *sdfOPT* begins with $dip = initIP - 1$, using *sdfFEAS* to check whether a feasible retiming exists for G with this *dip*. If so, it stores this retiming and computes the new IP of the retimed graph. Then it decreases this IP by 1 and calls *sdfFEAS* on the retimed graph with the new *dip*. If no feasible retiming exists then the previous one is an optimal retiming and the previous IP is the *optIP*; if a feasible retiming is found, then the procedure repeats until no feasible retiming is found for some *dip*.

The termination condition of *sdfOPT* is sharp because of the fact that the termination condition of *sdfFEAS* is sharp.

VII. Experimental evaluation

We implemented our methods and related methods in SDF3 [16]. For feasible retiming, the methods implemented include the traditional method—*byHSDF*, Algorithm 3 in [3]—*O'Neil01*, the algorithm in [14]—*Zhu10*, and our new method— *sdfFEAS*. For optimal retiming, the methods include Algorithm SDF_Retiming_Improved in [15] — *Liveris07*, and our new method — *sdfOPT* (both a depth-first variant and a breadth-first variant — sdfOPT-BFS).

To evaluate the efficiency of the various algorithms, we performed experiments on two sets of SDFGs, running on a 2.4GHz CPU with 12MB cache. The experimental results of these two sets are shown in Tables I and II, respectively.

TABLE I
Execution time of practical DSP examples

Graph information				
name	Samplerate	Satellite	MaxES	CEer
$\|V\|$	7	23	14	23
$\|E\|$	7	29	15	44
$\|V'\|$	613	4,516	1,289	43
$\|E'\|$	1,328	9,456	2,342	99
Initial and optimal iteration period				
InitIP	21	11	11,528	53,652
OptIP	6	2	8,192	47,128
Feasible retiming for dip=optIP				
Execution time (ms)				
sdfFEAS	0.0	0.2	0.1	0.3
Zhu10	4.6	56.9	9.7	30.0
O'Neil01	661.0	N	N	2.9
byHSDF	212.2	11,875.1	665.7	2.0
Memory used (MB)				
sdfFEAS	0.23	0.72	0.38	0.84
Zhu10	0.23	0.70	0.38	0.84
O'Neil01	2.49	N	N	0.92
byHSDF	1.58	10.71	2.94	0.94
Optimal retiming				
Execution time (ms) / Retiming steps				
sdfOPT	0.1 / 8	1.3 / 28	0.2 / 12	4.8 / 44
Liveris07	4.1 / 762	51.3 / 2175	9.5 / 1035	1.5 / 44
sdfOPT-BFS	0.4 / 8	1.8 / 28	0.3 / 12	3.7 / 44
Memory used (MB)				
sdfOPT	0.23	0.72	0.38	0.85
Liveris07	0.23	0.71	0.38	0.84
sdfOPT-BFS	0.23	0.74	0.38	0.85

Both tables have four parts. The first part gives the number of actors and edges in an SDFG ($|V|$ and $|E|$) and the number of actors and edges in its equivalent HSDFG ($|V'|$ and $|E'|$); the second part lists the initial iteration period and optimal iteration period of each graph; the third part shows execution times of different feasible retiming methods, and for the practical DSP examples in Table I also memory usage; the fourth part shows the results for different optimal retiming methods. Execution times are measured in milliseconds (ms); memory usage is measured in Megabytes (MB).

The first set of SDFGs consists of four practical DSP applications, including a sample rate converter (Samplerate) [23], a satellite receiver (Satellite) [24], a maximum entropy spectrum analyzer (MaxES) [25], and a channel equalizer (CEer); the latter is converted from the cyclo-static dataflow model [26] in [27]. Adopting the method in [2], by introducing to each model a dummy actor with computation time zero and edges with proper rates and delays to connect the dummy actor to the actors that have no incoming edges or no outgoing edges, we convert these models to strongly connected graphs. The information in the first part of Table I takes into account the dummy actor and its incoming and outgoing edges.

TABLE II
Execution time of synthetic examples

Graph information			
$\|V\|$	19	46	95
$\|E\|$	21	52	108
$\|V'\|$	1,983	4,983	9,962
$\|E'\|$	4,055	10,126	19,920
Initial and optimal iteration period			
InitIP	69	122	193
OptIP	55	77	118
Execution time (ms)			
Feasible retiming for dip=optIP			
sdfFEAS	0.2	0.6	1.7
Zhu10	16.0	76.0	278.3
O'Neil01	1.3E+03	2.1E+04	8.6E+04
byHSDF	1.4E+03	9.5E+03	5.2E+04
Execution time (ms)			
Optimal retiming			
sdfOPT	0.6	3.3	11.9
Liveris07	8.6	42.7	158.8
sdfOPT-BFS	0.9	4.7	17.0

The second set consists of 150 synthetic strongly connected SDFGs generated by SDF3, mimicking real DSP applications and scaling up the models. The number of actors in an SDFG and the sum of the elements in the repetition vector, i.e. $|V'|$, have significant impact on the performance of the various methods, so we generated these examples according to three combinations of these two parameters— (20,2000), (50,5000) and (100,10000), respectively. The generated graphs, 50 in each group, deviate a little from these values. The explicit difference in size among the three groups shows how the performance of each method changes with the size of the graph. The information in the first part of Table II gives averages for each group.

We evaluate the feasible retiming methods on each graph with *dip* equal to its optimal iteration period. Except for

O'Neil01, all other algorithms find feasible retimings for all examples. Our method sdfFEAS completes in no more than 2ms for all examples, dramatically more efficient than O'Neil01 and byHSDF and about 100 times or more faster than Zhu10, especially for those SDFGs whose equivalent HSDFGs have a large size, e.g. Satellite in Table I. To our surprise, when the graphs scale up, the traditional method, byHSDF, may have better performance than O'Neil01, as the second and third columns of Table II show. The reason may be that O'Neil01 needs to convert an SDFG to an HSDFG many times while byHSDF only needs to do this once.

Our optimal retiming method reduces execution time compared to Liveris07 on all examples except for CEer in Table I. For further analysis, Table I, under 'Execution time/Retiming steps', gives besides the execution time the total number of retiming steps in the complete optimization procedure. Except for CEer, all examples show a much smaller number of retiming steps for sdfOPT than for Liveris07.

Generally, the speed of a (feasible/optimal) retiming algorithm is affected by three aspects: (a) the procedure for computing the IP; (b) the procedure for computing retiming steps; and (c) the termination condition, which determines the number of retiming steps. (a) and (b) both execute once at each retiming step. In most cases, the better runtime of sdfOPT over Liveris07 mainly comes from (b) and, particularly, (c). Our procedure for computing the IP is sometimes slower than that of Liveris07. In most cases, this slowdown is relieved by (b) and (c). When the advantage of (b) and (c) of our method is not large enough to relieve the slowdown by (a), sdfOPT is slower than Liveris07. This happens on the example CEer. In this case, sdfOPT performs 44 iterations (c) with a total execution time of 4.2ms for computing the IP (a) and 0.6ms for computing retiming steps (b); for Liveris07, these numbers are 44, 0.9ms, and 0.6ms, respectively. Although our IP procedure itself is slower, the equation derived from its output, Equation (5), is crucial for the speedup of (b) and (c).

Since our methods work directly on SDFGs and do not explicitly construct their equivalent HSDFGs, one may expect a lower memory usage when compared to traditional methods that convert to an HSDFG. We measured the memory used by the different methods on the DSP examples by the tool Valgrind (http://valgrind.org/) . Our feasible retiming methods (sdfFEAS and our earlier work Zhu10) use much less memory than O'Neil and byHSDF. The memory used by all optimal retiming algorithms is almost the same (because all methods work directly on SDFGs). The results are shown in Table I.

Although we assumed strongly connected graphs, our methods can also be used on SDFGs that are not strongly connected, only by enlarging the set V_0 to contain those actors that have no incoming edges and using the combined termination condition mentioned in Section V.D. We remove the dummy actors of the DSP examples so that they are no longer strongly connected and run different methods on them. It turns out that our method is still much faster, as shown in Table III. The execution times of Liveris07 are not shown in this table, because it is only applicable on strongly connected graphs.

All the feasible retiming algorithms mentioned in this paper are of exponential worst case complexity. For O'Neil01 and byHSDF, worst cases are caused by the procedure that converts an SDFG to its HSDFG, affected heavily by the size and sample rates of the graphs. For sdfFEAS, worst cases are caused by the procedure sdfIP, which may visit an actor an exponential number of times when there are complex nested cycles in the equivalent HSDFG, due to the depth-first search (DFS) strategy. For the same reason, sdfOPT is also of exponential worst case complexity. That is the reason why sdfOPT performs poorly on the example CEer.

TABLE III
EXECUTION TIME OF PRACTICAL DSP EXAMPLES (NOT STRONGLY CONNECTED)

	Execution time (ms)			
name	Samplerate	Satellite	MaxES	CEer
sdfFEAS	0.2	0.4	0.3	0.2
Zhu10	3.5	109.3	8.6	29.7
O'Neil01	371.8	94,102.5	N	3.1
byHSDF	156.5	11,186.7	495.1	2.0
sdfOPT	0.1	1.1	0.6	4.6

Procedure sdfIP can be modified to a breadth-first search (BFS) with $O(|V'|)$ complexity to give sdfFEAS an $O(|V'||E'|)$ complexity and sdfOPT an $O(K|V'||E'|)$ complexity, where $K = initIP - maxt$ and $|V'|$ and $|E'|$ are the numbers of actors and edges of the equivalent HSDFG, respectively. However, the average execution time of the BFS method is worse than the method we present in this paper, as shown in Tables I and II. In most cases, a DFS only needs to visit part of the actors of V'; sometimes it even only needs to traverse the actors of V, as the examples in previous sections show, while a BFS needs some overhead before computing $criT$. Another variant may be to compute the IP by the procedure used by Liveris07 in [15], whose worst case complexity is $O(|E'|)$. Whether its output can be used in sdfFEAS needs further investigation, and it may output incorrect results on SDFGs that are not strongly connected.

A similar situation occurs for the procedure sdfOPT. The execution times using a binary search strategy are for almost all considered graphs larger than when using a linear search strategy, as we use in Algorithm 4.

VIII. CONCLUSION

In this paper, we have presented new methods for finding a feasible retiming to optimize an SDFG to meet an iteration period constraint and for finding an optimal retiming so that an SDFG achieves its smallest iteration period. Both methods work directly on the SDFG without converting it to its equivalent HSDFG.

Our feasible retiming method, sdfFEAS, mimics the steps of Algorithm FEAS [5]. It is a sufficient and necessary solution for the feasible retiming problem. We provide a sharp termination condition to eliminate redundant work. Our optimal retiming method, sdfOPT, using sdfFEAS as a subroutine, also has a sharp termination condition. Experimental results show that our feasible method is four orders of magnitude faster than the method of [3] and two orders of magnitude faster than the method of [14]; our optimal method is more than ten times faster than the method of [15].

References

[1] E. A. Lee and D. G. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. on Computers*, vol. 36, no. 1, 1987.

[2] V. Zivojnovic, S. Ritz, and H. Meyr, “Optimizing DSP programs using the multirate retiming transformation,” in *Proc. of EUSIPCO Signal Processing*, 1994.

[3] T. O’Neil and E. H. M. Sha, “Retiming synchronous data-flow graphs to reduce execution time,” *IEEE Trans. on Signal Processing*, vol. 49, no. 10, pp. 2397–2407, 2001.

[4] K. K. Parhi, *VLSI digital signal processing systems: design and implementation*. Wiley India Pvt. Ltd., 2007.

[5] C. E. Leiserson and J. B. Saxe, “Retiming synchronous circuitry,” *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.

[6] V. Zivojnovic, S. Ritz, and H. Meyr, “Retiming of DSP programs for optimum vectorization,” in *Proc. of the 1994 Acoustics, Speech, and Signal Processing Conf.*. IEEE, 1994.

[7] Z. Jeddi and E. Amini, “Power optimization of sequential circuits by retiming and rewiring,” in *Proc. of 49th Midwest Symposium on Circuits and Systems, MWSCAS’06.*, vol. 2. IEEE, 2007, pp. 585–589.

[8] N. Shenoy and R. Rudell, “Efficient implementation of retiming,” in *Proc. of the 1994 IEEE/ACM int. conf. on Computer-Aided Design*. IEEE, 1994, p. 233.

[9] G. Even, I. Y. Spillinger, and L. Stok, “Retiming revisited and reversed,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 3, pp. 348–357, 1996.

[10] S. Simon, E. Bernard, M. Sauer, and J. Nossek, “A new retiming algorithm for circuit design,” in *Proc. of IEEE Int. Symposium on Circuits and Systems, ISCAS’94.*, pp. 35–38.

[11] N. Shenoy, “Retiming: theory and practice,” *Integration, the VLSI journal*, vol. 22, no. 1-2, pp. 1–21, 1997.

[12] V. Zivojnovic and R. Schoenen, “On retiming of multirate DSP algorithms,” in *Proc. of the Acoustics, Speech, and Signal Processing Conf.* IEEE Computer Society, 1996, pp. 3310–3313.

[13] E. A. Lee, “A coupled hardware and software architecture for programmable digital signal processors,” Ph.D. dissertation, University of California, Berkeley, 1986.

[14] X. Y. Zhu, “Retiming multi-rate DSP algorithms to meet real-time requirement,” in *Proc. of the 13nd Design, Automation and Test in Europe*. IEEE, 2010, pp. 1785–1790.

[15] N. Liveris, C. Lin, J. Wang, H. Zhou, and P. Banerjee, “Retiming for synchronous data flow graphs,” in *Proc. of the 2007 Asia and South Pacific Design Automation Conf.* IEEE, 2007, pp. 480–485.

[16] S. Stuijk, M. Geilen, and T. Basten, “SDF3: SDF For Free,” in *Proc. of the 6th Int. Conf. on Application of Concurrency to System Design*. IEEE, 2006, pp. 276–278. http://www.es.ele.tue.nl/sdf3/.

[17] S. Sriram and S. S. Bhattacharyya, *Embedded multiprocessors: scheduling and synchronization*. CRC Press, 2009.

[18] G. Sih, “Multiprocessor scheduling to account for interprocessor communication,” Ph.D. dissertation, University of California, Berkeley, 1991.

[19] R. M. Karp and R. E. Miller, “Properties of a model for parallel computations: Determinancy, termination, queueing,” *SIAM Journal on Applied Mathematics*, pp. 1390–1411, 1966.

[20] L. F. Chao and E. H. M. Sha, “Scheduling data-flow graphs via retiming and unfolding,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 1259–1267, 1997.

[21] E. Lee and S. Ha, “Scheduling strategies for multiprocessor real-time DSP,” in *IEEE Global Telecommunications Conf. and Exhibition, GLOBECOM’89. Communications Technology for the 1990s and Beyond.*, 1989, pp. 1279–1283.

[22] O. Moreira and M. Bekooij, “Self-timed scheduling analysis for real-time applications,” *EURASIP Journal on Advances in Signal Processing*, vol. 2007, p. 14, 2007.

[23] P. K. Murthy, S. S. Bhattacharyya, and E. A. Lee, “Joint minimization of code and data for synchronous dataflow programs,” *Formal Methods in System Design*, vol. 11, no. 1, pp. 41–70, 1997.

[24] S. Ritz, M. Willems, and H. Meyr, “Scheduling for optimum data memory compaction in block diagram oriented software synthesis,” in *Proc. of the 1995 Acoustics, Speech, and Signal Processing Conf.* IEEE, 1995, pp. 2651–2654.

[25] [Online]. Available: http://ptolemy.eecs.berkeley.edu/

[26] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, “Cyclo-static dataflow,” *IEEE Trans. on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.

[27] A. Moonen, M. Bekooij, R. van den Berg, and J. van Meerbergen, “Practical and accurate throughput analysis with the cyclo static dataflow model,” in *Proc. of the 15th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 2007, pp. 238–245.

Xue-Yang Zhu received her MSc degree in Mathematics in 1999 from Fuzhou University and her PhD degree in Computer Software and Theory in 2005 from the Institute of Software, Chinese Academy of Sciences. She has been working as a visiting researcher at the Eindhoven University of Technology (2010.9-2011.8). She is an assistant professor in the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. Her research interests include the design of embedded systems, software architecture and formal method.

Twan Basten is a professor in the Electrical Engineering Department at Eindhoven University of Technology and a research fellow of the Embedded Systems Institute, both in the Netherlands. He holds an MSc and a PhD in Computing Science from Eindhoven University of Technology. His research interests include the design of embedded systems, reliable computing and computational models.

Marc Geilen is an assistant professor in the Electrical Engineering Department at Eindhoven University of Technology. He holds an MSc in Information Technology and a Ph.D. from the Eindhoven University of Technology. His research interests include modeling, analysis and synthesis of stream-processing systems, multiprocessor systems-on-chip and wireless sensor networks, and multi-objective optimization and trade-off analysis.

Sander Stuijk received his M.Sc. degree (with honors) in Electrical Engineering in 2002 and his Ph.D. degree in 2007 from the Eindhoven University of Technology. He is currently an assistant professor in the Department of Electrical Engineering at the Eindhoven University of Technology. His research interests include modeling methods and mapping techniques for the design, specification, analysis and synthesis of predictable hardware/software systems.