Static Rate-Optimal Scheduling of Multirate DSP Algorithms via Retiming and Unfolding

Xue-Yang Zhu
State Key Laboratory of Computer Science
Institute of Software Chinese Academy of Sciences
Beijing, China
zxy@ios.ac.cn

Marc Geilen, Twan Basten and Sander Stuijk
Department of Electrical Engineering
Eindhoven University of Technology
Eindhoven, the Netherlands
{m.c.w.geilen, a.a.basten, s.stuijk}@tue.nl

***Abstract*—This paper presents an exact method and a heuristic method for static rate-optimal multiprocessor scheduling of real-time multirate DSP algorithms represented by synchronous data flow graphs (SDFGs). Through exploring the state-space generated by a self-timed execution (STE) of an SDFG, a static rate-optimal schedule via explicit retiming and implicit unfolding can be found by our exact method. By constraining the number of concurrent firings of actors of an STE, the number of processors used in a schedule can be limited. Using this, we present a heuristic method for processor-constrained rate-optimal scheduling of SDFGs. Both methods do not explicitly convert an SDFG to its equivalent homogenous SDFG. Our experimental results show that the exact method gives a significant improvement compared to the existing methods; our heuristic method further reduces the number of processors used.**

I. Introduction

DSP algorithms are usually required to operate under real-time constraints and with limited resources. In this paper, we are concerned with constructing efficient static (compile-time) schedules for multirate DSP algorithms. We also take into account the number of processors used.

Dataflow models are widely used to represent DSP applications. The one often used for multirate DSP algorithms are *synchronous dataflow graphs* (SDFGs) [1]. Each node (also called actor) in an SDFG represents a computation or function and each edge models a FIFO channel; the sample rates of actors may differ. Practical multirate DSP applications modeled with SDFGs include a spectrum analyzer [2], a satellite receiver [3], etc.

DSP algorithms are nonterminating and repetitive. Static schedules are usually used for them. A *static schedule* arranges computations of an algorithm to be executed repeatedly. Execution of all the computations for the required number of times is referred to as an *iteration*. An iteration of an SDFG may include more than one execution, often called *firing* in dataflow, of an actor, and a different number of firings for different actors. The average computation time per iteration is called the *iteration period* of a schedule. DSP algorithms with recursions (or feedbacks) have an inherent lower bound on the iteration period, referred to as the *iteration bound* [4]. It is impossible to achieve an iteration period lower than the iteration bound, even when unlimited processors are available. A schedule whose iteration period equals the iteration bound is called a *rate-optimal* schedule.

The construction of rate-optimal schedules involves explicit or implicit *retiming* and *unfolding*. Unfolding turns f consecutive iterations into a *cycle* in the unfolded graph. f is called the *unfolding factor*. Unfolding can lead to a rate-optimal schedule, at the cost of multiplying the problem space by the unfolding factor. An unfolding factor is called an *optimal unfolding factor* if a rate-optimal schedule exists with the factor. Combined with retiming [5], a smaller optimal unfolding factor may be obtained [6], [7].

Much work [7], [8], [9] has been done on static rate-optimal scheduling of *homogenous synchronous dataflow graphs* (HSDFGs), which is a special type of SDFGs. All sample rates of actors of an HSDFG are one. In an iteration of an HSDFG, each actor fires *once*. The authors of [8] prove that the least common multiple of the delay counts in all the loops of an HSDFG is an optimal unfolding factor and present an algorithm for rate-optimal scheduling HSDFGs with explicit unfolding. They do not take into account the actor execution time. The mentioned optimal unfolding factor is usually larger than the one that can be found when the execution times of actors are considered. The authors of [7] prove the minimum rate-optimal unfolding factors of an HSDFG on different timing models and implementation styles. They also provide a rate-optimal scheduling algorithm with implicit retiming and unfolding that achieves these unfolding factors. The algorithm in [9] finds a rate-optimal schedule for an HSDFG if such a schedule exists when limiting the schedule to one iteration.

Little work, however, is concerned with general SDFGs. Theoretically, it is always possible to convert an SDFG to its equivalent HSDFG [10] and then use the available methods for HSDFGs. However, converting an SDFG to an HSDFG is very time-consuming when SDFGs scale up. The size of the HSDFG can be exponentially larger than the original SDFG in extreme cases [11]. To the best of our knowlege, only [12] discusses the rate-optimal scheduling of SDFGs. It converts

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

an SDFG to a precedence graph, which is a reduced form of its equivalent HSDFG, then computes the unfolding factor and schedules the precedence graph with the same method as [7]. A precedence graph of an SDFG has less edges than and the same number of actors as its equivalent HSDFG. The conversion procedure is still time and space-consuming.

All the above-mentioned methods, except for [9], are based on the assumption that unlimited processors are available. Besides a method based on the same assumption, we also consider rate-optimal scheduling under processor constraints. The static schedules we consider in this paper are the same as the static schedules under the integral timing model and pipelined design in [7]. The minimum optimal unfolding factor is the denominator of the iteration bound in its irreducible form.

In this paper, we work directly on SDFGs, without explicitly converting an SDFG to an HSDFG or any other graphs. We consider explicit retiming and implicit unfolding. All the above-mentioned methods provide solutions by analyzing the structure of dataflow graphs. We take another perspective – analyzing the behaviors of SDFGs. [13] proves that the iteration bound of an SDFG can be computed by exploring the state space generated by a self-timed execution (STE, also called as soon as possible execution). The method is very efficient. STE analysis was used in [14] to construct static periodic rate-optimal schedules with minimal buffer sizes for HSDFGs. In this paper, we consider STE analysis for SDFG scheduling. We show that the state space includes the information of a retiming, an optimal unfolding factor and a rate-optimal schedule of the retimed and unfolded graph. Furthermore, the number of processors needed by the rate-optimal schedule can be found. Based on this, we present an exact method for rate-optimal scheduling of SDFGs. Limiting the number of the concurrent firings of actors in an STE, we can limit the number of processors it uses, and therefore the number of processors used by the schedule obtained from the STE. However, such a schedule is not always rate-optimal. We present a heuristic method trying to find a static rate-optimal schedule of an SDFG using as few processors as possible.

For evaluating our new methods, we implemented them and related methods in the open source tool SDF3 [15]. We compare the execution time of our exact method with the method in [12], which is faster than the methods that need to convert an SDFG to its HSDFG. While our method guarantees rate-optimal schedules, we do not achieve the minimum unfolding factors in all cases. We compare the unfolding factor obtained by our method with the minimum one proven in [7]. We show how our heuristic method improves the processor utilization of our exact method and show its execution time. The experiments were carried out on hundreds of synthetic SDFGs and several models of real applications.

We first introduce the related concepts and formulate the problems in Section II. The basic idea of our methods is introduced by an example in Section III. Section IV describes the definition and properties of self-timed execution of SDFGs. We formally define our methods in Sections V and VI.

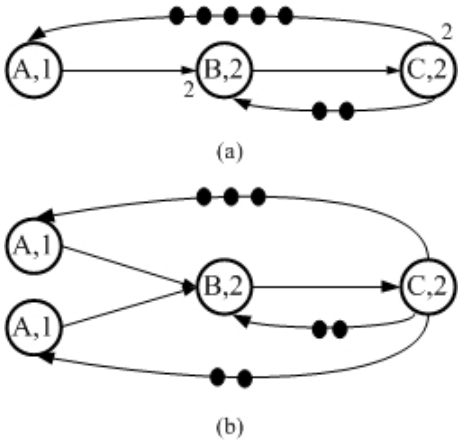

Fig. 1. (a) The SDFG G_1, where the sample rates are omitted when they are 1 and the computation time of each actor is attached inside the node; (b) the equivalent HSDFG of G_1.

Section VII provides an experimental evaluation. Finally, in Section VIII the conclusions and future work are presented.

II. Preliminaries and Problem Formulation

A *synchronous dataflow graph* (SDFG) is a finite directed graph $G = \langle V, E, t, d, prd, cns \rangle$, in which V is the set of actors, modeling the functional elements of the system; E is the set of directed edges, modeling interconnections between functional elements. Each actor v is weighted with its computation time $t(v)$, a nonnegative integer. Each edge e is weighted with three properties: $d(e)$, a nonnegative integer that represents the number of initial tokens associated with e; $prd(e)$, a positive integer that represents the number of tokens produced onto e by each firing of the source actor of e; $cns(e)$, a positive integer that represents the number of tokens consumed from e by each firing 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 are denoted as $src(e)$ and $snk(e)$, respectively. If $prd(e) = cns(e) = 1$ for each $e \in E$, then we say that G is a *homogeneous SDFG* (HSDFG).

The edge e with source actor u and sink actor v is denoted by $e = \langle u, v \rangle$. The set of incoming edges to actor v is denoted by $InE(v)$, and the set of outgoing edges from v by $OutE(v)$. An *initial delay distribution* of the SDFG G is a vector containing delays on all edges of G, denoted as $d(G)$.

An SDFG G is *sample rate consistent* [1] if and only 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*. We use q to represent the repetition vector directly. For example, a balance equation can be

constructed for each edge of G_1 in Fig. 1 (a). By solving these balance equations, we have G_1's repetition vector $q = [2, 1, 1]$.

One *iteration* of an SDFG G is a firing sequence in which each actor v occurs exactly $q(v)$ times. An iteration of G_1 in Fig. 1(a), for example, includes two firings of actor A and one firing of B and C, respectively.

Only sample rate consistent and deadlock-free SDFGs are meaningful in practice. Henceforth we consider only such SDFGs.

A *static schedule* is a function S, mapping a firing to its start time. The i^{th} firing of actor v starts at time $S(v, i)$. DSP algorithms are nonterminating, implying that $i \in [1, \infty)$. If $S(v, i + f \cdot q(v)) = S(v, i) + T$ for every firing of v, we say that the schedule S has an *unfolding factor* f and a *cycle period* T. Each f iterations are called a *cycle* in the schedule. Such a schedule S can be represented by the first f iterations, which is the part of the schedule defined by $S(v, i)$ with $1 \leq i \leq f \cdot q(v)$ for all v.

The *iteration period* (IP) of a static schedule is the average computation time of an iteration, that is, $\frac{T}{f}$. The *iteration bound* (IB) is the greatest lower bound of the IP. If the IP of an schedule equals its IB, it is a *rate-optimal schedule* and f is a *rate-optimal unfolding factor*. The IB of an HSDFG is given by its *maximum cycle mean*. That is,

$$IB = \max_{\forall l \in G} \frac{t(l)}{d(l)}, \tag{2}$$

where l is a loop (a simple cycle) in the graph, $t(l)$ is the sum of the execution times of all actors in l and $d(l)$ is the sum of the delays in l.

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 [10]. The IB of an SDFG equals the IB of its equivalent HSDFG. For example, the IB of G_1 in Fig. 1 (a) is $\frac{5}{2}$, which can be computed by using Equation (2) on it equivalent HSDFG shown in Fig. 1 (b). [13] presents a method to compute the IB directly on an SDFG, which we introduce later.

An optimal unfolding factor may be reduced when retiming is used. For example, the optimal unfolding factor of Fig. 1 (b) according to [8] is 6, while its smallest optimal unfolding factor is 2 when the actor execution time is taken into account and an implicit retiming is used [7].

Retiming is a graph transformation technique that redistributes the graph's delays while its functionality remains unchanged [5], [2]. Retiming an actor once means firing this actor once. Given an SDFG $G = \langle V, E, t, d, prd, cns \rangle$, a *retiming* of G is a function $r : V \rightarrow \mathbb{Z}$, 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 $e = \langle u, v \rangle$ by the equation:

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

A retiming r of a sample-rate consistent and deadlock-free SDFG is *legal* if the retimed graph $r(G)$ is also sample-rate consistent and deadlock-free. Only legal retimings are meaningful. It is sufficient to check if the initial delay distribution of the retimed graph is nonnegative to ensure that a retiming is legal [16]. Note that retiming does not affect the iteration bound of an SDFG.

When there are unlimited processors available, a static rate-optimal schedule can always be found [8]. The first problem we address is: given an SDFG G, finding a legal retiming r, an unfolding factor f and a rate-optimal schedule S with f of the retimed graph $r(G)$. The second problem we address is: given an SDFG G, trying to find a rate-optimal schedule using as few processors as possible.

By inserting precedence constraints with a finite number of delays between the source and sink actors of an SDFG, any SDFG can be converted to a strongly connected graph [5], [2]. We therefore only consider strongly connected SDFGs when developing our ideas.

III. Basic Idea by an Example

[13] proves that the IB (i.e., the reciprocal of the maximum throughput in [13]) of an SDFG can be obtained by exploring the state space generated by an STE, in which actors must fire as soon as possible [10]. An STE of an SDFG ultimately goes into a repetitive pattern, which is called the *periodic phase*. The periodic phase includes one or more complete iterations. The firing sequence before the periodic phase is called the *transient phase*. The average iteration computation time in the periodic phase is exactly the IB of the SDFG.

For example, Fig. 2 (a) gives the actor firing sequence obtained by an STE of G_1. At time 6, the delays on each edge are the same as at time 1. Thus the firings enabled are the same at these two time points. They are two firings of B. The firings of actors between time 1 and 6 will be repeated infinitely. There are two iterations in the periodic phase, which has a duration of 5. Then the IB of G_1 is $\frac{5}{2}$, which is exactly the maximum cycle mean of the equivalent HSDFG of G_1, shown in Fig. 1 (b).

Take a closer look at the firing sequence generated by the STE. After the firings of the transient phase finish, each of the following firings will repeatedly start with time displacement 5 and the number of firings of each actor v in each repetition is $2q(v)$. The firings in the periodic phase exactly form a rate-optimal schedule with unfolding factor 2 and cycle period 5 of the SDFG transformed by executing the transient phase. Recall that retiming an actor once means firing it once. The firings of the transient phase form a retiming.

By the STE shown in Fig. 2 (a), the retiming $r = [5, 0, 0]$ is obtained. The retimed graph is shown in Fig. 2 (b). A rate-optimal schedule of the retimed SDFG, shown in Fig. 2 (c), is in fact the firing sequence appearing in the periodic phase of the STE, as shown in Fig. 2 (a).

The STE also reveals the number of concurrent firings at each time point. The maximum number of concurrent firings in the periodic phase of the STE implies the number of processors needed by a rate-optimal schedule, 4 in the example.

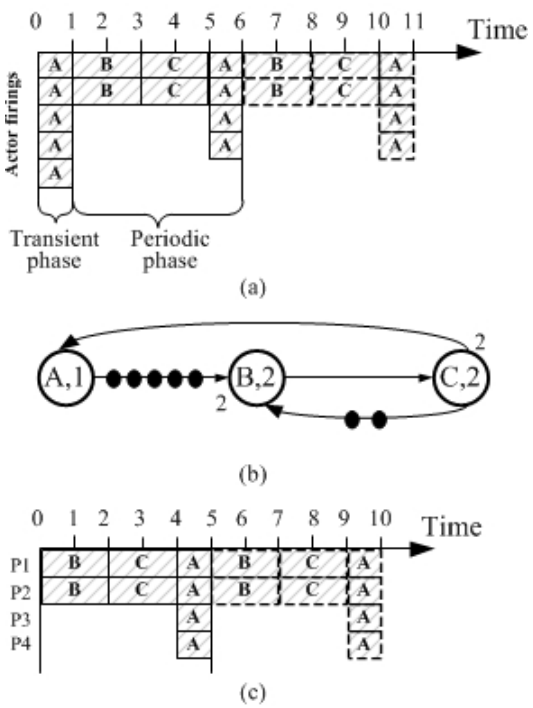

Fig. 2. (a) The actor firing sequence obtained by the STE of G_1; (b) a retimed SDFG of G_1, with $r(A) = 5$ and $r(B) = r(C) = 0$; (c) a rate-optimal schedule of the retimed SDFG with unfolding factor 2.

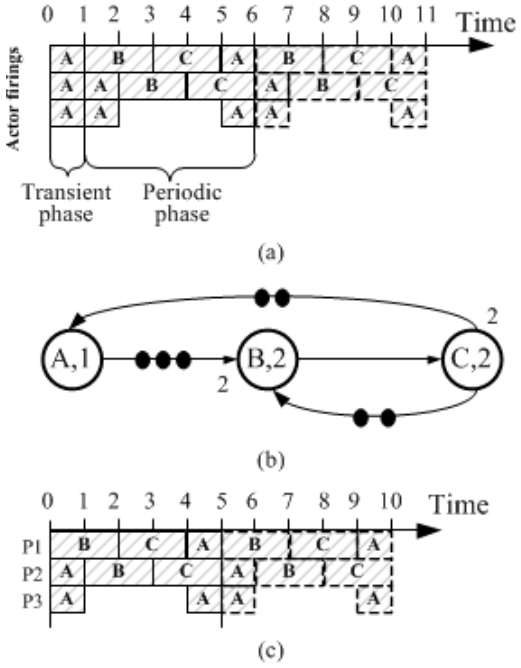

Fig. 3. (a) The actor firing sequence obtained by a constrained STE of G_1 in which the number of concurrent firings is limited to 3; (b) a retimed SDFG of G_1, with $r(A) = 3$ and $r(B) = r(C) = 0$; (c) a rate-optimal schedule with 3 processors.

Limiting the number of concurrent firings in an STE may limit the number of processors used for a rate-optimal schedule. For example, when there are only 3 processors available, we can limit the number of concurrent firings of G_1 to no more than 3 and then get a constrained STE as shown in Fig. 3 (a). From this, we obtain a retimed SDFG as shown in Fig. 3 (b) and a schedule for the retimed graph, as shown in Fig. 3 (c). It is rate-optimal. Therefore 3 processors is feasible for a rate-optimal schedule of a retimed G_1.

For developing our method, we formalize the definition and properties of self-timed execution in the next section.

IV. Self-Timed Execution

A. An Operational Semantics of SDFGs

We define the behavior of an SDFG G in terms of a *labeled transition system* (LTS), represented by $LTS(G)$. An LTS includes a set of states, a set of actions, an initial state and a set of transitions which define rules on how to change states depending on different actions. Before defining the LTS of an SDFG, we introduce some notations to simplify the later illustrations.

We use a vector $tn(E)$ to model the change of the delay distribution of G during its execution. $tn(e)$ is the current number of delays on edge e. The SDFG is a concurrent model of computation. It allows simultaneous firings of an actor. For different concurrent firings of an actor, the one first to start is the one first to end. We use a queue $tr(v)$ to contain the remaining times of the concurrent firings of actor v. The i^{th} element of $tr(v)$ is the remaining time of the i^{th} unfinished firing of v.

Generally, the behavior of an SDFG can be observed on two levels — the global behavior represented by the change of the delay distribution and the local behavior that indicates which actors are firing and how much time remains for each firing. The $tn(E)$ characterizes the global behavior and $tr(V)$ characterizes the local behavior.

Our purpose is to construct fast schedules, so we need to hold a global clock, *glbClk*, to record the time progress.

A *state* of $LTS(G)$ is a 2-tuple that consists of the values of $tn(E)$ and $tr(V)$. The *initial state* of $LTS(G)$ is denoted as s_0. In s_0, $tn(E)$ is the initial delay distribution $d(G)$; no firings have been started, so each element of $tr(V)$ is empty.

The behavior of an SDFG consists of a sequence of *firings* of actors. We use actions $sFiring(v)$ and $eFiring(v)$ to model the start and end of a firing of actor v, and use $readyS(v)$ and $readyE(v)$ as their enabling conditions, respectively. In parallel with actor firing, time elapses, represented by the increase of the global clock *glbClk*. A time step is modeled by the action

clk.

The guard *readyS*(v) tests if there are sufficient tokens on the incoming edges of actor v for a firing of v. That is,

$$readyS(v) \equiv_{def} \quad \forall e \in InE(v) : tn(e) \geq cns(e).$$

An actor v starting a firing reduces delays of all its incoming edges according to the consumption rates and inserts its computation time, $t(v)$, into queue $tr(v)$. That is,

$$\begin{aligned} sFiring(v) \equiv_{def} \quad & (\forall e \in InE(v) : tn'(e) = tn(e) - cns(e)) \\ & \wedge tr'(v) = ENQ((tr(v), t(v)), \end{aligned}$$

where $tn'(e)$ and $tr'(v)$ refer to the value of $tn(e)$ and $tr(v)$ in the new state, respectively; $ENQ(tr(v), t(v))$ inserts $t(v)$ at the end of $tr(v)$. For conciseness, we omit the elements of states if their values remain unchanged.

When the remaining time of a firing of v is zero, the firing is ready to end. This is modeled by the guard *readyE*(v).

$$readyE(v) \equiv_{def} \quad HeadQ(tr(v)) = 0,$$

where $HeadQ(tr(v))$ returns the first element of $tr(v)$.

An actor v ending a firing increases delays of all its outgoing edges according to the production rates, and removes the first element from queue $tr(v)$. That is,

$$\begin{aligned} eFiring(v) \equiv_{def} \quad & (\forall e \in OutE(v) : tn'(e) = tn(e) + prd(e)) \\ & \wedge tr'(v) = DLQ((tr(v)), \end{aligned}$$

where $DLQ(tr(v))$ removes the first element of $tr(v)$.

Time progresses as much as possible when no actor is ready to end. The largest possible time step is the minimal element of $tr(v)$ for all v. We use *mStep* to represent it.

$$mStep = \min_{c \in tr(v) \wedge v \in V} c.$$

A time step reduces the remaining times of all firings by *mStep* and increases the global clock by *mStep*. That is,

$$\begin{aligned} clk \equiv_{def} \quad & (\forall v \in V : \neg isEmpty(tr(v)) \\ & \Rightarrow (\forall x \in tr(v) : x' = x - mStep)) \\ & \wedge glbClk' = glbClk + mStep, \end{aligned}$$

where $isEmpty(tr(v))$ tests if $tr(v)$ is empty. Its enabling conditions guarantee that a *clk* action leads to a nonnegative value in $tr(v)$. The delay distribution remains unchanged by a time step. In a time step, time may not progress if an actor with zero execution time has started firings.

An *action* of $LTS(G)$ is any of the *sFiring*, *eFiring* and *clk* actions. A *transition* from state to state of $LTS(G)$ is caused by any of its actions constrained by their enabling conditions.

An action happens at a certain time point followed by a state at the same time. We use $a.glbClk$ to represent the time when the action or state a occurs.

An *execution* of an SDFG G is an infinite alternating sequence of states and transitions of $LTS(G)$. We use actions to represent transitions that are caused by them.

The IB can be obtained from one specific type of execution, self-timed execution [13]. We will use the information of a self-timed execution to deliver a rate-optimal schedule. A *self-timed execution* (STE) is an execution in which the time step only occurs when no actors are ready to start [17]. That is, in an STE, firings of actors start immediately when they are enabled. The time step cannot progress if there are actors ready to end or start a firing.

B. Properties of Self-timed Execution

In an STE, between two *clk* actions, there can be some interleaving of simultaneous *sFiring* and/or *eFiring* actions. The order of these actions may differ. However, no matter what order of these actions is selected, the sequences of actor firings according to any STE of a strongly connected SDFG are the same. For example, Fig. 2 (a) shows the only sequence of actor firings of all STEs of G_1. Hence we consider only one order at each time point: end all firings that are ready to end and then start any firing that are ready to start. The state at each time point is therefore unique according to this order. Self-timed SDFG behavior is deterministic if we only consider its effects on the actor firing sequence [13]. Seen from this perspective, there is only one STE for an SDFG.

For a strongly connected SDFG, its STE is deterministic and the values of $tn(E)$ and $tr(V)$ are finite; therefore, the STE includes finitly many states that are distinct. The STE ultimately goes into a repetitive pattern.

Property 1: [13] Given the STE σ of a strongly connected SDFG, we have $hasCycle(\sigma) = true$, where

$$hasCycle(\sigma) \equiv_{def} \exists s_1, s_2 \in \sigma : s_1 = s_2.$$

By Property 1, it is easy to see that the state-space of the STE includes a finite sequence of states and actions (*transient phase*), followed by an infinite sequence that is periodically repeated (*periodic phase*). The first pair of states of σ to make $hasCycle(\sigma)$ true is the beginning state of the periodic phase, denoted as s_b, and the end state of the periodic phase, denoted as s_e. For example in the STE shown in Fig. 2 (a), s_b is the state when $glbClk = 1$ and s_e is the state when $glbClk = 6$.

Although an STE is infinite, we can find it in finitely many steps of exploration by Property 1: beginning with the initial state s_0 and ending at s_e. We directly call such a finite state sequence an STE in the remainder of the paper.

For an STE σ, we denote its transient phase as σ_t and its periodic phase as σ_p. The time that passes in σ_p is

$$CP(\sigma_p) = s_e.glbClk - s_b.glbClk.$$

Property 2: [13] The periodic phase of an STE consists of a whole number of iterations.

If σ_p consists of n iterations, we denote it as $nIter(\sigma_p) = n$. For each actor v, the numbers of $sFiring(v)$ and $eFiring(v)$ actions in σ_p are both $n \cdot q(v)$.

The iteration period in the periodic phase is defined as

$$IP(\sigma_p) = CP(\sigma_p)/nIter(\sigma_p).$$

Property 3: [13] Given the STE σ of an SDFG, its iteration bound $IB = IP(\sigma_p)$.

V. Rate-Optimal Scheduling

In this section, we first present an algorithm to explore the state-space of an SDFG according to self-timed execution; then we deliver our rate-optimal scheduling algorithm based on the state-space.

Since self-timed execution is deterministic, we can explore its state space according to macro steps that enforce an order of actions. A *macro step* includes: starting all firings ready to start, then a time step, and at last ending all firings ready to end. Algorithm 1 returns the STE of an SDFG according to macro steps.

Algorithm 1 STE(G)

Require: A strongly connected SDFG G
Ensure: The STE σ of G

```
1:  ts = LTS(G)
2:  s = ts.s_0
3:  while not hasCycle(σ) do
4:    for all v ∈ G do
5:      while readyE(v) do
6:        eFiring(v)
7:        σ ← σ + eFiring(v)
8:      end while
9:    end for
10:   for all v ∈ G do
11:     while readyS(v) do
12:       sFiring(v)
13:       σ ← σ + sFiring(v)
14:     end while
15:   end for
16:   σ ← σ + s
17:   clk
18: end while
19: return σ
```

The termination of Algorithm 1 is guaranteed by Property 1. Only one state is stored in each macro step as we explained before. The information of start firings is related with scheduling, so at each moment we store the state after all enabled firings have been started.

According to the operational semantics, the delay distribution decreases only after an *sFiring* action. The enabling condition of *sFiring* guarantees that the delay distribution never goes negative in an execution. Hence, the transient phase of the STE forms a legal retiming.

Theorem 4: Given the STE σ of an SDFG G, the retiming r defined as below is legal. For each v,

$$r(v) = \text{ the number of } sFiring(v) \text{ actions in } \sigma_t.$$

By Properties 2 and 3, in the periodic phase of an STE σ, the average computation time for each iteration equals the IB. If the SDFG is equivalently transformed to a new graph whose initial delay distribution is the same as the delay distribution of a state in σ_p, then the *sFiring* actions in σ_p, combined with the times they happen shifted by $s_b.glbClk$, form a rate-optimal schedule of the new graph. It is obvious that a retiming obtained from σ_t transforms the SDFG to such a new graph.

Theorem 5: Given the STE σ of an SDFG G and the retiming obtained from σ_t, r, a rate-optimal schedule S of $r(G)$ with the unfolding factor $f = nIter(\sigma_p)$ is defined as:

$$S(v,i) = sFiring(v).glbClk - s_b.glbClk,$$

where $sFiring(v)$ is the i^{th} firing of v in σ_p and $1 \le i \le f \cdot q(v)$.

For SDFG G_1, shown in Fig. 1 (a), its STE is shown in Fig. 2 (a). The retiming obtained by the STE is $r(A) = 5, r(B) = r(C) = 0$; the retimed graph is shown in Fig. 2 (b). The optimal unfolding factor is 2 and the rate-optimal schedule for the retimed graph is shown in Fig. 2 (c).

See the rate-optimal schedule shown in Fig. 2 (c). Concurrent firings need to occupy different processors. When there are 4 concurrent firings, 4 processors are needed. This is the number of processors needed for the schedule obtained from the periodic phase of the STE shown in Fig. 2 (a). If we can find the maximal number of concurrent firings in the periodic phase of an STE, we know how many processors are needed in a rate-optimal schedule.

The vector $tr(V)$ contains the remaining times of concurrent firings of all actors. Then the number of concurrent firings at a state s is the sum of all $|s.tr(v)|$, where $|s.tr(v)|$ represents the size of queue $s.tr(v)$.

Theorem 6: Given the STE σ of an SDFG G and the retiming obtained from σ_t, r, the number of processors needed by a rate-optimal schedule of $r(G)$ is

$$numP = \max_{s \in \sigma_p} \sum_{v \in V} |s.tr(v)|.$$

An algorithm for rate-optimal scheduling SDFGs is shown in Algorithm 2. Lines 2 - 6 accumulate the number of start firing actions in the transient phase to form the retiming. Lines 8 - 13 record the time when each start firing action of the periodic phase happens as the schedule. The correctness of the algorithm is guaranteed by Theorems 4, 5, and 6.

The schedule computed using our method is static. It schedules f iterations of the actors in an SDFG as a cycle and it executes this cycle repeatedly. The use of retiming allows our algorithm to construct a rate-optimal schedule in which the start time of all firings in the first cycle are not larger than T, i.e., $S(v,i) \le T$ for $i \le f \cdot q(v)$. This implies that all firings that belong to the one iteration of the cycle will be started before the first firing in the next iteration of the cycle. Alternative rate-optimal scheduling techniques presented in [7] and [12] do not have such a property.

VI. Rate-Optimal Scheduling under Processor Constraints

Static rate-optimal Scheduling of data flow graphs is usually discussed under the assumption that there are unlimited processors available [8], [18], [7]. The paper addressing rate-optimal scheduling of SDFGs [12] also does not consider the

Algorithm 2 optSch(G)

Require: A strongly connected SDFG G

Ensure: A legal retiming r, a rate-optimal schedule S of $r(G)$ with unfolding factor f and a cycle period T, and the number of processors needed by S, $numP$

1: $\sigma = STE(G)$
2: **for all** actions $a \in \sigma_t$ **do**
3: **if** $a = sFiring(v)$ **then**
4: $r(v) = r(v) + 1$
5: **end if**
6: **end for**
7: set $i(v) = 1$ for all $v \in G$
8: **for all** actions $a \in \sigma_p$ **do**
9: **if** $a = sFiring(v)$ **then**
10: $S(v, i(v)) = a.glbClk - s_b.glbClk$
11: $i(v) = i(v) + 1$
12: **end if**
13: **end for**
14: $f = nIter(\sigma_p)$
15: $T = CP(\sigma_p)$
16: $numP = \max_{s \in \sigma_p} \sum_{v \in V} |s.tr(v)|$
17: **return** r, f, T, S and $numP$

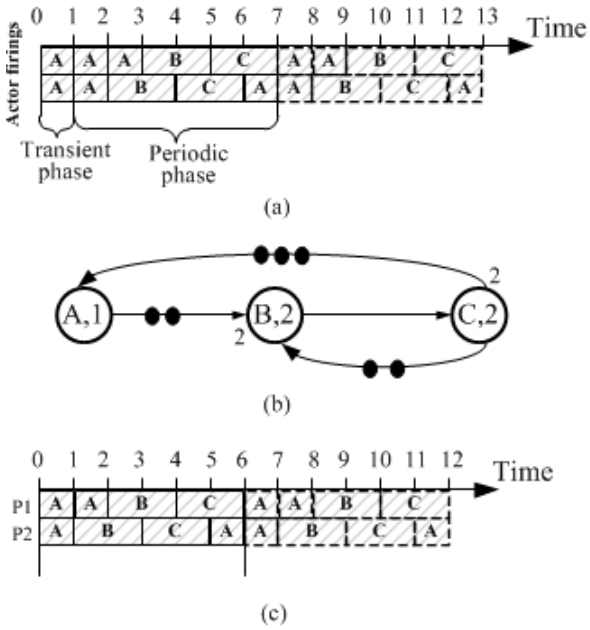

Fig. 4. (a) The STE of G_1 with the number of concurrent firings limited to 2; (b) a retimed SDFG of G_1, with $r(A) = 2$ and $r(B) = r(C) = 0$; (c) a fastest schedule with 2 processors.

processor constraints. We present in this section a heuristic method trying to find a static rate-optimal schedule of an SDFG using as few processors as possible.

By Theorem 6, we know that the number of concurrent firings affects the number of processors used in a schedule. Then, if we limit the number of concurrent firings in an STE of an SDFG to an integer P, may we find a rate-optimal schedule of the SDFG or its retimed graph using no more than P processors? It depends.

The constraint can be put on the guard of *sFiring* actions. We denote the new guard as $readyS_P(v)$.

$$readyS_P(v) \equiv_{def} \quad readyS(v) \wedge (\sum_{v \in V} |tr(v)| < P).$$

The procedure to obtain such a constrained STE of G, $conSTE(G, P)$, is a variation of Algorithm 1, in which $readyS(v)$ is replaced with $readyS_P(v)$. For example, Fig. 3 (a) is a constrained STE of G_1 with $P = 3$. By a procedure similar to Algorithm 2, we get the retimed graph in Fig. 3 (b) and a rate-optimal schedule of it in Fig. 3 (c). We call this procedure $conSch(G, P)$. It is a variation of Algorithm 2, in which $STE(G)$ is replaced with $conSTE(G, P)$.

For an STE, at each macro step, all the firings start if they are ready to start, so the order of *sFiring* actions does not matter. In a constrained STE with P processors, at a macro step, the sum of simultaneously ready firings and already active concurrent firings may be more than P and these ready firings may be of different actors. Then, which ones are chosen to start may affect the state space of the constrained STE and therefore the T and f of a schedule gotten from it. Therefore, a constrained STE no longer always leads to the IP of its periodic phase being equal to the IB; and the schedule obtained by $conSch(G, P)$ is not always rate-optimal. We use a fixed random order to select the firings. A smarter selection mechanism, i.e. the method used in [19], may improve the results, but that is left for future work.

There is a lower bound on the number of processors for a rate-optimal schedule. If an SDFG is scheduled on a single processor, the total time used by an iteration is the sum of $t(v)$ times $q(v)$ for all v, denoted as $sumT$. In a multi-processor schedule, an iteration may be folded and set onto different processors. If the schedule is rate-optimal, the number of processors it needs is at least the quotient of the sumT and the IB. That is, the lower bound on the number of processors for a rate-optimal schedule is

$$minP = \left\lceil \frac{\sum_{v \in V} t(v)q(v)}{IB} \right\rceil.$$

For example, $minP$ for G_1 is 3. This means that with 2 processors available, no rate-optimal schedules exist for G_1 or any retimed graph of it. When $P = 2$, the constrained STE of G_1 is shown in Fig. 4 (a). The IP of its periodic phase is 3 and 2 processors are fully used.

Suppose a rate-optimal schedule uses nP processors. The closer nP to $minP$, the fuller processors are used. We define the *rate of processors used* by a schedule as $\frac{minP}{nP}$. The fewer processors used by a rate-optimal schedule, the higher the rate is. We use this rate in Section VII to measure the processor utilization of a scheduling algorithm.

If the T, f and $numP$ obtained from $conSch(G, P)$ meet $\frac{T}{f} = IB$, then we say that $numP$ is feasible for a rate-optimal schedule of G. A feasible $numP$ does not linearly decrease with P. When $\frac{T}{f} > IB$, there still may exist an integer P'

less than P such that $conSch(G, P')$ returns a feasible $numP$. Furthermore, for two integers P_1 and P_2 with $P_1 > P_2$, a feasilbe $numP$ returned by $conSch(G, P_1)$ may be less than a feasible $numP$ returned by $conSch(G, P_2)$.

Nevertheless, a constrained STE does help to reduce the number of processors used by a rate-optimal schedule, as our experimental results show in the next section. Algorithm 3 is a heuristic algorithm trying to find a rate-optimal schedule using as few processors as possible.

Algorithm 3 minOptSch(G)

Require: A strongly connected SDFG G

Ensure: A legal retiming r, a rate-optimal schedule S of $r(G)$ with the number of processors

1: Get the IB and $numP$ from $STE(G)$
{According to Property 3 and Theorem 6}

2: Perform a binary search over $[minP, numP]$; assuming P' is the number of processors considered, use $conSch(G, P')$ to test whether the T and f returned by $conSch(G, P')$ satisfy $\frac{T}{f} = IB$
{$\frac{T}{f} = IB$ implies that $numP$ obtained by $conSch(G, P')$ is feasible for a rate-optimal schedule}

3: **return** r, S and $numP$ obtained by $conSch(G, P')$

Because the feasibility check for P' in the binary search in Line 2 might result in a false negative, Algorithm 3 might lead to a suboptimal result. The binary search can be replaced with a linear search starting from $minP$ in an attempt to get a further reduction of the number of processors at the cost of a longer execution time, as shown in our experimental results in the next section.

VII. Experimental Evaluation

A. Experimental Setup

We implemented *optSch* (opt), *minOptSch* (min), *minOptSch* with linear search (minLS), and the scheduling algorithm for SDFGs in [12] (GG95) in SDF3 [15]. We also implemented the method to compute the smallest unfolding factor for HSDFGs in [7] (CS95). Converting an SDFG to its equivalent HSDFG, we compute the smallest unfolding factor of the SDFG by CS95. We performed experiments on two sets of SDFGs, running on a 2.67GHz CPU with 12MB cache. The experimental results of these two sets are shown in Tables II, III, IV and V. All execution times are measured in milliseconds (ms).

The first set of SDFGs consists of five practical DSP applications, including a sample rate converter (SaRate) [20], a satellite receiver (Satellite) [3], a maximum entropy spectrum analyzer (MaxES) (http://ptolemy.eecs.berkeley.edu/), an Mp3 playback application (Mp3) (http://www.es.ele.tue.nl/sdf3/) and a channel equalizer (CEer) [21]. 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 second set of test models consists of 540 synthetic strongly connected SDFGs generated by SDF3, mimicking real DSP applications. The number of actors in an SDFG, denoted as nA, and the sum of the elements in the repetition vector, denoted as nQ, have significant impact on the performance of the various methods. We distinguish three different ranges of nA: 10-15, 20-25, and 50-65; and three different ranges of nQ: 1000-1500, 2000-2500, and 4000-6000. The state-space of an SDFG may increase with the increase of its delay count. A large delay count may slowdown the *STE* procedure and therefore our scheduling methods. The SDF3 parameter 'initialTokens prop', denoted as nD, is used to control the amount of delays in a generated SDFG in SDF3. The delay count changes from small to large when it is set to be from 0 to 1. To measure the worst cases we may deal with, we choose two values of nD: 0 and 0.9. Then we generate SDFGs according to different combinations of nA, nQ and nD to form 18 groups. Each group includes 30 SDFGs. The explicit difference in nA, nQ and nD among these groups is helpful for showing how the performance of each method changes with them.

TABLE I
Acronyms and symbols

Acronym	Meaning
nA	The number of actors
nQ	The sum of the elements in the repetition vector
nD	The SDF3 parameter 'initialTokens prop'
IB	The iteration bound
opt	The result returned by *optSch*
min	The result returned by *minOptSch*
minLS	The result returned by *minOptSch* with linear search
GG95	The result returned by [12]
CS95	The result returned by [7]
Mp3	The Mp3 playback application
SaRate	The sample rate converter [20]
MaxES	The maximum entropy spectrum analyzer
CEer	The channel equalizer [21]
Satellite	The satellite receiver [3]

B. Experimental Results

Table II gives the information about and results for the practical DSP examples. There are four parts in Table II. The first part is the information on the graphs, including the number of actors in an SDFG (nA), the sum of the elements in the repetition vector (nQ), and the iteration bound (IB); the second part shows the optimal unfolding factor our method obtained (opt) and the the smallest one proven in [7] (CS95); the third part measures the rates of processors used by the schedules returned by *optSch* (opt), *minOptSch* (min) and the procedure when a linear search is used in *minOptSch* (minLS); the last part includes the execution times of different methods. The

information in the first part of Table II includes the dummy actors introduced for strong connectedness. Tables III, IV and V give the results for the synthetic examples. Each point in Tables III and IV is an average of 30 graphs in the same group. Each point in Table V is an average of 270 graphs with the same value of *nD*.

TABLE II
EXPERIMENTAL RESULTS FOR PRACTICAL DSP EXAMPLES

Graph Information					
name	Mp3	SaRate	MaxES	CEer	Satellite
nA	5	7	14	23	23
nQ	10602	613	1289	43	4516
IB	116424	5.25	5764	47128	1.83
Unfolding Factor					
CS95	N	4	1	1	6
opt	1	4	2	1	6
Rate of Processors Used					
opt	24.0%	72.6%	0.5%	14.6%	19.4%
min	24.0%	77.3%	0.5%	43.7%	68.4%
minLS	24.0%	82.2%	0.5%	43.7%	67.5%
Execution Time (ms)					
GG95	N	2,024	580	38,459	3,558,957
opt	6.7	0.7	1.0	0.3	4.0
min	38.3	15.3	3.3	2.0	127.3
minLS	85	341	255	2	33,513

For the practical DSP examples, CS95 and GG95 cannot finish on the Mp3 playback model in ten hours. For other examples, the optimal unfolding factors obtained by *optSch* reach the ones computed by CS95. *optSch* is much faster than GG95 for all examples, as shown in the last part of Table II. As shown in the third part of Table II, in three examples, *minOptSch* improves the rate of processors used of *optSch*. Using a linear search further improves the rate of processors used in one case at the cost of more execution time as shown in the last part. Interestingly, the rate of processors used drops for the Satellite receiver. This is due to the fact that the linear search terminates for a tested P_1 that is lower than the P_2 tested successfully by the binary search, whereas the *numP* returned by $conSch(G, P_1)$ is greater than the *numP* returned by $conSch(G, P_2)$. This effect is rare; it only occurs for this one example among all the evaluted cases, including the synthetic cases.

From the results for the practical DSP examples, it seems that the larger the difference between *nA* and *nQ*, the more efficient *optSch* is in comparison to GG95. Our experiments on the synthetic examples confirm this conclusion, as shown in Tables III. The execution time of GG95 is affected more by the growth of *nQ* than the growth of *nA*. Both *nA* and *nQ* affect the speed of our methods, but not as much as *nD* does, as the difference between the results shown in Tables III and IV, especially for the linear search method of *minOptSch*. *nD* has almost no effect on the speed of GG95, so we don't show its execution time in Tables IV.

TABLE III
EXECUTION TIMES (MS) FOR SYNTHETIC EXAMPLES WITH $nD = 0$

	10-15	20-25	50-65	*nA* / *nQ*
GG95	51,018	58,034	45,263	1k-1.5k*
opt	0.3	0.5	2.0	
min	4.1	6.8	31.7	
minLS	307	290	1,420	
GG95	493,385	520,603	644,658	2k-2.5k
opt	0.5	0.7	2.5	
min	6.8	12.1	40.2	
minLS	748	879	2,477	
GG95	6,446,939	7,388,939	6,875,761	4k-6k
opt	0.9	1.7	4.1	
min	14.7	24.0	57.0	
minLS	2,720	5,547	8,969	

* 1k=1000.

TABLE IV
EXECUTION TIMES (MS) FOR SYNTHETIC EXAMPLES WITH $nD = 0.9$

	10-15	20-25	50-65	*nA* / *nQ*
opt	1.3	1.8	3.4	1k-1.5k
min	45	60	61	
minLS	2,036	1,213	2,577	
opt	9.2	3.3	3.6	2k-2.5k
min	119	104	120.8	
minLS	14,109	8,235	8,240	
opt	6.7	2.6	7.6	4k-6k
min	287	50.4	510.5	
minLS	22,397	12,663	30,148	

Both *nA* and *nQ* have no significant impact on the optimal unfolding factors and the rate of processors used, but *nD* does. We show the average results of all graphs with $nD = 0$ (d0) and $nD = 0.9$ (d9) in Table V, respectively. When the delay count is large, the optimal unfolding factor and the rate of processors used is generally large. *minOptSch* increases the rate of processors used of *optSch* by about 90 percent. A linear search strategy takes much more time than the binary search of *minOptSch*, while the rate of processors used is not quite much improved.

VIII. CONCLUSION

In this paper, we have presented an exact method and a heuristic method for static rate-optimal scheduling of SDFGs, using explicit retiming and implicit unfolding. Both methods

TABLE V
UNFOLDING FACTOR AND RATE OF PROCESSORS USED FOR SYNTHETIC EXAMPLES

	Unfolding Factor		Rate of Processors Used		
	CS95	opt	opt	min	minLS
d0	1.11	1.18	10.5%	20.1%	20.1%
d9	2.18	2.73	21.7%	40.4%	45.0%

do not explicitly convert an SDFG to its equivalent HSDFG. The heuristic method takes into account a constraint on the number of processors.

Our exact method guarantees to get a static rate-optimal schedule with unfolding factor close to the smallest one proven in [7]. The number of processors used can be obtained as well. Our heuristic method further reduces the number of processors used by a rate-optimal schedule. Our experimental results show that the first method is 4-7 orders of magnitude faster than the existing method [12]; our second method further reduces the number of processors needed for a rate-optimal schedule by about 90 percent.

The work we reported in this paper used a fixed random order of firings to generate a constrained STE in the heuristic method. Smarter ways, i.e., the method in [19], might lead to better results for the heuristic. Finding a rate-optimal schedule with minimal buffer sizes of an SDFG seems also interesting.We will invesgate these in the future.

REFERENCES

[1] E. Lee and D. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput*, vol. 36, no. 1, pp. 24–35, 1987.

[2] V. Zivojnovic, S. Ritz, and H. Meyr, "Optimizing DSP programs using the multirate retiming transformation," *Proc. EUSIPCO Signal Process. VII, Theories Applicat*, 1994.

[3] 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.

[4] R. Reiter, "Scheduling parallel computations," *Journal of the ACM (JACM)*, vol. 15, no. 4, pp. 590–599, 1968.

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

[6] L. Lucke, A. Brown, and K. Parhi, "Unfolding and retiming for high-level DSP synthesis," in *Circuits and Systems, 1991., IEEE International Sympoisum on*. IEEE, 1991, pp. 2351–2354.

[7] L. Chao and E. Hsing-Mean Sha, "Static scheduling for synthesis of DSP algorithms on various models," *The Journal of VLSI Signal Processing*, vol. 10, no. 3, pp. 207–223, 1995.

[8] K. Parhi and D. Messerschmitt, "Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding," *Computers, IEEE Transactions on*, vol. 40, no. 2, pp. 178–195, 1991.

[9] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini, "Throughput constraint for synchronous data flow graphs," *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pp. 26–40, 2009.

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

[11] V. Zivojnovic and R. Schoenen, "On retiming of multirate DSP algorithms," in *Proceedings of the Acoustics, Speech, and Signal Processing, 1996. on Conference Proceedings., 1996 IEEE International Conference-Volume 06*. IEEE Computer Society, 1996, pp. 3310–3313.

[12] R. Govindarajan and G. Gao, "Rate-optimal schedule for multi-rate DSP computations," *The Journal of VLSI Signal Processing*, vol. 9, no. 3, pp. 211–232, 1995.

[13] A. Ghamarian, M. Geilen, S. Stuijk, T. Basten, A. Moonen, M. Bekooij, B. Theelen, and M. Mousavi, "Throughput analysis of synchronous data flow graphs," in *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*. IEEE, 2006, pp. 25–36.

[14] O. Moreira, T. Basten, M. Geilen, and S. Stuijk, "Buffer sizing for rate-optimal single-rate data-flow scheduling revisited," *Computers, IEEE Transactions on*, vol. 59, no. 2, pp. 188–201, 2010.

[15] 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. http://www.es.ele.tue.nl/sdf3/, pp. 276–278.

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

[17] 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.

[18] L. Jeng and L. Chen, "Rate-optimal DSP synthesis by pipeline and minimum unfolding," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 1, pp. 81–88, 1994.

[19] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "Exploring trade-offs between performance and resource requirements for synchronous dataflow graphs," in *Embedded Systems for Real-Time Multimedia, 2009. ESTIMedia 2009. IEEE/ACM/IFIP 7th Workshop on*. IEEE, 2009, pp. 96–105.

[20] 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.

[21] 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.