Multi-Constraint Static Scheduling of Synchronous Dataflow Graphs via Retiming and Unfolding

Xue-Yang Zhu, *Member, IEEE,* Marc Geilen, *Member, IEEE,*
Twan Basten, *Senior Member, IEEE,* and Sander Stuijk

***Abstract*—Synchronous dataflow graphs (SDFGs) are widely used to represent DSP algorithms and streaming media applications. This paper presents several methods for binding and scheduling SDFGs on a multiprocessor platform. Exploring the state-space generated by a self-timed execution (STE) of an SDFG, we present an exact method for static rate-optimal scheduling of SDFGs via implicit retiming and unfolding. By modeling a constraint as an extra enabling condition for the STE, we get a constrained STE which implies a schedule under the constraint. We present a general framework for scheduling SDFGs under constraints on the number of processors, buffer sizes, auto-concurrency, or combinations of them. Exploring the state-space generated by the constrained STE, we can check whether a retiming, which leads to a rate-optimal schedule under the processor (or memory) constraint, exists. Combining this with a binary search strategy, we present heuristic methods to find a proper retiming and a static scheduling that schedules the retimed SDFG with optimal rate and with as few processors (or as little storage space) as possible. None of the methods explicitly converts an SDFG to its equivalent homogenous SDFG, the size of which may be tremendously larger than the original SDFG. We perform experiments on several models of real applications and hundreds of synthetic SDFGs. The results show that the exact method outperforms existing methods significantly; our heuristics reduce the resources used and are computationally efficient.**

***Keywords*-Timing optimization, resource optimization, scheduling, mapping, multi-constraint.**

I. Introduction

Synchronous dataflow graphs (SDFGs) [1] are often used for modeling multirate DSP algorithms and streaming media applications, such as a spectrum analyzer [2], or a satellite receiver [3]. Each node (actor) in an SDFG represents a computation or function and each edge models a FIFO channel; the sample rates of actors may differ. *Homogenous synchronous dataflow graphs* (HSDFGs) are a special type of SDFGs. All sample rates of actors of an HSDFG are one.

Algorithms modeled with an SDFG are usually nonterminating and repetitive. They often operate under real-time requirements and a strict resource budget. Static schedules are typically used to reduce runtime overhead. A *static schedule* arranges the computations to be executed repeatedly in a fixed sequence. In this paper, we are concerned with constructing efficient static multiprocessor schedules for SDFGs with constraints on the throughput, number of processors, buffer sizes, and auto-concurrency.

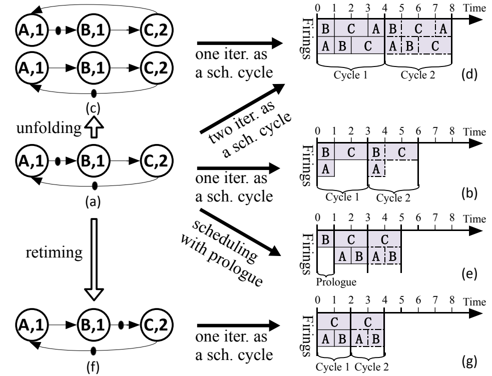

Fig. 1. (a) HSDFG G_0; (b) a periodic schedule of G_0; (c) an unfolded graph of G_0, with unfolding factor 2; (d) a rate-optimal periodic schedule of G_0 with two iterations as a schedule cycle; (e) a schedule with prologue; (f) a retimed graph of G_0; (g) a rate-optimal periodic schedule of G_0 with one iteration as a schedule cycle. The sample rates are omitted when they are 1 and the computation time of each actor is attached inside the node. Black dots represent initial tokens on the edges.

An *iteration* of an SDFG may include more than one firing of an actor, and a different number of firings for different actors, because different sample rates are allowed in an SDFG. In a schedule, the average computation time per iteration is called the *iteration period* (IP). The IP is the reciprocal of the *throughput*. SDFGs with recursions have an inherent lower bound on their IP, referred to as the *iteration bound* (IB) [4]. It is impossible for a schedule to achieve an IP lower than the IB, even when unlimited resources are available. A schedule whose IP equals the IB is called a *rate-optimal* schedule.

The construction of rate-optimal schedules involves explicit or implicit *retiming* and *unfolding*. An unfolded SDFG with an *unfolding factor* f describes consecutive iterations of the original graph [5]. Fig. 1 (c) for example is an unfolded graph of G_0 (Fig. 1 (a)), with unfolding factor $f = 2$. One iteration of graph (c) includes two iterations of graph (a). The IB of G_0 is 2 (the maximum cycle mean of the graph [6]). A periodic schedule of G_0 is shown in (b); its IP is 3. A periodic schedule of the unfolded graph of G_0 ((c)) is shown in (d); its IP is 4. The schedule in (d) is in fact a schedule of G_0 when considering two iterations as a schedule cycle and its IP is 2,

This work was supported in part by National Key Basic Research Program of China (Grant No. 2014CB340701) and the National Natural Science Foundation of China (Grant Nos. 61572478, 61472406 and 61472474).

X. Y. Zhu is with State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China (e-mail: zxy@ios.ac.cn).

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

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

and therefore it's a rate-optimal schedule of G_0. An unfolding factor is called an *optimal unfolding factor* if a rate-optimal schedule exists with this factor.

Unfolding can always lead to a rate-optimal schedule, at the cost of increasing the problem space by the unfolding factor [5]. Combined with retiming [7], a smaller optimal unfolding factor may be obtained [8], [9]. *Retiming* is a graph transformation technique that redistributes the graph's initial tokens, while its functionality remains unchanged. Retiming an actor once means firing the actor once. Retiming may reduce the IP of static periodic schedules of an SDFG [10]. Consider the HSDFG G_0 in Fig. 1 again. A schedule of G_0 with a prologue, shown in (e), consists of some initial firings followed by iterations whose execution time is shorter than the IP of (b). According to this observation, we get the retimed graph of G_0 shown in (f) by retiming B once. A periodic schedule of the retimed graph is shown in (g). It is rate-optimal and with an unfolding factor 1, which is smaller than that of (d). Retiming and unfolding can also be used to reduce the resource requirements of schedules as we illustrate later.

In this paper, we optimize SDFGs via implicit retiming and unfolding. That is, we find schedules of the optimized SDFGs from the original SDFGs, without explicitly transforming them into retimed and unfolded graphs. We assume a homogeneous multiprocessor platform and assume that different firings of an actor can execute on different processors.

This paper is an extension of our previous work [11] and [12]. New contributions in this paper are the following:

1) We extend the operational semantics of SDFGs to capture claiming/releasing processors on actor firings.
2) We incorporate processor allocation of actor firings in the scheduling process.
3) We present a general framework for static scheduling under various constraints.

Theoretically, a static schedule of an SDFG can always be found by converting it to its equivalent HSDFG [6] and using the available methods for HSDFGs. However, the conversion procedure may be very time-consuming. In addition, the size of the HSDFG can be exponentially larger compared to the original SDFG [13]. Our methods work on SDFGs directly, without explicitly converting them to HSDFGs.

Our methods are implemented in SDF^3 [14]. Experiments are carried out on hundreds of synthetic SDFGs and several models of real applications. We compare the execution time of our exact method with the method in [15], 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 optimal unfolding factor in all cases. We compare the unfolding factors obtained by our method with the minimum ones proven in [9]. We show how our heuristic method which tries to minimize the number of processors improves the processor utilization. One of our other heuristics reduces the minimal storage requirement compared to the solution found by [16]. We compare the unfolding factors and the execution times of our static scheduling methods that consider allocating actors to processors with methods ([11] and [12]) that do not consider processor allocation. Finally, we compare our processor-constrained scheduling method with the methods from [17] and [18]. Experimental results show that our methods are still computationally efficient despite their theoretical exponential complexity.

The remainder of this paper is organized as follows. The next section discusses related work. Section III introduces the main concepts and formulates the problems. The basic idea of our methods is introduced by an example in Section IV. Section V describes the definition and properties of self-timed execution of SDFGs. Our scheduling methods are illustrated in Sections VI, VII and VIII. The experimental evaluation is shown in Section IX. Finally, Section X concludes.

II. Related Work

A survey of different multi-processor scheduling approaches can be found in [19]. The scheduling problem addressed in this paper falls under design-time mapping, targets homogeneous architectures and considers multiple optimization goals including throughput (IP), memory and the number of processors. The optimization techniques we consider are retiming and unfolding. The scheduling problems considered in this paper are NP-complete.

By removing edges with initial tokens from an HSDFG, we can obtain the corresponding directed acyclic graph (DAG). Heuristic methods to schedule and map a DAG, or a task graph, while minimizing its makespan are comprehensively surveyed in [20]. [21] optimizes the makespan of a given acyclic HSDFG, considering data parallelisms (by allowing a firing to be executed across more than one processors) as well as task parallelism. Exact methods using complete search techniques such as integer linear programming and model checking have also been studied [22] [23]. [24] uses pipelining, which can be performed with retiming, to maximize the makespan. The makespan of a schedule is equal to its IP when considering only one iteration as a schedule cycle. However a schedule with optimal makespan may not be rate-optimal. In this paper, we focus on finding rate-optimal schedules.

Rate-optimal scheduling usually involves unfolding. In [5] it is proven that a rate-optimal schedule of an arbitrary HSDFG can always be achieved with unfolding. It also presents an upper bound on the number of processors for rate-optimal scheduling of HSDFGs. The authors of [9] prove the minimum rate-optimal unfolding factors for an HSDFG with different timing models and implementation styles. They also provide a rate-optimal scheduling algorithm with implicit retiming and unfolding that achieves these unfolding factors. In [25] it is first proven that the order of retiming and unfolding is immaterial for scheduling an HSDFG. [15] discuses the rate-optimal scheduling of SDFGs without other constraints such as the number of processors used. It converts an SDFG to a precedence graph, which has fewer edges than and the same number of actors as its equivalent HSDFG; it then computes the unfolding factor and schedules the precedence graph with the method from [9]. The conversion procedure is still time- and space-consuming. Paper [26] extends [15] with minimizing buffer requirements. STE analysis was used in [27] to construct static periodic rate-optimal schedules with minimal buffer sizes for HSDFGs. Besides rate-optimal

scheduling assuming no resource constraints as the above-mentioned work, we also consider rate-optimal scheduling under different constraints in this paper.

In [17] heuristics are presented to optimize the number of processors for a given IP and to optimize the IP for a fixed number of processors. It returns the best results on these two problems. Our methods deal with the second problem (fixed number of processors) and the special case of the first problem as described in [17], which applies the IB as a given IP. [28] uses explicit unfolding and retiming to increase the throughput of SDFGs when compiling them onto scratch pad memory based embedded multicore processors, which have processor and memory constraints. This method needs to convert an SDFG to a so-called single appearance SDFG, in which the firings of an actor must start consecutively. They focus on using code and data overlay, which are available by the particular architecture they consider, to reduce memory usage. In this paper, we work directly on SDFGs, without explicitly converting an SDFG to an HSDFG or any other kind of graph.

STE analysis was used in [29] to compute the maximal throughput of SDFGs and in [16] to explore the throughput-buffering trade-off of SDFGs. In [30] it was used to compute the bottleneck of a resource-aware SDFG, in which the resource platform and resource requirement are known. In [31] and [18], it was used to compute throughput after processor allocation for mapping multiple applications. All firings of an actor are allocated in a same processor. Our previous work [11] and [12] uses the STE to find the required retiming, unfolding factor and schedule. In this paper, we take the processor allocation into account and present a multi-constraint scheduling framework. Our method can easily be extended to other resource constraints that are feasible to be represented by the limitation of the concurrency of firings of actors, providing a flexible and extensible framework.

We assume in this paper that different firings of an actor can be executed on different processors to support data parallelism as in [24]. An actor with internal state can be explicitly represented as a self-loop or by an auto-concurrency constraint. We have not considered the delay introduced by processor communication since we assumed that this cost is small compared to the actor execution times. Considering communication cost is left as a future work.

III. Preliminaries and Problem Formulation

A. Synchronous Dataflow Graph

A *synchronous dataflow graph* (SDFG) is a directed graph $G = \langle V, E\rangle$, in which V is the set of actors, modeling the computations of the system; E is the set of directed edges, modeling dependencies between computations. 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*, resp. The source actor and sink actor of e are denoted as $src(e)$ and $snk(e)$, resp. If $prd(e) = cns(e) = 1$ for each $e \in E$, then G is a *homogeneous SDFG* (HSDFG).

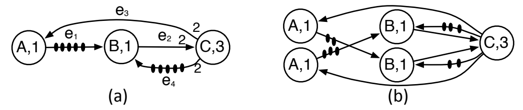

Fig. 2. (a) SDFG G_1; (b) the equivalent HSDFG of G_1. The sample rates are omitted when they are 1.

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,

$$q(src(e)) \cdot prd(e) = q(snk(e)) \cdot cns(e). \tag{1}$$

Eqn. (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, the balance equations for the graph shown in Fig. 2 (a) yield the repetition vector $q = [2, 2, 1]$ corresponding to actors A, B and C.

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. 2(a), for example, includes two firings of actors A and B and one of C. 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 [6]. Fig. 2 (b) is the equivalent HSDFG of G_1.

A sample rate consistent SDFG is *deadlock-free* if there is no zero-delay cycle in its equivalent HSDFG. Only sample rate consistent and deadlock-free SDFGs are meaningful in practice. Therefore we consider only such SDFGs.

B. Scheduling and Optimization

A *static schedule* arranges computations of an algorithm to be executed repeatedly according to a schedule cycle. A static schedule of an SDFG is in fact a schedule of its unfolded graph with an *unfolding factor* f, i.e., a *schedule cycle* contains f consecutive iterations of the SDFG.

Definition 1. A *static schedule* of SDFG G with unfolding factor f is a function $S(G, f)$ defining the time arrangement and the processor allocation of firings periodically according to a period CP. For the ith firing of actor v, denoted by (v, i), $i \in [1, \infty)$:

1) $S(v, i).st$ is the start time of (v, i), which implies there should be sufficient tokens on each $e \in InE(v)$ for a firing of v at the moment $S(v, i).st$;
2) $S(v, i).pa$ is the processor assigned to (v, i), which should be available at the moment $S(v, i).st$;
3) $S(v, i + f \cdot q(v)).st = S(v, i).st + CP$; and
4) $S(v, i + f \cdot q(v)).pa = S(v, i).pa$.

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. For conciseness, we may omit the parameters of S when clear in the context. The definition of static schedule extends the definition in [11] and [12] with 2) and 4) to express the processor allocation.

The *iteration period* (IP) of a static schedule S, denoted by $S.IP$, is the average computation time of an iteration, i.e., $S.IP = \frac{CP}{f}$. When G is arranged to run according to schedule S, certain resources are required. We call the number of processors used by S the *processor requirement* of S, denoted by $S.pRe$. Let $S.mRe(e)$ be the buffer size required by edge e of G in S. The *buffer requirement* of S is an integer vector $S.mRe(E)$. The sum of all elements of $S.mRe(E)$ is the storage space of S. The degree of auto-concurrency is defined as the maximal number of auto-concurrent firings of each actor, denoted by vector $S.aRe(V)$. In summary, given a static schedule, its *IP*, *pRe*, *mRe* and *aRe* are decided.

The *iteration bound* (IB) of SDFG G, denoted by $G.IB$, is the greatest lower bound of the IPs of schedules of G. If $S(G, f).IP = G.IB$, then S is a *rate-optimal schedule* and f is an *optimal unfolding factor* of G. The IB of an HSDFG is given by its *maximum cycle mean* [6].

The IB of an SDFG equals the IB of its equivalent HSDFG. For example, the IB of G_1 in Fig. 2 (a) is $\frac{5}{2}$, which can be computed by the maximum cycle mean of its equivalent HSDFG shown in Fig. 2 (b). There are methods to directly compute the IB on an SDFG, e.g. as an eigenvalue of a specific max-plus matrix and through symbolic simulation [29].

Retiming is a graph transformation technique that redistributes the graph's initial tokens while its functionality remains unchanged [7], [2]. Retiming an actor once means firing this actor once. Given an SDFG $G = \langle V, E\rangle$, a *retiming* of G is a function $r : V \rightarrow \mathbb{Z}$, specifying a transformation r of G into a new SDFG $r(G) = \langle V, E_r\rangle$, where the delay-function d_r is defined for each edge $e = \langle u, v\rangle$ by the equation: $d_r(e) = d(e) + prd(e)r(u) - cns(e)r(v)$. A retiming r of a consistent and deadlock-free SDFG is *legal* if the retimed graph $r(G)$ is also deadlock-free. Only legal retimings are meaningful. Note that retiming does not affect the IB of an SDFG, that is, $r(G).IB = G.IB$.

C. Problem Formulation

When there are unlimited resources available, a static rate-optimal schedule can always be found [5]. Schedules may however slow down when resource limitations are introduced. Depending on the resource limitations considered, we have to find for a graph G a legal retiming r, an unfolding factor f and a static schedule $S(r(G), f)$, such that:

1) $S.IP = G.IB$;
2) $S.pRe <= P \wedge S.mRe \preccurlyeq M \wedge S.aRe \preccurlyeq N$ for given integer P, integer vectors $M(E)$ and $N(V)$, where $X \preccurlyeq Y$ means that $X(e) \leq Y(e)$ for each e.
3) $S.IP = G.IB$ and $S.pRe$ are as small as possible;
4) $S.IP = G.IB$ and $\sum_{e\in E} S.mRe(e)$ are as small as possible.

Our solution for the first problem (no resource limitations) is exact, which guarantees that the returned schedule is rate-optimal. Our solution for the second problem returns a schedule satisfying the requirement but it may not be the fastest one under the constraints. Our solutions for the last two problems are heuristics, which return rate-optimal schedules with resources requirements that are not necessarily optimal. None of our solutions guarantee the smallest unfolding factors.

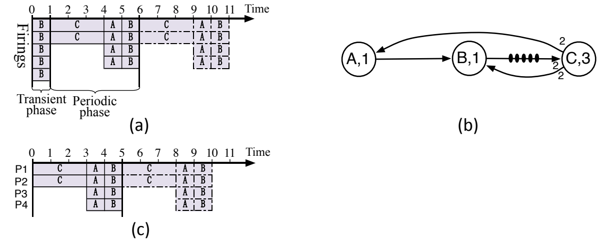

Fig. 3. (a) The actor firing sequence obtained by the STE of G_1; (b) a retimed SDFG of G_1, with $r(B) = 5$ and $r(A) = r(C) = 0$; (c) a rate-optimal schedule of the retimed SDFG with unfolding factor 2.

IV. Basic Idea by an Example

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 [2], [7]. We limit ourselves to strongly connected SDFGs to develop our ideas. The concept can be generalized to other graphs.

The IB of an SDFG can be obtained by exploring the state space generated by an STE [29], in which actors must fire as soon as possible (ASAP) [6]. 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 the IB of the SDFG. For example, Fig. 3 (a) shows the firing sequence obtained by an STE of G_1 of Fig. 2 (a). At time 6, the number of tokens on each edge is the same as that at time 1. Thus the firings enabled are the same at these two time points, i.e., two firings of C. The firings of actors between time 1 and 6 will be repeated indefinitely. 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 equal to the IB of its equivalent HSDFG.

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 a time displacement of 5 time units and the number of firings of each actor v in each repetition is $2q(v)$. The firings in the periodic phase form exactly a rate-optimal schedule with unfolding factor 2 and schedule 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.

Modeling the available processors by a set, we can get a mapping from firings to processors. When a firing starts, a processor is used and removed from the set; when the firing ends, the processor is released and put back to the set. In this way, we arrange each firing on a fixed processor. Using the STE from Fig. 3(a), the retiming $r = [0, 5, 0]$ is obtained. The retimed graph $r(G_1)$ is shown in Fig. 3(b). A rate-optimal schedule of $r(G_1)$, shown in Fig. 3(c), is the firing sequence appearing in the periodic phase of the STE in Fig. 3(a).

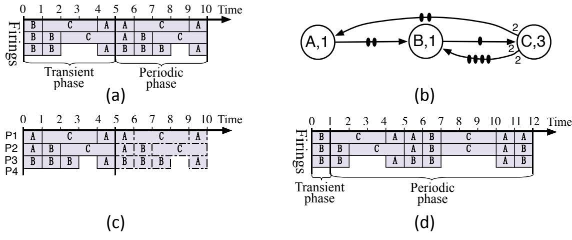

Fig. 4. (a) Actor firing sequence obtained by a constrained STE of G_1, in which the number of concurrent firings is limited to 3; (b) a retimed SDFG of G_1, with $r(A) = r(C) = 2$ and $r(B) = 5$; (c) a rate-optimal schedule with 3 processors. (d) another constrained STE of G_1 with the number of concurrent firings limited to 3, which leads to a schedule with $IP = \frac{11}{4} > IB$.

The STE also reveals the number of concurrent firings at each point in time. Since a firing occupies one processor, the maximum number of concurrent firings in the periodic phase of the STE implies the processor requirement of a rate-optimal schedule, which is 4 in our example. Limiting the number of concurrent firings in an STE may reduce the processor requirement of a schedule. For example, when there are only 3 processors available, we can limit the number of concurrent firings of G_1 to 3 and then get a constrained STE as shown in Fig. 4(a). The retimed SDFG obtained is shown in Fig. 4(b) and a rate-optimal schedule for the retimed graph is shown in Fig. 4(c). However, an STE with constraints may no longer be deterministic, because during execution a choice may have to be made among the enabled firings due to resource limits. Fig. 4(d) shows another STE of G_1 with the same constraint, which leads to a schedule with $IP = \frac{11}{4} > IB$. We come back to this in Section VII-B. Similarly, we can also find buffer sizes needed by each edge at each time point in the STE. The sum of the maximum buffer sizes of all edges in each state of the periodic phase of the STE is the buffer size requirement of the rate-optimal schedule delivered by the STE. The more concurrent firings of an actor, the larger buffer size needed by its outgoing edges. By limiting the number of concurrent firings of the STE to a given constraint, we can check whether there exists a schedule with given buffer size limitation.

V. Self-Timed Execution

This section formalize the definition and properties of the self-timed execution. We assume here that there are no resource constraints. The extension of the semantics with constraints is presented in Section VII.

A. An Operational Semantics of SDFGs

We define the behavior of an SDFG G in terms of a *labeled transition system* (LTS) [32], 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 according to different actions.

We use a vector $tn(E)$ to model the change of the delay distribution of G in the execution. Element $tn(e)$ is the current number of tokens on edge e. The SDFG model of computation allows simultaneous firings of an actor (auto-concurrency). For different concurrent firings of an actor, the one first to start is the one first to end. We use queues $tr(v)$ and $op(v)$ to

TABLE I
Formal Semantics

Acronym	Formal Semantics*
$readyS(v)$	$\forall e \in InE(v) : tn(e) \geq cns(e)$
$sFiring(v)$	$(\forall e \in InE(v) : tn'(e) = tn(e) - cns(e)) \wedge ps' = ps \backslash \{\min(ps)\}$ $\wedge tr'(v) = tr(v) \cup \{t(v) \wedge op'(v) = op(v) \cup \{\min(ps)\}\}$
$readyE(v)$	$hQ(tr(v)) = 0$
$eFiring(v)$	$(\forall e \in OutE(v) : tn'(e) = tn(e) + prd(e)) \wedge ps' = ps \cup \{hQ(op(v))\}$ $\wedge tr'(v) = dQ(tr(v)) \wedge op'(v) = dQ(op(v))$
$readyClk$	$\forall v \in V : \neg readyS(v) \wedge \neg readyE(v)$
clk	$(\forall v \in V : \neg(tr(v) = \emptyset) \Rightarrow \forall x \in tr(v) : x' = x - mS)$ $\wedge glbClk' = glbClk + mS$

* $tn'(e)$, $tr'(v)$, $op'(v)$ and ps' refer to the value of $tn(e)$, $tr(v)$, $op(v)$ and ps in the new state, resp. For conciseness, we omit the elements of states if their values remain unchanged. $\min(ps)$ returns minimal value of ps. $hQ(qu)$ and $dQ(qu)$ returns and removes the first element of queue qu, resp. $mS = \min_{c \in \bigcup_{v \in V} tr(v)} c$.

contain the remaining times and the occupied processors of the concurrent firings of actor v, resp. For the ith unfinished firing of v, its remaining time and the processor it uses are captured by the ith element of $tr(v)$ and $op(v)$, resp.

Our purpose is to construct fast schedules, so we use a global clock, denoted by *glbClk*, to monitor the time progress. A set ps is used to contain the indices of available processors. Set ps may be infinite if there are no processor constraints.

A *state* of the $LTS(G)$ is a 3-tuple that consists of the values of $tn(E)$, $tr(V)$ and $op(V)$. The *initial state* of $LTS(G)$ is denoted as s_0. In the initial state of $LTS(G)$, $tn(E)$ is the initial delay distribution $d(G)$; no firings have been started, so each element of $tr(V)$ and $op(V)$ are empty. That is $s_0 = (d, \emptyset, \emptyset)$. 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 predicates $readyS(v)$ and $readyE(v)$ as their enabling conditions, resp. In parallel with actor firing, time elapses, represented by the increase of the global clock *glbClk*. A time step is modeled by action *clk*. All actions are formalized in Table I. Note that elements in $tr(v)$ and in $op(v)$ are always inserted or removed synchronously, therefore at each state their sizes are the same. The guard $readyS(v)$ tests if there are sufficient tokens on the incoming edges of actor v to enable a firing. When v starts a firing, action $sFiring(v)$ consumes the tokens on its incoming edges according to the consumption rates and occupies a processor. Without loss of generality, assume that a new firing always takes a processor with minimal index in ps. When its remaining time is zero, the firing is ready to end. This is modeled by the predicate $readyE(v)$. When v ends a firing, $eFiring(v)$ produces tokens to its outgoing edges according to the production rates and releases the occupied processor. The released processor is put back to ps. Time progresses as much as possible when no actor is ready to end or to start. The enabling condition for time step, *readyClk*, guarantees an ASAP execution of an SDFG. The largest possible time step, denoted by mS, is the minimal element of $tr(v)$ for all v. A time step reduces the remaining times of all firings by mS and increases the global clock by mS. The elements in $tr(v)$ are decreased by mS only when $tr(v)$ is not empty and mS is the minimal element in all $tr(v)$. This guarantees that a *clk* action never leads to a negative value in $tr(v)$. The delay distribution remains unchanged by *clk*. An *action* of $LTS(G)$ is any of the $sFiring(v)$, $eFiring(v)$ and *clk* actions. A *transition* from state

to state of $LTS(G)$ is caused by any of its actions guarded by their enabling conditions. An *execution* of an SDFG G is an infinite alternating sequence of states and transitions of $LTS(G)$. We use actions to represent transitions which they cause. By the definition of the enabling condition of a time step we restrict an execution to a *self-timed execution* (STE) [33]. The STE can be used to derive a rate-optimal schedule.

The semantics defined above extends the semantics in [11] by adding $op(V)$ to capture the processor occupation during the execution of the SDFG. The vector used for recording the storage requirement defined in [12] is removed. We show later that it can be computed by the variables we introduced above. The semantics is based on the assumption that there are no resource constraints. Subsequent sections show how to extend the semantics to get executions under constraints.

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 because of the enabling condition of a time step. The state at each time point is therefore unique according to this order. Self-timed behavior is deterministic if we only consider its effects on the actor firing sequence [29]. 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 $tn(E)$, $tr(V)$ and $op(V)$ are finite; therefore, the STE includes finitely many states that are distinct and ultimately goes into a repetitive pattern.

Property 1. 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 in σ to make $hasCycle(\sigma)$ true are denoted by s_b and s_e, resp. For example in the STE shown in Fig. 3 (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 by Property 1; beginning with the initial state s_0 and ending at state s_e. We directly call such a finite state sequence an STE in the remainder of the paper.

In an STE σ, the subsequence from s_b to s_e forms its periodic phase, denoted by σ_p; the subsequence from s_0 to the translation before s_b forms its transient phase, denoted by σ_t. The time that passes in σ_p is

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

Let $\sigma : s_0 \rightarrow s_1 \rightarrow ... \rightarrow s_i \rightarrow ...$ and $\sigma' : s'_0 \rightarrow s'_1 \rightarrow ... \rightarrow s'_i \rightarrow ...$ be the STEs of an SDFG obtained by the semantics defined above and by the semantics defined in [11] (i.e., the semantics without the $op(V)$ vector of occupied processors), resp. According to the semantics defined above, vector $op(V)$ has no effect on the behavior of an SDFG, because none of the enabling conditions depend on its values. Therefore, for each $i \in [0, \infty]$, $s_i|_{(tr,tn)} = s'_i$, where $s_i|_{(tr,tn)}$ is a projection of s_i on (tr, tn). Let $\sigma|_{(tr,tn)}$ be the sequence containing all $s_i|_{(tr,tn)}$. It is obvious that $\sigma' = \sigma|_{(tr,tn)}$. Based on this analysis and the fact that the value of $op(V)$ is finite, the following properties hold for our new semantics.

Property 2. The periodic phase of an STE consists of a whole number of iterations [29].

The number of iterations in σ_p is denoted by $nIter(\sigma_p)$. For each actor v, the numbers of $sFiring(v)$ and $eFiring(v)$ actions in σ_p are both $nIter(\sigma_p) \cdot q(v)$. Then the iteration period in the periodic phase is: $IP(\sigma_p) = \frac{CP(\sigma_p)}{nIter(\sigma_p)}$.

Compared with σ'_p, σ_p may contain more iterations, because a state in σ_p includes one more vector, $op(V)$, and the $hasCycle(\sigma)$ checking needs to compare not only the equality of tr and tn but also the equality of op when comparing states. There exists an integer $N \geq 1$ such that $nIter(\sigma_p) = N \cdot nIter(\sigma'_p)$ and $CP(\sigma_p) = N \cdot CP(\sigma'_p)$. N can be large depending on the number of combinations of the values of tr, tn and op. This in turn leads to a large unfolding factor.

Property 3. Given the STE σ of SDFG G, the iteration period of σ_p equals the IB of G. That is $IP(\sigma_p) = G.IB$ [29].

In summary, the addition of vector op does not affect the behavior of an SDFG and it does not violate some of the important properties of an STE [29], [11]. The addition does however lead to an improvement in processor and memory usage of our methods. Because the number of iterations in σ_p may be larger than those in σ'_p, the optimal unfolding factors may however be larger than that of [11]. Our results in Section IX confirm this observation.

VI. Rate-Optimal Scheduling

In this section, we first present an algorithm to explore the state-space of an SDFG according to its self-timed execution; then we introduce our rate-optimal scheduling algorithm. The algorithm schedules SDFGs under the assumption that there are unlimited resources available and actors are allowed to fire concurrently as many times as possible. We consider the construction of schedules under constraints in the next section.

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: ending all firings ready to end, starting all firings ready to start, and then a time step. Algorithm 1 returns the STE of an SDFG. At Lines 2 and 3, the current state is set to the initial state of the LTS of G and is stored as the first element of σ. Lines 4 to 20 explore the state space according to the semantics defined in Section V-A and form the STE. The information of start firings is related with scheduling, so we store *sFiring* actions and the states after them at Lines 13 and 14, reps. The termination of Algorithm 1 is guaranteed by Property 1. Although we only consider deadlock-free SDFGs, deadlock can also be detected by Algorithm 1 by checking *glbClk*. When deadlock occurs, the execution will halt on the action *clk* and *glbClk* will go to

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:  σ ← s
4:  while not hasCycle(σ) do
5:     for all v ∈ G do
6:        while readyE(v) do
7:           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:          σ ← σ + s
15:       end while
16:    end for
17:    if readyClk then
18:       clk
19:    end if
20: end while
21: return σ
```

infinite [29]. For an efficient implementation of Algorithm 1, it is not necessary to check $hasCycle(\sigma)$ after every iteration of the loop (Lines between Line 4 and Line 20) but it is sufficient to check for a recurrent state after every completion of one specific arbitrarily chosen actor firing in an iteration.

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. [2] proves that it is sufficient to check if the initial delay distribution of the retimed graph is nonnegative to ensure that a retiming is legal. Hence, the transient phase of the STE forms a legal retiming.

Theorem 1. 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. Recall the definition of $sFiring(v)$. When a firing starts, it takes a processor from ps and push it to the end of $op(v)$. So, the last element of $op(v)$ in the state caused by a $sFiring(v)$ is exactly the processor used by the firing.

Theorem 2. Given the STE σ of an SDFG G and the retiming obtained from it, r, schedule $S(r(G), nIter(\sigma_p))$ defined below is rate-optimal: for each actor $v \in G$ and $1 \leq i \leq f \cdot q(v)$,

$$S : \begin{cases} S(v,i).st = sFiring(v, i + r(v)).glbClk - s_b.glbClk, \\ S(v,i).pa = lQ(s(v, i + r(v)).op(v)), \end{cases} \tag{2}$$

where $sFiring(v, j)$ is the jth firing of v in σ, $s(v, j)$ is the state caused by it, and $lQ(qu)$ returns the end element of queue qu.

For SDFG G_1 (Fig. 2(a)), the retiming obtained by its STE (Fig. 3(a)) is $r(B) = 5, r(A) = r(C) = 0$; the retimed graph is shown in Fig. 3(b). The optimal unfolding factor is 2 and Fig. 3(c) is the rate-optimal schedule for the retimed graph.

Consider the rate-optimal schedule shown in Fig. 3(c). Concurrent firings need to occupy different processors. When there are 4 concurrent firings, the number of processors needed at this point is 4. If we can find the maximal number of concurrent firings in the periodic phase of an STE, we know how many processors are required in a rate-optimal schedule. The vector $op(V)$ records concurrently occupied processors by actors. Then at state s, the number of processors occupied by actor v is $|s.op(v)|$, where $|s.op(v)|$ represents the size of queue $s.op(v)$. The processor requirement of schedule S is:

$$S.pRe = \max_{s \in \sigma_p} \sum_{v \in V} |s.op(v)|. \tag{3}$$

In line with [16], we choose a relatively conservative storage abstraction to leave more room for implementation. That is, when one actor starts firing, it claims the space for the tokens it will produce, and it releases the space of the tokens it consumes only when the firing ends. Under this assumption, at each state, the required buffer size of edge e, denoted by $tnb(e)$, is decided by the number of tokens on e and the numbers of firings of its source actor and sink actor.

$$tnb(e) = tn(e) + |tr(src(e))| \cdot prd(e) + |tr(snk(e))| \cdot cns(e). \tag{4}$$

We consider separate memories for each edge. At each schedule point, edge e requires at least buffer size $tnb(e)$. Therefore the maximal $tnb(e)$ in the periodic phase of an STE forms the buffer requirement for each e in a schedule S obtained from the STE, denoted by $S.mRe(e)$.

$$S.mRe(e) = \max_{s \in \sigma_p} s.tnb(e), \text{ for all } e \in E. \tag{5}$$

The degree of auto-concurrency of schedule S is:

$$S.aRe(v) = \max_{s \in \sigma_p} |s.tr(v)|, \text{ for all } v \in V. \tag{6}$$

An algorithm for static rate-optimal scheduling of SDFGs is shown in Algorithm 2. Lines 2 - 6 accumulate the number of start firing actions in the transient phase to form a retiming. Lines 8 - 14 compute the schedule. The correctness of the algorithm is guaranteed by Theorems 1 and 2.

The schedule computed using our method is fully static [5]. It schedules f iterations as a schedule cycle. The same firing in different schedule cycles uses the same processor. The use of retiming allows our algorithm to construct a rate-optimal schedule in which the start time of all firings in the first schedule cycle are not larger than CP, i.e., $S(v,i).st \leq CP$ for $i \leq f \cdot q(v)$. This implies that all firings that belong to the one iteration of the schedule cycle will be started before the first

Algorithm 2 *rOptSch*(G)

Require: A strongly connected SDFG G.

Ensure: A legal retiming r, an unfolding factor f, a rate-optimal schedule $S(r(G), f)$, $S.IP$, $S.pRe$, $S.mRe$ and $S.aRe$.

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:     $S(v, i(v)).st = a.glbClk - s_b.glbClk$
11:     $S(v, i(v)).pa = lQ(s(v, i(v)).op(v))$
12:     $i(v) = i(v) + 1$
13:   **end if**{by Theorem. 2}
14: **end for**
15: $f = nIter(\sigma_p)$
16: $IP = \frac{CP(\sigma_p)}{f}$
17: get pRe, mRe and aRe by Eqns. (3), (5) and (6), resp.
18: **return** r, f, S, IP, pRe, mRe and aRe

firing in the next iteration of the schedule cycle. Alternative rate-optimal scheduling techniques presented in [9] and [15] do not have such a property.

VII. Scheduling under Constraints

As discussed in the previous section, the number of concurrent firings affects the resource requirement of a schedule. If we can limit the number of concurrent firings in an STE according to different constraints, we can find a schedule of the SDFG or its retimed graph under these constraints. In this section, we first present a unified framework for scheduling SDFGs under constraints. Considered constraints are formalized subsequently.

A. General Scheduling Framework

When there are infinite resources available, actor firings occur as soon as possible in a STE. When some enabled firings are blocked because of insufficient resources, we call the STE a *constrained STE*. When $x = p$, m, a or mC, let CON_x be the formula capturing the constraint on the number of processors, the buffer sizes, auto-concurrency or their combination. Constraint CON_x can be modeled as a limitation of concurrent firings in an STE as we explain in the following sections; therefore it can be put on the guard of *sFiring* actions. We denote the new guard as $readyS_x(v)$.

$$readyS_x(v) \equiv_{def} \quad readyS(v) \wedge CON_x(v).$$

The definitions of actions and other guards remain unchanged.

The procedure to obtain such an x-constrained STE of G is denoted by $STE_x(G, X)$, where the given value $X = P$, M, N or $[P, M, N]$ correspond to $x = p$, m, a or mC, resp. Procedure $STE_x(G, X)$ is a variation of Algorithm 1, in which $readyS(v)$ is replaced with $readyS_x(v)$.

The procedure to obtain the retimed graph and its schedule under a constraint x is called $Sch_x(G, X)$. It is a variation of Algorithm 2, in which the $STE(G)$ is replaced with $STE_x(G, X)$. Different from the STE, the IP of the periodic phase of an STE_x no longer always equals the IB. Hence, the schedule returned by Sch_x is not necessarily rate-optimal.

A general framework for scheduling SDFGs under constraints is summarized in Table II, which includes three parts. The first part is the resource requirement limitations under different constraints. For example, when a processor constraint is considered ($x = p$), the processor requirement of the targeted schedule (pRe) is not allowed to exceed P. The second part includes the enabling condition (Cond.) of action *sFiring*, the name of the execution used to deliver a proper schedule (Exec.) and the scheduling procedure (Sch.) under different constraints. The third part shows the possible IPs of schedules under different constraints, clarifying whether the schedule is guaranteed to be rate-optimal.

TABLE II
A General Framework for Scheduling SDFGs

Cons.	Rate opt.	Proc. ($x = p$)	Mem. ($x = m$)	Auto-con. ($x = a$)	Multi. ($x = mC$)
pRe	∞	P	∞	∞	P
mRe	∞	∞	M	∞	M
aRe	∞	∞	∞	N	N
Cond.	$readyS$	$readyS_p$	$readyS_m$	$readyS_a$	$readyS_{mC}$
Exec.	STE	STE_p	STE_m	STE_a	STE_{mC}
Sch.	$rOptSch$	Sch_p	Sch_m	Sch_a	Sch_{mC}
IP	$= IB$	$\geq IB$	$\geq IB$	$\geq IB$	$\geq IB$

The key factor that distinguishes scheduling procedures under different constraints is the formula CON_x. We discuss how it is constructed and why it leads to a x-constrained schedule in the following subsections.

B. Processor Constraint

As defined in Section V-A, the available processors are modeled by the set ps. When there are no processor constraints, ps is infinite. When only P processors are available, the size of ps is set to P, where $P \geq 1$. While ps is empty, all processors are occupied by firings. Therefore, the condition to test whether there is a processor available for a firing to start is defined as:

$$CON_p(v) \equiv_{def} \neg(ps = \emptyset).$$

The condition $CON_p(v)$ guarantees that at each state of the p-constrained STE, the number of processors occupied does not exceed P; therefore the resulting schedule uses no more than P processors. For example, Fig. 4(a) is a p-constrained STE of G_1 with $P = 3$. By procedure $Sch_p(G, 3)$, we get a retimed graph and its schedule, shown in Fig. 4(b) and (c), resp. Fig. 5 is a part of Fig. 4(a) with detailed states and transitions. At state S_1, there are sufficient tokens on $\langle A, B\rangle$ and $\langle C, B\rangle$ for *five* concurrent firings of B. The number of available processors, however, is only *three*. Hence, only three firings of B are allowed to start.

The schedule obtained by $Sch_p(G, P)$ is not always rate-optimal. In fact, there is a lower bound on the number

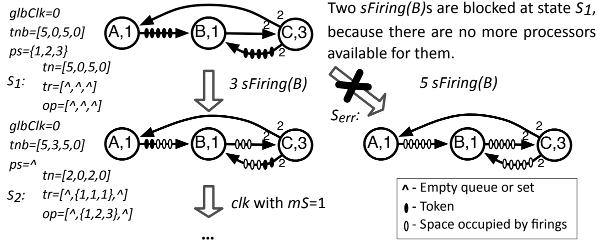

Fig. 5. A part of the p-constrained STE of G_1 with $P = 3$.

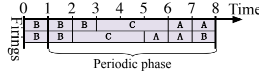

Fig. 6. Actor firing sequence from a p-constrained STE of G_1 with $P = 2$.

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 by $sumT$. 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 $sumT$ and the IB. That is, the lower bound on the number of processors for a rate-optimal schedule is

$$lbP = \left\lceil \frac{\sum_{v \in V} t(v)q(v)}{IB} \right\rceil. \tag{7}$$

For example, lbP for G_1 is 3, i.e., with less than 3 processors available, no rate-optimal schedule exists for G_1 or any retimed graph of it. A p-constrained STE with $P = 2$ of G_1 is shown in Fig. 6. It delivers a schedule with $IP = \frac{7}{2}$ which is larger than the IB of G_1 ($\frac{5}{2}$). There is no schedule faster than it because the 2 processors are fully used.

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 p-constrained STE with P processors, at a macro step, the sum of simultaneously ready firings and concurrent firings may be more than P and these ready firings may be of different actors. Which ones are chosen to start may affect the state space of the constrained STE and the IP of its periodic phase. Therefore the schedule returned by $Sch_p(G, P)$ may not be the fastest one under processor constraint P. See Fig. 4 (a) and (d) for example. At time point 6, there are sufficient tokens for *three* firings of B and *one* firing of C to start, but one of them has to be blocked because of the processor constraint $P = 3$. If the priority of C is higher than that of B, the p-constrained STE is shown in Fig. 4 (a); otherwise the p-constrained STE is another one shown in Fig. 4 (b). The IP of schedule obtained by the former is $\frac{5}{2}$, while that of the latter is $\frac{11}{4}$. We use a fixed random order of the firings in our implementation. A procedure to find a rate-optimal schedule with as few processors as possible is presented in Section VIII.

C. Memory Constraint

If the storage space of a schedule is constrained by a vector $M(E)$, limiting the buffer size of each edge, then an enabled firing can only start when there is sufficient space on its outgoing edges. For example, Fig. 7 is a part of a

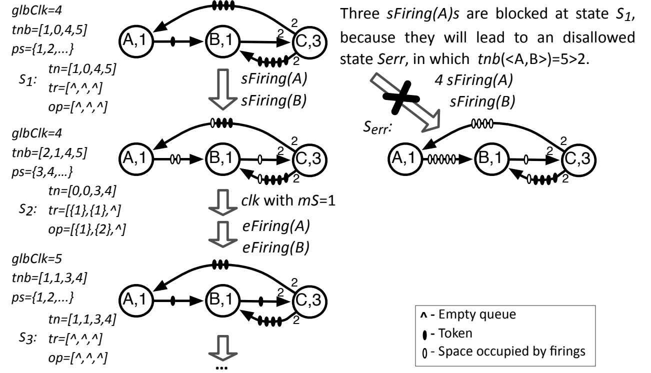

Fig. 7. A part of the m-constrained STE of G_1 with $M = [2, 4, 4, 5]$, corresponding to $\langle A, B\rangle$, $\langle B, C\rangle$, $\langle C, A\rangle$ and $\langle C, B\rangle$.

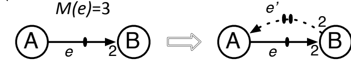

Fig. 8. An example of e' in Lemma 1.

m-constrained STE of G_1 with $M = [2, 4, 4, 5]$. At state S_1, tokens on edges are available for actor A to fire *four* times and B *once*. However, the buffer size of edge $\langle A, B\rangle$ is limited to 2, leaving space only enough for tokens that will be produced by *one* firing of A. Thus the *three* other firings of A are blocked. The condition to test whether there is sufficient storage space available on the outgoing edges of actor v is defined as:

$$CON_m(v) \equiv_{def} \forall e \in OutE(v) : prd(e) \le M(e) - tnb(e),$$

where $tnb(e)$ is the already occupied and claimed space on edge e, defined by Eqn.(4).

Similar to procedure $Sch_p(G, P)$, the schedule obtained by $Sch_m(G, M)$ is not always rate-optimal. For a deadlock-free G, even when there is only one processor available, the p-constraint STE will never go to a deadlock state. In the m-constrained STE, however, it is not the case. Insufficient storage space may eventually prevent any actor to start firing. Therefore, there may be no valid schedule of G under the memory constraint M. In this case, the periodic phase of the m-constrained STE contains only one action, the time step *clk*; its transient phase can still form a legal retiming.

Because of our assumption on the storage model, in an m-constrained STE, the order of concurrent *sFiring*s is irrelevant. A different order leads to the same state. At state S_1 in Fig. 7, for example, no matter $sFiring(A)$ or $sFiring(B)$ occurs first, once all firings are started, the resulting state is always S_{err}. Hence, at S_1, three $sFiring(A)$s have to be blocked.

These buffer size constraint can also be modeled by adding an incoming edge with tokens to model available storage space [16]. Therefore, an m-constrained STE of an SDFG is in fact an STE of another SDFG in which a reverse edge with proper initial tokens is added for each edge. An example of this graph transformation is shown in Fig. 8.

Lemma 1. An m-constrained-STE of SDFG $G = \langle V, E\rangle$ with memory constraint $M(E)$ is an STE of SDFG $G' = \langle V, E \cup E'\rangle$, in which $E' = \{\langle v, u\rangle | \langle u, v\rangle \in E\}$ and for each $e' = \langle v, u\rangle \in E'$: $d(e') = M(e) - d(e)$, $prd(e') = cns(e)$ and $cns(e') = prd(e)$.

By Lemma 1, under buffer constraint M, no other schedules will be faster than the schedule derived by $STE_m(G, M)$.

Theorem 3. Procedure $Sch_m(G, M)$ is sufficient and necessary for a fastest schedule under memory constraint M.

A procedure to find a rate-optimal schedule with as little storage space as possible is presented in Section VIII.

D. Auto-Concurrency Constraint

The constraint on the number of auto-concurrent firings of each actor v to $N(v)$ can be represented by limiting the size of $tr(v)$:

$$CON_a(v) \equiv_{def} \quad |tr(v)| < N(v).$$

The constraint CON_a on G has the same result for an ASAP execution as a new graph in which each actor v is added a self-loop edge with $N(v)$ initial tokens. Therefore the a-constrained STE is as fast as the STE of the new graph. Under auto-concurrency constraint N, no other schedules are faster than the schedule returned by $Sch_a(G, N)$.

Theorem 4. Procedure $Sch_a(G, N)$ is sufficient and necessary for a fastest schedule under auto-concurrency constraint N.

E. Multi-Constraint

Multiple constraints can be considered simultaneously by combining them as enabling conditions of action *sFiring*. This gives a multi-constrained STE through which a multi-constrained schedule can be obtained. When all above mentioned constraints are considered, CON_{mC} can be defined as:

$$CON_{mC}(v) \equiv_{def} \quad CON_p(v) \wedge CON_m(v) \wedge CON_a(v).$$

Procedure $Sch_{mC}(G, [P, M, N])$ does not always return the fastest schedule under the constraints because when a processor constraint is considered, the different choices for the firing order affect the IP of the resulting schedule.

VIII. Rate-Optimal Scheduling with Minimal Resources

A. Processor Optimal Scheduling

Suppose a rate-optimal schedule uses nP processors. The closer nP to lbP (Eqn. (7)), the more occupied processors are. We define the *rate of processors used* by a schedule as $\frac{lbP}{nP}$. The rate becomes higher when fewer processors are used by a schedule. We use this rate in Section IX to measure the processor utilization of a schedule.

Algorithm 3 is a heuristic algorithm which tries to find a rate-optimal schedule using as few processors as possible.

Algorithm 3 $rOptSch_p(G)$

Require: A strongly connected SDFG G
Ensure: A legal retiming r, a rate-optimal schedule S of $r(G)$ and $S.pRe$

1: Get the IB and pRe from $rOptSch(G)$ {According to Property 3 and Eqn. (3)}
2: Perform a binary search over $[lbP, pRe]$; assuming P is the number of processors considered, test whether the IP returned by $Sch_p(G, P)$ equals IB
3: **return** r, S and pRe obtained by $Sch_p(G, P)$

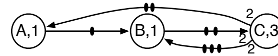

Fig. 9. Retimed G_1 allowing rate-optimal schedule with buffers $[2, 4, 4, 5]$.

Let S be the schedule returned by $Sch_p(G, P)$. $S.pRe$ does not linearly decrease with P. That is, for two integers P_1 and P_2 with $P_1 > P_2$, pRe returned by $Sch_p(G, P_1)$ may be less than that returned by $Sch_p(G, P_2)$. The reason is that the feasibility check for P in the binary search on Line 2 might result in a false negative (as explained in Section VII-B). Hence, Algorithm 3 might lead to a suboptimal result. Nevertheless, our experimental results show that it helps to reduce the number of processors used by a rate-optimal schedule. The binary search can be replaced with a linear search starting from lbP which might lead to a further reduction in the number of processors at the cost of a longer execution time.

B. Memory Optimal Scheduling

As illustrated in Section VII-C, using retiming to shift the initial tokens, we may obtain a retimed graph that can be scheduled with a given buffer constraint. By decreasing the buffer constraint and testing whether a rate-optimal schedule can be found under the constraint, we may find a retimed SDFG whose rate-optimal schedule requires less storage space than the proven minimal storage requirement of the original graph. For example, Fig. 9 shows a retimed graph of G_1 that can be rate-optimally scheduled with buffer size requirement $[2, 4, 4, 5]$, which is smaller than $[5, 4, 4, 5]$, the minimal buffer size of the original graph computed by [16].

If the IP returned by $Sch_m(G, M)$ equals the IB of G, we say that M is *feasible* for a rate-optimal schedule of G. We use a binary search per edge on the memory constraints to seek a rate-optimal schedule with minimal storage space. The $mRe(E)$ of an STE records the memory required by a rate-optimal schedule without any resource limitation. Since this is always feasible, we use it as an upper bound on the binary search. The lower bound can be set to $\langle 0, ..., 0 \rangle$ or a storage distribution to avoid deadlock obtained by [16]. The former is too low and the latter itself takes time to compute. In our implementation, we therefore use a compromise between them. Let $LB(E)$ be a vector, in which each $LB(e) = prd(e) + cns(e) - \gcd(prd(e), cns(e))$. A deadlock-free execution uses at least $LB(e)$ buffer size for e [34]. For example, consider edge $e = \langle A, B \rangle$ with $prd(e) = 3$ and $cns(e) = 2$. We have $LB(e) = 4$. Each firing of B consumes two out of three tokens produced by the previous firing of A and then has to wait for the next firing of A, which will increase the number of tokens on e to four. This is one of the firing sequences of A and B that limits the buffer size. We use $LB(E)$ as a lower bound on the binary search.

The procedure is shown in Algorithm 4. It includes $|E|$ binary searches and begins with the memory constraint $optB = mRe$ of $rOptSch(G)$. Each time one edge e is considered and the buffer sizes of other edges remain unchanged. A binary search over $LB(e)$ and $optB(e)$ is used to find the smallest buffer size of e, and then $optB(e)$ is set to the smallest value. After all edges are checked, we get a smallest feasible storage distribution $optB$. The schedule returned by $Sch_m(G, optB)$ is rate-optimal and requires $minB$ for storage.

Algorithm 4 $rOptSch_m(G)$

Require: A strongly connected SDFG G

Ensure: A legal retiming r, a rate-optimal schedule S of $r(G)$ and the size of its storage space

1: Get the IB and mRe from $rOptSch(G)$

2: Let $optB = mRe$

3: **for all** $e \in E$ **do**

4: Perform binary search over $[LB(e), optB(e)]$; assuming x is considered value, let vector M be defined as $M(e) = x$ and $M(e') = optB(e')$ if $e' \neq e$; use $Sch_m(G, M)$ to test whether M is feasible for a rate-optimal schedule and let $optB = M$ if so.

5: **end for**

6: get r, S and mRe by $Sch_m(G, optB)$

7: $minB = \sum_{e \in E} mRe(e)$

8: **return** r, S and $minB$

Algorithm 4 is a heuristic. In general, buffer sizes can not be determined independently from each other. Hence, the results may differ when the order of edges chosen for the search changes. As shown in our experimental results, however, in most cases, Algorithm 4 does return a storage requirement less or equal to the proven minimal feasible storage space returned by [16], which does not use retiming. We use a random edge order in our experiments. Whether an optimal edge order exist for binary search is left as future work.

For a constrained STE σ, its transient phase σ_t may need more processors or storage space than σ_p needs. In a real implementation of an SDFG, if the retiming process is carried out at runtime, the firings corresponding to the retiming r can be arranged to run under the processor and buffer size requirement of S with a slower speed (less concurrent firings) instead of an as soon as possible execution.

IX. Experimental Evaluation

A. Experimental Setup

We have implemented Algorithms 2 (rOpt-m), 3 (rOptP-m) and 4 (rOptM-m) in SDF3 [14]. Their respective versions that do not consider processor allocation, Algorithms 2 and 3 in [11] and Algorithm 1 in [12], are denoted in this paper by rOpt, rOptP and rOptM, resp. The rate-optimal scheduling algorithm that directly works on SDFGs in [15] (GG95), which addresses the same problem as rOpt, has been implemented and its execution time is used to evaluate the performance of rOpt. We have also implemented the method to compute the smallest optimal unfolding factor for HSDFGs in [9] (CS95). Converting an SDFG to its equivalent HSDFG, we compute the smallest rate-optimal unfolding factor of the SDFG by CS95, which is compared with the unfolding factors returned by rOpt and rOpt-m. We compare the storage space for the retimed graphs returned by our method (rOptM and rOptM-m) with the minimal storage requirements of the original graphs computed using the algorithm from [16] (SGB08). We also compare our processor-constrained scheduling method with the methods from [17] and [18]. We have performed experiments on three sets of SDFGs, running on a 2.67GHz

TABLE III
Acronyms and symbols

Acronym	Meaning
MP3	The MP3 playback application [36]
SaRate	The sample rate converter [35]
MaxES	The maximum entropy spectrum analyzer
CEer	The channel equalizer [37]
Satellite	The satellite receiver [3]
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
GG95	The result returned by [15]
CS95	The result returned by [9]
SGB08	The result returned by [16]
rOpt	The result returned by *optSch* (Algorithm 2) of [11]
rOptP	The result returned by *minOptSch* (Algorithm 3) of [11]
rOptM	The result returned by *conOptSch* (Algorithm 1) of [12]
rOpt-m	The result returned by *rOptSch* (Algorithm 2)
rOptP-m	The result returned by $rOptSch_p$(Algorithm 3)
rOptM-m	The result returned by $rOptSch_m$(Algorithm 4)

CPU with 12MB cache. The experimental results are shown in Tables IV, V, VI and VII and discussed in Section IX-B. All execution times are measured in milliseconds (ms).

The first set of SDFGs consists of five practical applications, including a sample rate converter (SaRate) [35], a satellite receiver (Satellite) [3], a maximum entropy spectrum analyzer (MaxES) (http://ptolemy.eecs.berkeley.edu/), an MP3 playback application (MP3) [36] and a channel equalizer (CEer) [37]. 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 outgoing edges and no incoming edges, we convert these models to strongly connected graphs. The second set consists of 540 synthetic strongly connected SDFGs generated with SDF3, mimicking real DSP applications. The number of actors in an SDFG (*nA*) and the sum of the elements in the repetition vector (*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 changed from 0 to 1. To measure the extreme cases we may deal with, we choose two values of *nD*: 0 and 0.9. we generated 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 these parameters. The third set consists of five benchmark graphs from [17]. The acronyms and symbols used in this section are summarized in Table III.

B. Experimental Results

Before discussing the results, we first clarify the differences between the schedules returned by rOpt-m, rOptP-m and rOptM-m and those returned by rOpt, rOptP and rOptM.

TABLE IV
Experimental results for practical DSP examples

Graph Information					
name	MP3	SaRate	MaxES	CEer	Satellite
nA	4	6	13	22	22
nQ	10601	612	1288	42	4515
IB	116424	5.25	5764	47128	1.83
Unfolding Factor					
CS95	N[a]	4	1	1	6
rOpt	1	4	2	1	6
rOpt-m	1	4	2	1	6
Rate of Processors Used					
rOpt(-m)	24.0%	72.6%	0.5%	14.6%	19.4%
rOptP(-m)	24.0%	77.3%	0.5%	43.7%	68.4%
Storage Requirement					
SGB08	N	N	N	73	N
rOptM(-m)	2916	1328	2087	73	15168
Execution Time (ms)					
GG95	N	2,024	580	38,459	3,558,957
rOpt	7	1	1	0	4
rOpt-m	27	2	2	1	19
rOptP	38	15	3	2	127
rOptP-m	111	79	18	2	2,410
rOptM	213	45	55	1	5,807
rOptM-m	548	592	115	2	20,612
Sch_p[b]	16	4	4	0	104
Sch_m	12	12	0	0	48
Sch_{pm}	12	8	4	0	80

[a] no results available because of timeout.
[b] Sch_p uses the minimal number of processors returned by rOptP(-m) as a constraint; Sch_m uses the minimal buffer requirement returned by rOptM(-m) as a constraint; Sch_{pm} uses their combination.

TABLE V
Execution times (ms) for synthetic examples

	nD = 0			*nD* = 0.9			nD,nA / nQ
	a10	a20	a50	a10	a20	a50	
Rate-optimal scheduling without mapping [11], [12]							
GG95	51s	58s	45s	57s	59s	43s	1k-1.5k*
rOpt	0	1	2	1	2	3	
rOptP	4	7	32	45	60	61	
rOptM	12	14	73	161	249	321	
GG95	8m	9m	11m	8m	8m	11m	2k-2.5k
rOpt	1	1	3	9	3	4	
rOptP	7	12	40	119	104	121	
rOptM	22	35	10	467	371	409	
GG95	107m	123m	115m	117m	122m	114m	4k-6k
rOpt	1	2	4	7	3	8	
rOptP	15	24	57	287	50	511	
rOptM	33	104	229	742	223	745	
Rate-optimal scheduling with mapping							
rOpt-m	3	4	8	15	22	23	1k-1.5k
rOptP-m	22	31	90	417	654	225	
rOptM-m	31	35	158	1s	4s	1s	
rOpt-m	5	6	26	270	35	28	2k-2.5k
rOptP-m	37	49	302	11s	852	440	
rOptM-m	70	96	354	29s	39s	1s	
rOpt-m	6	13	25	4s	27	7s	4k-6k
rOptP-m	78	147	290	197s	453	8s	
rOptM-m	85	350	685	425s	1s	21s	

* 1k=1000.

TABLE VI
Unfolding Factors and Improvement for Synthetic Examples

	Unfolding Factor			Rate of Pro. Used		Mem. Imp.
	CS95	rOpt	rOpt-m	rOpt	rOptP(-m)	rOptM(-m)
nD = 0	1.11	1.18	1.51	10.5%	20.1%	6.0%
nD = 0.9	2.18	2.73	8.12	21.7%	40.4%	4.8%

The former include allocation information while the latter schedules do not include it; both require the same number of processors and buffer sizes; however the former may have larger unfolding factors. The common aspects and differences are consequences of the semantics of their STEs (see Section V-B). The advantage of the proposed algorithms (*-m) is that besides determining the start times of actor firings, they also allocate actors to processors. Since the results are identical, we show only one and label them as *(-m), e.g. rOpt(-m).

Table IV summarizes the results for the practical examples. There are five parts in Table IV. The first part is the information on the graphs, including the number of actors (*nA*), the sum of the elements in the repetition vector (*nQ*) and the iteration bound (*IB*). The second part shows the optimal unfolding factor obtained by rOpt and rOpt-m and the smallest one proven in [9] (CS95). The third part measures the rates of processors used by the schedules returned by rOpt and rOptP. The fourth part compares the storage requirements returned by our method (rOptM) with that of SGB08. The last part includes the execution times of different methods. The information and the storage space do not include the dummy actors and edges.

For the practical examples, CS95 and GG95 do not finish on the MP3 playback model in ten hours. For other examples, except for MaxES, the optimal unfolding factors obtained by rOpt and rOpt-m are equal to those computed by CS95. In three examples, rOptP improves the rate of processors used compared to schedules returned by rOpt. For the models which SGB08 can analyze in ten hours, our method reaches the minimal storage requirement of the original graph. The reason for the inefficiency of SGB08 is the amount of auto-concurrency in the SDFG execution. When auto-concurrency is disallowed, SGB08 runs much faster, as shown by the experimental results in [16]. The proposed binary search can cope well with auto-concurrency. The model that takes the longest execution time is the satellite model, which has a relatively large *nA* and *nQ*. Note that a single execution of Sch_x has a small run-time compared to rOptP-m and rOptM-m as they both call Sch_x many times.

From the results for the practical DSP examples, it seems that rOpt is much faster than GG95 and the larger the difference between *nA* and *nQ*, the more efficient rOpt is in comparison with GG95. Our experiments on the synthetic examples confirm this conclusion.

Tables V and VI give the results for the synthetic examples. Each point in Tables V is an average of graphs in the same group. Each point in Table VI is an average of graphs with the same value of *nD*. 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. The number of delays (*nD*) has almost no effect on the speed of GG95. 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 and *nD* = 0.9 in Table VI, resp. When the delay count is larger, the optimal unfolding factor and the rate of processors used is generally larger. Procedure rOptP increases the rate of processors used of rOpt by about 90 percent in both cases. The optimal unfolding factors returned by rOpt

are close to the proven smallest one (CS95). In the cases with *nD* = 0.9, the average of unfolding factors of rOpt-m seems quite a bit larger than that of rOpt. This may be caused by the large number of initial tokens, which increases the number of values of *tn*. The number of combination of values of *tn* and *tr* and *op* is then enlarge the state space of STE. However, the detailed experimental data shows that in 94% of the tested cases, the unfolding factors of rOpt-m are the same or twice that of rOpt; the largest case is 30 times, largely increasing the average; the others are in the order of *three* to *six* times.

The average storage improvements are also shown in Table VI. Of 54%, 44% and 2% of all the tested models for which SGB08 finished within 30 minutes, our method returns storage spaces smaller than, equal to and larger than that returned by SGB08 [16], resp. The average improvement is about 5.4%.

The procedures rOpt-m, rOptP-m and rOptM-m are slower than their respective versions that do not consider processor allocation (rOpt, rOptP and rOptM). The reason is that, in the semantics of SDFGs considered in this paper, a state of an LTS includes one more vector, $op(V)$, than that of [11] and [12]. This makes the state larger and therefore enlarges the state space of STEs, which need longer time to explore. This is also the reason that the unfolding factors of rOpt-m sometimes are larger than that of rOpt. The details are discussed in Section V-B.

The heuristics in [17] return the best results on the processor constrained scheduling problems of HSDFGs. The state-of-the-art design flow in [18] maps and schedules SDFGs on heterogeneous multiprocessor platforms, taking into account communication cost. The scheduler in [18] maps all firings of an actor to one processor and doesn't use retiming or unfolding. We have adapted it to deal with our models. We ran our algorithm (Sch_p) on the five benchmark graphs in [17] and compared our results with [17] and [18] in Table VII. For the five graphs, 22 processor bounds (#P) are checked. For the 22 returned IPs, compared to [17], our method performs better on 8 cases, worse on 5 cases and equal on all others. [17] considers not only the ASAP start time of an actor but also its slack time, leaving more room for the actor to be scheduled, while our method considers only ASAP start times. As a result, our method performs worse on some cases. [17] only deals with integer IPs. This may be the reason that our method performs better on some cases. Our method in all the tested cases returns better IPs than the method of [18] except for the cases when #P=1, since [18] binds firings of an actor on the same processor while we allocate firings on available processors. [18] does not find schedules for some cases. When #P=1, there are no differences between actor allocation and firing allocation. Results shown in Table VII also reveals partially how the IPs deviated as the processor constraints vary. We observe that when #P decreases, the products of #P and the IP may tend to equal from a certain #P. The reason is that when #P is large enough, the parallelism of an SDFG is mainly decided by its structures; when #P decreases to a threshold, the parallelism of the SDFG will mainly decided by #P, and therefore the IP may be dependent on #P linearly.

TABLE VII
EXPERIMENTAL RESULTS OF PROCESSOR-CONSTRAINED SCHEDULING FOR BENCHMARK GRAPHS (HSDFGS) IN [17] (THE IB OF EACH GRAPH IS SHOWN IN BOLD)

Graph name	#P	IP[17]	IP[18]	IP	#P	IP[17]	IP[18]	IP
Second-order section	4	**3**	-	3.5	2	6	-	6
	3	4	-	4	1	12	-	12
Jaumann filter	3	**16**	20	**16**	1	33	33	33
	2	17	20	16.5	-	-	-	-
All-pole filter	3	**14**	19	14.5	1	31	31	31
	2	16	19	15.5	-	-	-	-
16-point FIR filter	8	4	-	4.1	4	8	11	7.8
	7	5	8	4.5	3	11	12.2	10.3
	6	6	9.8	5.2	2	16	16.5	15.5
	5	7	-	6.2	1	31	31	31
Fith-order elliptic filter	4	**16**	22	**16**	2	22	23	23
	3	17	21	18	1	42	42	42

X. CONCLUSION

In this paper, we have proposed methods for multi-constraint static scheduling of SDFGs via retiming and unfolding. The methods presented include: an exact method that considers only optimal rate (maximal throughput), a general framework that schedules SDFGs under constraints on the number of processors, memory, auto-concurrency, or combinations of these, and heuristic methods that schedule SDFGs with optimal rate and with as few processors (or as little storage space) as possible. None of the methods explicitly converts an SDFG to its equivalent HSDFG.

We have carried out experiments on hundreds of synthetic SDFGs and several models of real applications. Our rate-optimal scheduling method is 4-7 orders of magnitude faster than the existing method [15] and it returns often optimal unfolding factors close to the smallest ones proven in [9]. Our processor minimal rate-optimal scheduling method further reduces the number of processors needed for a rate-optimal schedule by about 90%. In 54% of the tested models, our memory minimal rate-optimal scheduling method leads to a retimed SDFG whose rate-optimal schedule that requires less storage space than the proven minimal storage requirement of the original graph [16], and in 44% of the cases, the returned storage requirements equal the minimal ones. The average improvement is about 5.4%. The results also show that our methods are computationally efficient.

REFERENCES

[1] E. Lee and D. Messerschmitt, “Static scheduling of synchronous data flow programs for digital signal processing,” *IEEE Trans. on 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,” in *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. Acoustics, Speech, and Signal Processing Conf.*, 1995, pp. 2651–2654.

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

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

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

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

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

[9] 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.

[10] X.-Y. Zhu *et al.*, “Efficient retiming of multirate DSP algorithms,” *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 6, pp. 831–844, 2012.

[11] X.-Y. Zhu *et al.*, “Static Rate-Optimal Scheduling of Multirate DSP Algorithms via Retiming and Unfolding,” in *Proc. 18th Real-Time and Embedded Technology and Applications Symposium*, 2012, pp. 109–118.

[12] X.-Y. Zhu *et al.*, “Memory-constrained static rate-optimal scheduling of synchronous dataflow graphs via retiming,” in *Proc. 17th Design, Automation and Test in Europe (DATE)*, 2014, pp. 1–6.

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

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

[15] 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.

[16] S. Stuijk, M. Geilen, and T. Basten, “Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs,” *IEEE Trans. on Computers*, vol. 57, no. 10, pp. 1331–1345, 2008.

[17] S. H. De Groot, S. H. Gerez, and O. E. Herrmann, “Range-chart-guided iterative data-flow graph scheduling,” *IEEE Trans. on Circuits and Systems I: Fundamental Theory and Applications*, vol. 39, no. 5, pp. 351–364, 1992.

[18] S. Stuijk *et al.*, “Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs,” in *44th Design Automation Conference.*, 2007, pp. 777–782.

[19] A. K. Singh *et al.*, “Mapping on multi/many-core systems: Survey of current and emerging trends,” in *Proc. 50th Ann. Design Automation Conf. (DAC)*, 2013, p. 1.

[20] Y.-K. Kwok and I. Ahmad, “Static scheduling algorithms for allocating directed task graphs to multiprocessors,” *ACM Computing Surveys (CSUR)*, vol. 31, no. 4, pp. 406–471, 1999.

[21] Z. Zhou *et al.*, “Scheduling of parallelized synchronous dataflow actors for multicore signal processing,” *J. Signal Process. Syst.*, pp. 1–20, 2014.

[22] M. Kudlur and S. Mahlke, “Orchestrating the execution of stream programs on multicore platforms,” in *ACM SIGPLAN Notices*, vol. 43, no. 6, 2008, pp. 114–124.

[23] A. Malik and D. Gregg, “Orchestrating stream graphs using model checking,” *ACM Trans. Archit. Code Optim.*, vol. 10, no. 3, pp. 19:1–19:25, 2013.

[24] H. Yang and S. Ha, “Pipelined data parallel task mapping/scheduling technique for mpsoc,” in *Proc. Conference on Design, Automation and Test in Europe*, 2009, pp. 69–74.

[25] L. Chao and E. Sha, “Scheduling Data-Flow Graphs via Retiming and Unfolding,” *IEEE Trans. on Parallel and Distributed Systems*, vol. 8, no. 12, pp. 1259–1267, 1997.

[26] R. Govindarajan, G. R. Gao, and P. Desai, “Minimizing buffer requirements under rate-optimal schedule in regular dataflow networks,” *The Journal of VLSI Signal Processing*, vol. 31, no. 3, pp. 207–229, 2002.

[27] O. Moreira *et al.*, “Buffer sizing for rate-optimal single-rate data-flow scheduling revisited,” *IEEE Trans. on Computers*, vol. 59, no. 2, pp. 188–201, 2010.

[28] W. Che and K. S. Chatha, “Unrolling and retiming of stream applications onto embedded multicore processors,” in *Proc. 49th Annual Design Automation Conference (DAC)*, 2012, pp. 1272–1277.

[29] A. Ghamarian *et al.*, “Throughput analysis of synchronous data flow graphs,” in *Proc. International Conference on Application of Concurrency to System Design (ACSD).*, 2006, pp. 25–36.

[30] Y. Yang *et al.*, “Automated bottleneck-driven design-space exploration of media processing systems,” in *Proc. the Conference on Design, Automation and Test in Europe (DATE)*, 2010, pp. 1041–1046.

[31] A. K. Singh, A. Kumar, and T. Srikanthan, “A hybrid strategy for mapping multiple throughput-constrained applications on mpsocs,” in *Proc. 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, 2011, pp. 175–184.

[32] F. Kröger and S. Merz, *Temporal Logic and State Systems*. Springer-Verlag Berlin, 2008.

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

[34] M. Adé, R. Lauwereins, and J. Peperstraete, “Data memory minimisation for synchronous data flow graphs emulated on DSP-FPGA targets,” in *Proc. 34th Design Automation Conference (DAC)*, 1997, pp. 64–69.

[35] 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.

[36] M. H. Wiggers, M. J. Bekooij, and G. J. Smit, “Efficient computation of buffer capacities for cyclo-static dataflow graphs,” in *Proc. 44th Design Automation Conference (DAC).*, 2007, pp. 658–663.

[37] A. Moonen *et al.*, “Practical and accurate throughput analysis with the cyclo static dataflow model,” in *Proc. 15th Int. Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, 2007, pp. 238–245.

Xue-Yang Zhu (M’12) received the M.Sc. degree in mathematics from Fuzhou University, Fuzhou, China, in 1999, and the Ph.D. degree in computer software and theory from the Institute of Software, Chinese Academy of Sciences, Beijing, China, in 2005.

She is currently an Assistant Professor with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences. Her current research interests include the design of embedded systems, software architecture, and formal methods.

Marc Geilen (M’11) received the M.Sc. degree in information technology and the Ph.D. degree from the Eindhoven University of Technology, Eindhoven, The Netherlands.

He is currently an Assistant Professor with the Department of Electrical Engineering, Eindhoven University of Technology. His current research interests include modeling, analysis and synthesis of streamprocessing systems, multiprocessor systems-on-chip and wireless sensor networks, and multiobjective optimization and tradeoff analysis.

Twan Basten (M’98-SM’06) received the M.Sc. and Ph.D. degrees in computing science from the Eindhoven University of Technology (TU/e), Eindhoven, The Netherlands.

He is currently a Professor with the Department of Electrical Engineering, TU/e, where he chairs the Electronic Systems group. He is also a Senior Research Fellow with TNO Embedded Systems Innovation, Eindhoven. His current research interests include the design of embedded and cyber-physical systems, reliable computing, and computational models.

Sander Stuijk received the M.Sc. (with honors) and Ph.D. degrees in electrical engineering from the Eindhoven University of Technology, Eindhoven, The Netherlands, in 2002 and 2007, respectively.

He is currently an assistant professor in the Department of Electrical Engineering at Eindhoven University of Technology. He is also a visiting researcher at Philips Research Eindhoven working on bio-signal processing algorithms and their embedded implementations. His research focuses on modeling methods and mapping techniques for the design and synthesis of predictable systems.