Memory-Constrained Static Rate-Optimal Scheduling of Synchronous Dataflow Graphs via Retiming

Xue-Yang Zhu
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
China, Beijing 100190
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*—Synchronous dataflow graphs (SDFGs) are widely used to model digital signal processing (DSP) and streaming media applications. In this paper, we use retiming to optimize SDFGs to achieve a high throughput with low storage requirement. Using a memory constraint as an additional enabling condition, we define a memory constrained self-timed execution of an SDFG. Exploring the state-space generated by the execution, we can check whether a retiming exists that leads to a rate-optimal schedule under the memory constraint. Combining this with a binary search strategy, we present a heuristic method to find a proper retiming and a static scheduling which schedules the retimed SDFG with optimal rate (i.e., maximal throughput) and with as little storage space as possible. Our experiments are carried out on hundreds of synthetic SDFGs and several models of real applications. Differential synthetic graph results and real application results show that, in 79% of the tested models, our method leads to a retimed SDFG whose rate-optimal schedule requires less storage space than the proven minimal storage requirement of the original graph, and in 20% of the cases, the returned storage requirements equal the minimal ones. The average improvement is about 7.3%. The results also show that our method is computationally efficient.**

I. Introduction and Related Work

Dataflow models are widely used to represent DSP and streaming media applications, which are usually required to operate under real-time and resources constraints. *Synchronous dataflow graphs* (SDFGs) [1] are often used to model multirate DSP algorithms. Each node (also called actor) in an SDFG represents a computation and each edge models a FIFO channel; the sample rates of actors may differ. An example SDFG, G_1, is shown in Fig. 1 (a). In this paper, we are concerned with constructing efficient static schedules of SDFGs, where the schedules need to satisfy memory constraints.

A *static schedule* arranges the 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*. The average computation time per iteration is called the *iteration period*. SDFGs with recursion (or feedback) have an inherent lower bound on the iteration period, referred to as the *iteration bound* (i.e., the reciprocal of the maximum throughput). It is impossible to achieve an iteration period lower than the iteration bound, even when unlimited resources are available. A schedule whose iteration period equals the iteration bound is called a *rate-optimal* schedule.

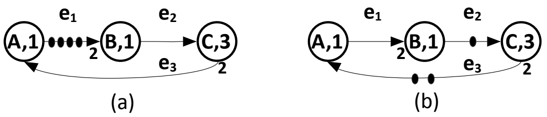


Fig. 1. (a) The SDFG G_1; (b) $R(G_1)$, a retimed graph of G_1 by retiming the actor B twice and C once. The sample rates are omitted when they are 1 and the computation time of each actor is attached inside the node. Black dots represent initial tokens on the edges.

Many studies have been conducted to investigate storage aspects for SDFGs [2], [3], [4], [5]. Most of them focus on finding a minimal storage requirement for a deadlock-free execution or for an execution under throughput constraints. None of them are concerned with how an SDFG can be optimized to get a smaller storage requirement.

Retiming [6] is a graph transformation technique that redistributes the graph's initial tokens, while its functionality remains unchanged. Retiming may reduce the iteration period for static periodic schedules of an SDFG. We show in this paper that it can also reduce the memory used by a graph. For example, according to [5], a rate-optimal schedule of G_1 needs at least storage space 10 with distribution $[4, 2, 4]$ corresponding to the edges e_1, e_2 and e_3, while for the retimed graph $R(G_1)$, shown in Fig. 1 (b), the minimal storage needed is 8 with distribution $[2, 2, 4]$.

Then, how to find such a retiming and how to schedule the retimed SDFG to achieve both optimal rate and low storage requirement? We answer these questions in this paper.

We compute such retimings and schedules by analyzing the behaviors of SDFGs. [7] proves that the iteration bound of an SDFG can be computed by exploring the state space generated by a self-timed execution (STE). STE analysis is used in [4] to construct static periodic rate-optimal schedules with minimal buffer sizes for *homogenous synchronous dataflow graphs* (HSDFGs), which is a special type of SDFGs. It is used in [5] to compute the minimal storage space under a given throughput constraint, and it is used in [8] to get a rate-optimal schedule

This work is partially supported by the National Basic Research Program of China (2014CB340701); by the Hi-tech Research and Development Program of China (2012AA011206); and by the National Natural Science Foundation of China (61061130541).

978-3-9815370-2-4/DATE14/©2014 EDAA

of an SDFG with and without processor constraints.

The state space includes the information of a retiming and a rate-optimal schedule of the retimed graph [8]. We show in this paper that the state space also includes the information about memory required by a schedule. Using a memory constraint as an additional enabling condition, we define a memory constrained STE (MC-STE). Exploring the state-space generated by the MC-STE, we can check whether a retiming exists that leads to a rate-optimal schedule under the memory constraint. Combining this with a binary search strategy, we present a heuristic method to find a proper retiming and a static scheduling which schedules the retimed SDFG with optimal rate and with as little storage space as possible.

To evaluate our new method, we implemented it in the tool SDF3 [9]. We compare the storage requirements of the retimed graphs with the minimal storage requirements of the original graphs computed with [5]. We also show the execution time of our method. Our experiments were carried out on hundreds of synthetic SDFGs and several models of real applications. The experimental results show that our method is computationally efficient and the retimed SDFGs improve or equal the minimal required storage space of the original SDFGs in about 79% and 20% of the tested models, resp.

The remainder of this paper is organized as follows. We describe the definitions and an operational semantics of SDFGs in Sections II and III, resp. Our main results are illustrated in Sections IV and V. Section VI provides an experimental evaluation. Section VII concludes.

II. Preliminaries

A *synchronous dataflow graph* (SDFG) is a finite directed graph $G = \langle V, E\rangle$. 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: $d(e)$, the number of initial tokens on e; $prd(e)$, a positive integer that represents the number of tokens produced onto e by each firing of the source 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. Note that for technical reasons explained later, we allow $d(e)$ to be negative. The source actor and sink actor of e are denoted as $src(e)$ and $snk(e)$, resp. The set of incoming edges to actor v is denoted by $InE(v)$, and the set of outgoing edges from v by $OutE(v)$. If $prd(e) = cns(e) = 1$ for each $e \in E$, G is a *homogeneous SDFG* (HSDFG).

An SDFG G is *sample rate consistent* [1] if and only if there exists a positive integer vector $q(V)$ satisfying the *balance equations*, $q(src(e)) \times prd(e) = q(snk(e)) \times cns(e)$ for all $e \in E$. 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 equations, we have G_1's repetition vector $q = [2, 1, 1]$. Only sample rate consistent and deadlock-free SDFGs are meaningful in practice. We consider only such SDFGs, which can be verified efficiently [1].

An *iteration* is a firing sequence in which each actor v occurs exactly $q(v)$ times. A *static schedule* is a function S, mapping a firing to its start time. The i^{th} firing of actor v starts at time $S(v, i)$, $i \in [1, \infty)$. If $S(v, i + f \cdot q(v)) = S(v, i) + T$ for every firing of v, we say that the schedule S arranges each f iterations as a *cycle* with a *cycle period* T. Such a schedule can be represented by the first f iterations and period T. It 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 schedule equals IB, it is a *rate-optimal schedule*.

The IB of an HSDFG is given by its maximum cycle mean [10]. 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 the maximum cycle mean of its equivalent HSDFG. [7] presents a method to compute the IB directly on an SDFG, which we introduce later.

Retiming [6] is a graph transformation technique that redistributes the graph's initial tokens while the functionality of the graph remains unchanged. Retiming an actor once means firing this actor once. The SDFG $R(G_1)$ shown in Fig. 1 (b), for example, is a retimed graph of G_1 by retiming R, which is defined as $R(A) = 0$, $R(B) = 2$ and $R(C) = 1$. A retiming r of G is *legal* if the number of initial tokens of each edge of $r(G)$ is nonnegative [11]. Only legal retimings are meaningful. Note that retiming does not affect the IB of an SDFG.

By inserting precedence constraints with a finite number of initial tokens between the source and sink actors of an SDFG, any SDFG can be converted to a strongly connected graph [12]. We therefore only consider strongly connected SDFGs.

III. An Operational Semantics of SDFGs

For developing our method, we define the behavior of an SDFG G in terms of a *labeled transition system*, represented by $LTS(G)$, similar to [8]. We use $tn(e)$ to record the current number of tokens on edge e. SDFGs allow 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. Vector $tnb(E)$ is used for memory analysis. $tnb(e)$ is the buffer usage of e at each moment. Our purpose is to construct fast schedules, so we need a global clock, $glbClk$, to record the time progress.

A *state* of $LTS(G)$ is a 3-tuple that consists of the values of $tn(E)$, $tr(V)$ and $tnb(E)$. The *initial state* of $LTS(G)$ is denoted as s_0. At s_0, $tn(E)$ and $tnb(E)$ are the initial delay distribution $d(E)$; each element of $tr(V)$ is an empty queue. The behavior of an SDFG consists of a sequence of *firings*. We use actions $sFiring(v)$ and $eFiring(v)$ to model the start and end of a firing of v, and use $readyS(v)$ and $readyE(v)$ as predicates capturing their enabling conditions, resp. In parallel with actor firing, time elapses, represented by the increase of $glbClk$. A time step is modeled by the action clk.

In line with [5], we choose a relatively conservative storage abstraction to leave more room for implementation. That is, when an actor starts firing, it claims the space of the tokens it will produce, and it releases the space of the tokens it consumes only when the firing ends.

The guard *readyS*(v) tests if there are sufficient tokens on the incoming edges of actor v to enable a firing.

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

When a firing of v starts, it reduces the number of tokens of its incoming edges according to the consumption rates and inserts its computation time, $t(v)$, into queue $tr(v)$. At the same time, the buffer size needed by the end of the firing is claimed.

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

where $tn'(e)$, $tnb'(e)$ and $tr'(v)$ refer to the value of $tn(e)$, $tnb(e)$ and $tr(v)$ in the new state, resp.; $ENQ(tr(v), t(v))$ inserts $t(v)$ at the end of $tr(v)$. For conciseness, we omit the elements of states if their values remain unchanged. The space for the consumed tokens on the incoming edges of v is not yet released at this moment; therefore the *tnb* of those edges remains unchanged. The effect of *sFiring* can be illustrated by the change from state S_1 to state S_2 in Fig. 2.

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 $tr(v)$.

When a firing of v ends, it increases tokens of its outgoing edges according to the production rates and removes the first element from queue $tr(v)$. The space for the consumed tokens on the incoming edges of v is released.

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

where $DLQ(tr(v))$ removes the first element of $tr(v)$. Note that the space for the increased $prd(e)$ tokens on the outgoing edges of v has been claimed when the firing starts, so the end of the firing does not increase the required space of it outgoing edges.

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.

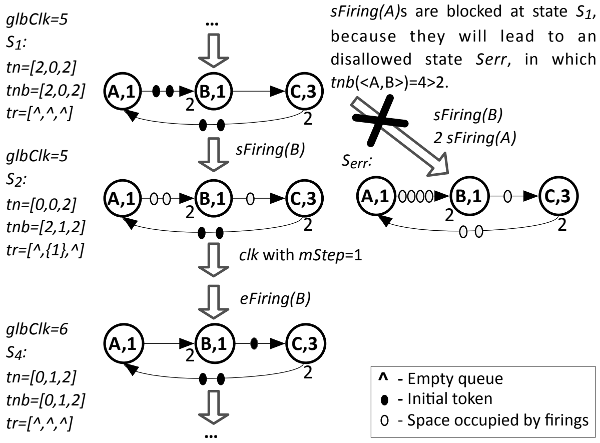
$$mStep = \min_{c \in \bigcup_{v \in V} tr(v)} c.$$

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

$$\begin{aligned} clk \equiv_{def} & (\forall v \in V : |tr(v)| > 0 \Rightarrow (\forall x \in tr(v) : x' = x - mStep)) \\ & \wedge\, glbClk' = glbClk + mStep, \end{aligned}$$

where $|tr(v)|$ means the length of $tr(v)$. Its enabling condition guarantees that a *clk* action leads to a nonnegative value in $tr(v)$. In a time step, time may not progress if any 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 *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.



*sFiring(A)*s are blocked at state S_1, because they will lead to an disallowed state *Serr*, in which *tnb*(<A,B>)=4>2.

Fig. 2. A part of the MC-STE of G_1 with $MC = [2, 2, 4]$).

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

The operational semantics extends the semantics in [8] by adding $tnb(E)$ to record the storage requirement.

IV. Memory Constrained Self-timed Execution

A *self-timed execution* (STE) is an execution in which time steps only occur when no actors are ready to start [10]. An STE ultimately goes into a repetitive pattern (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 [7].

Without taking into account any resource constraints, an STE runs as soon as possible. A rate-optimal schedule can be delivered from it [8]. A storage requirement can also be computed according to the semantics we define in Section III. If an STE cannot get sufficient storage space when it is ready to go, some of its enabled firings have to be blocked. We call such an STE a *memory constrained STE* (MC-STE).

In an MC-STE, besides sufficient tokens on its incoming edges, enough space on its outgoing edges is needed for an actor to fire. See some states of an MC-STE shown in Fig. 2, for example. At state S_1, tokens on edges are available for actor A to fire twice and B to fire once. When the storage space of the edge $\langle A, B \rangle$ is limited to 2, there is insufficient space left for the tokens that will be produced by the firings of A, then the firings of A are blocked.

Suppose a memory constraint is modeled by a vector $MC(E)$. The enabling condition for a firing of an actor now also needs to check the remaining space of its outgoing edges. We denote the new enabling condition as $readyS_{mc}$.

$$\begin{aligned} readyS_{mc}(v) \equiv_{def} & readyS(v) \\ & \wedge (\forall e \in OutE(v) : prd(e) \leq MC(e) - tnb(e)). \end{aligned}$$

An SDFG with memory-constrained buffers is an instance of the Resource-Aware SDF model of [13], with the edge buffers as resources. These buffers can also be modeled by adding an incoming edge with tokens to model available storage space [5]. Therefore, an MC-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.

Theorem 1. An MC-STE of SDFG $G = \langle V, E\rangle$ with memory constraint $MC(E)$ 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 $e' = \langle v, u\rangle \in E'$: $d(e') = MC(e) - d(e), prd(e') = cns(e)$ and $cns(e') = prd(e)$, where $e = \langle u, v\rangle \in E$.

In G', there may exist $e' \in E'$ with $d(e') < 0$ when the buffer capacity $MC(e)$ is smaller than $d(e)$. Redistribution of those initial tokens may reduce the buffer requirement of e.

By Theorem 1, the properties of the STE [7] are still satisfied by the MC-STE. An MC-STE of an SDFG, σ, includes a periodic phase and a transient phase, denoted as σ_p and σ_t, resp. The beginning state and the end state of σ_p are denoted as s_b and s_e, resp. The periodic phase consists of a whole number of iterations, denoted as $nIter(\sigma_p)$. The average iteration computation time in σ_p is

$$IP(\sigma_p) = (s_e.glbClk - s_b.glbClk)/nIter(\sigma_p).$$

However, since memory constraints limit the throughput, $IP(\sigma_p)$ does not always equal the IB of the SDFG considered, but rather the IB of G' mentioned in Theorem 1.

We can find an MC-STE in finitely many steps: beginning with the initial state s_0 and ending at s_e. We directly call such a finite state sequence an MC-STE in the remainder of the paper. The procedure to obtain an MC-STE of G with a memory constraint MC, $conSTE(G, MC)$, is a variation of Algorithm 1 in [8], in which $readyS(v)$ is replaced with $readyS_{mc}(v)$.

According to the operational semantics, the number of tokens on each edge decreases only after an *sFiring* action. The enabling condition of *sFiring* guarantees that the number of tokens never goes negative in an execution. Hence, the transient phase of an MC-STE forms a legal retiming of G.

Theorem 2. Given the MC-STE σ of an SDFG G, the retiming r defined as, $r(v)$ = the number of $sFiring(v)$ actions in σ_t for each v, is legal.

V. Rate-Optimal Scheduling under Memory Constraints

If an 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 of an MC-STE, then the times that *sFiring* actions in σ_p happen shifted by $s_b.glbClk$, form a schedule of the new graph. It is obvious that a retiming obtained from σ_t transforms the SDFG to such a new graph. The vector *tnb* of a state records the memory required at each moment. It implies the memory required by the schedule.

Theorem 3. Given the MC-STE σ of an SDFG G and the retiming obtained from σ_t, r, a schedule S of $r(G)$ with its IP= $IP(\sigma_p)$ is defined as: for $1 \le i \le nIter(\sigma_p) \cdot q(v)$,

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

where $sFiring(v, i)$ starts the i^{th} firing of v in σ_p. The schedule requires storage space $\sum_{e\in E} TNB(e)$, where $TNB(e) = \max_{s\in\sigma_p} s.tnb(e)$.

A procedure for scheduling G under memory constraint MC, $conSch(G, MC)$, is a variation of Algorithm 2 in [8], in which $STE(G)$ is replaced with $conSTE(G, MC)$. If the IP returned by $conSch(G, MC)$ equals the IB of G, we say that MC is *feasible* for a rate-optimal schedule of G.

For an MC-STE σ, σ_t may need more storage space than σ_p. In a real implementation of an SDFG, if the retiming process is carried out at runtime, the firings corresponding to r can be arranged to run under the buffer size of S with a slower speed (less concurrent firings) instead of an as soon as possible execution like in the MC-STE.

According to our assumption of the storage model, in an MC-STE, the order of *sFiring*s at the same moment is irrelevant. A different order leads to the same state. At state S_1 in Fig. 2, for example, no matter whether $sFiring(A)$ or $sFiring(B)$ occurs first, after all the enabled firings start, the resulting state is always S_{err}. Hence, at S_1, $sFiring(A)$ has to be blocked. Therefore, if procedure $conSch(G, MC)$ does not return a rate-optimal schedule, no other MC-STE of G with MC can lead to a rate-optimal schedule.

We can then present an algorithm to seek a retiming which can lead to a rate-optimal schedule with the storage space as small as possible. We use a binary search per edge on the memory constraints. The $TNB(E)$ of an STE records the memory required by a rate-optimal schedule without any resource limitation. It is already feasible. We use it as an upper bound of the binary search. The lower bound can be set to $\langle 0, ..., 0\rangle$ or a storage distribution to avoid deadlock gotten by [2]. The former is too low and the latter itself takes time to compute. We use a compromise between them. Let $LB(E)$ be a vector, in which each $LB(e)$ is the larger one of $prd(e)$ and $cns(e)$. A deadlock-free execution uses at least $LB(e)$ buffer size for e. We use $LB(E)$ as a lower bound of the binary search.

```
Algorithm 1 conOptSch($G$)
Input: A strongly connected SDFG $G$
Output: A legal retiming $r$, a rate-optimal schedule $S$ of $r(G)$
    with the storage space $minB$
Get the $IB$ and $TNB$ from $STE(G)$
// By Algorithm 1 in [8]
Let $optB = TNB$
for all $e \in E$ do
    Perform a binary search over $[LB(e), optB(e)]$ ; assuming $x$ is the value considered, let vector $MC$ be defined as $MC(e) = x$ and $MC(e') = optB(e')$ if $e' \neq e$; use $conSch(G, MC)$ to test whether $MC$ is feasible for a rate-optimal schedule and let $optB = MC$ if so.
end for
get $r$, $S$ and $TNB$ from $\sigma = conSTE(G, optB)$
$minB = \sum_{e\in E} TNB(e)$
return $r$, $S$ and $minB$
```

The procedure is shown in Algorithm 1. It includes $|E|$ binary searches and begins with the memory constraint $optB = TNB$ of STE. Each time one edge e is considered, while 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 $conSch(G, optB)$ is rate-optimal and requires *minB* for storage.

Algorithm 1 is heuristic. In general, the 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 1 does return a storage requirement less or at most equal to the proven minimal feasible storage space returned by [5], which does not use retiming. We use a random edge order in our experiments.

VI. Experimental Evaluation

A. Experimental Setup

We implemented our algorithm *conOptSch* in SDF3 [9]. We compare the storage space for the retimed graphs returned by our method (minB) with the minimal storage requirements of the original graphs computed using the algorithm of [5] (SGB08) and show the execution time of them. We performed experiments on two sets of SDFGs, running on a 2.67GHz CPU with 12MB cache. The experimental results are shown in Tables I, II and III. 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) [14], a satellite receiver (Satellite) [15], a maximum entropy spectrum analyzer (MaxES), an Mp3 playback application (Mp3) [16] and a channel equalizer (CEer) [17]. Adopting the method in [12], 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 tested 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. A large delay count may slowdown the STE procedure and therefore our method. 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. 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 our method changes with them.

For both sets, we consider each SDFG in two cases: with and without auto-concurrency. In the former case, at the same time, there can be concurrent firings of the same actor. In the latter case, the number of concurrent firings of an actor is limited to one. To analyze an SDFG without auto-concurrency, we use a method different from [5], in which a self-loop with one initial token is added to each actor to model the limitation. We do this by not allowing a size of queue $tr(v)$ larger than one. Recall that $tr(v)$ contains the remaining times of the concurrent firings of actor v; when its size is limited to one, no concurrent firings of the same actor are allowed.

B. Experimental Results

Table I gives the information about and results for the practical DSP examples. There are three parts in Table I. The first part is the information on the graphs, including the number of actors (*nA*) and the sum of the elements in the repetition vector (*nQ*); the second part shows the iteration bound (*IB*) of each graph and the returned storage requirement and execution time of our method (minB) and [5] (SGB08) when auto-concurrency is allowed; the third part shows the same items for the cases without auto-concurrency. The information and the storage space do not include the dummy actors and edges.

TABLE I. 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
Storage Requirement					
minB	2916	1328	2087	73	15168
SGB08	N	N	N	73	N*
Execution Time (ms)					
minB	213	45	55	1	5807
SGB08	N	N	N	1	N
Without Auto-concurrency					
IB	120000	960	8192	47128	1056
Storage Requirement					
minB	2902	34	1322	73	1544
SGB08	N	34	N	73	1544
Execution Time (ms)					
minB	193	16	59	1	91
SGB08	N	18	N	3	897

* no results available because of timeout.

For the practical DSP models for which SGB08 can finish in ten hours and return the results, our method reaches the minimal storage requirement of the original graphs. The model that takes the longest execution time is the satellite model, which has a relatively large *nA* and *nQ*.

Tables II and III give the results for the synthetic examples. Besides the ranges of *nA* and *nQ* of the graphs, each table includes two parts for cases with and without auto-concurrency, resp. Each part shows the storage improvement of our method comparing with [5] and the execution times. Each point includes the average, maximal and minimal values (AVG/MAX/MIN) of graphs in the same group.

The storage improvements show no clear difference when *nQ* changes, so we show only the results divided by *nA*. For the same *nA*, the average of the improvements of the cases without auto-concurrency is larger. Of 79%, 20% and 1% of all

TABLE II. Experimental results for synthetic examples with $nD = 0$

	10-15	20-25	50-65	*nA* *nQ*
Storage Improvement (AVG/MAX/MIN) (%)				
imp.	2.5/45.7/-29.7	5.4/48.0/-1.5	8.9/59.7/-20.0	1k-6k*
Execution Time (AVG/MAX/MIN) (ms)				
minB	12/56/0	14/90/1	73/355/3	1k-1.5k
	22/68/1	35/345/2	101/847/4	2k-2.5k
	33/214/2	104/685/3	229/2,560/6	4k-6k
SGB08	11% of the tested graphs does not finish within 30 minutes.			
Without Auto-concurrency				
Storage Improvement (AVG/MAX/MIN) (%)				
imp.	8.7/65.8/0.2	7.7/72.5/0.3	11.9/60.8/0.8	1k-6k
Execution Time (AVG/MAX/MIN) (ms)				
minB	25/99/3	53/344/7	412/2,051/52	1k-1.5k
SGB08	68/1,042/5	497/11,226/19	346/781/132	
minB	66/177/4	131/688/16	942/5,400/64	2k-2.5k
SGB08	128/1,049/8	1,582/57,034/36	2,793/48,719/294	
minB	96/640/10	415/4,321/14	2,119/9,313/177	4k-6k
SGB08	192/1,761/23	N/N/41	2,970/28,577/548	

* 1k=1000.

TABLE III. Experimental results for synthetic examples with $nD = 0.9$

	10-15	20-25	50-65	*nA* *nQ*
Storage Improvement (AVG/MAX/MIN) (%)				
imp.	2.4/51.6/-0.1	4.1/38.6/0.0	6.2/45.2/-0.2	1k-6k
Execution Time (AVG/MAX/MIN) (ms)				
minB	161/972/1	249/2,220/2	321/1,869/2	1k-1.5k
	467/9,313/1	371/2,043/1	409/6,076/8	2k-2.5k
	742/4,393/16	223/1,674/3	745/5,325/10	4k-6k
SGB08	24% of the tested graphs does not finish within 30 minutes.			
Without Auto-concurrency				
Storage Improvement (AVG/MAX/MIN) (%)				
imp.	8.7/90.0/-11.2	7.0/84.5/0.3	9.0/46.9/0.3	1k-6k
Execution Time (AVG/MAX/MIN) (ms)				
minB	102/296/6	340/1,996/17	2,513/31,154/32	1k-1.5k
SGB08	84/349/8	245/1,163/23	1,218/6,562/152	
minB	176/1,116/6	804/9,281/13	2,699/51,197/126	2k-2.5k
SGB08	333/3,697/13	435/2,203/44	1,980/13,063/369	
minB	483/2,346/80	529/2,128/26	4,788/28,829/177	4k-6k
SGB08	9,167/267,587/22	N/N/56	12,013/278,440/630	

the tested models that finished within 30 minitues, our method returns storage spaces smaller than, equal to and larger than that returned by [5], resp. The average improvement is 7.3%.

The maximal execution time on all models is about 51 seconds. In the same group, for both methods, the execution times of different models may differ largely, but the differences among groups are clear, the execution times generally increase with the *nA*, *nQ* and *nD*. For most graphs, for the same model, our method runs the cases allowing auto-concurrency faster than the cases without auto-concurrency. On the contrary, [5] runs faster on the latter cases.

VII. Conclusion

In this paper, we have presented an efficient heuristic method to use retiming to optimize SDFGs. The retimed graph is statically scheduled with optimal rate and with a storage requirement which is often less than the proven minimal storage requirement of the original graph. Our experimental results show that our method does not reach the minimum only in 1% of the hundreds of tested models, while in 79% of the cases it returns a lower required storage space than the minimum of the original graph.

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] M. Adé, R. Lauwereins, and J. Peperstraete, “Data memory minimisation for synchronous data flow graphs emulated on dsp-fpga targets,” in *Proc. of the 34th annual DAC*, 1997, pp. 64–69.

[3] 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.

[4] O. Moreira, T. Basten, M. Geilen, and S. Stuijk, “Buffer sizing for rate-optimal single-rate data-flow scheduling revisited,” *IEEE Trans. Comput*, vol. 59, no. 2, pp. 188–201, 2010.

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

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

[7] 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 *Sixth International Conference on Application of Concurrency to System Design, ACSD 2006*. IEEE, 2006, pp. 25–36.

[8] X.-Y. Zhu, M. Geilen, T. Basten, and S. Stuijk, “Static rate-optimal scheduling of multirate dsp algorithms via retiming and unfolding,” in *Proc. of the 18th Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012*. IEEE, 2012, pp. 109–118.

[9] 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.

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

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

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

[13] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, “Automated bottleneck-driven design-space exploration of media processing systems,” in *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*, 2010, pp. 1041–1046.

[14] 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.

[15] 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.

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

[17] 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*, 2007, pp. 238–245.