Efficient Algorithm for the Iteration Period Computation of Unfolded Synchronous Dataflow Graphs

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

***Abstract*—Synchronous dataflow graphs (SDFGs) are widely used to model streaming applications. Unfolding is one of the most important techniques for performance optimization of SDFGs. It may reduce the iteration period (IP) without affecting functionality. We present a novel method to compute the IPs of SDFGs by state-space exploration, without converting them to their equivalent homogeneous SDFGs (HSDFGs), and without further unfolding the HSDFGs. The conversion procedures are time and space-consuming. We also consider the cases when there are resource constraints, which cannot be dealt with by existing methods. Combining with retiming technique, we further present a method to compute the reduced IP of unfolded SDFGs. Our experimental results show that the proposed method outperforms the existing methods significantly.**

I. Introduction and Related Work

Dataflow models of computation are widely used to represent digital signal processing (DSP) applications. Each node (also called actor) in such a model represents a computation or function and each edge models a FIFO channel. One of the most useful dataflow models for designing multi-rate DSP algorithms is *synchronous dataflow graphs* (SDFGs) [1]. The sample rates of actors of an SDFG may differ. The graph G_1 in Fig. 1(a), for example, is an SDFG. The real multi-rate DSP algorithms modeled with SDFGs include the simplified spectrum analyzer [2] and the satellite receiver [3]. *Homogenous synchronous dataflow graphs* (HSDFGs) are a special type of SDFGs. All sample rates of actors of an HSDFG are one.

DSP algorithms are often repetitive. Execution of all the computations for the required number of times is referred to as an *iteration*. A DSP algorithm repeats iterations periodically. The number of firings of actors in an SDFG is decided by its repetition vector, which we explain later. An iteration of G_1 in Fig. 1(a), for example, includes two executions, often called *firings*, of actor A, two firings of B and one of C. The *iteration period* (IP) is the minimal achievable average execution time per iteration of an algorithm [4]. It's an important property of an SDFG, deciding how fast the application that is modeled by the SDFG can be. Reducing the IP of an SDFG may improve the performance of the corresponding application.

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

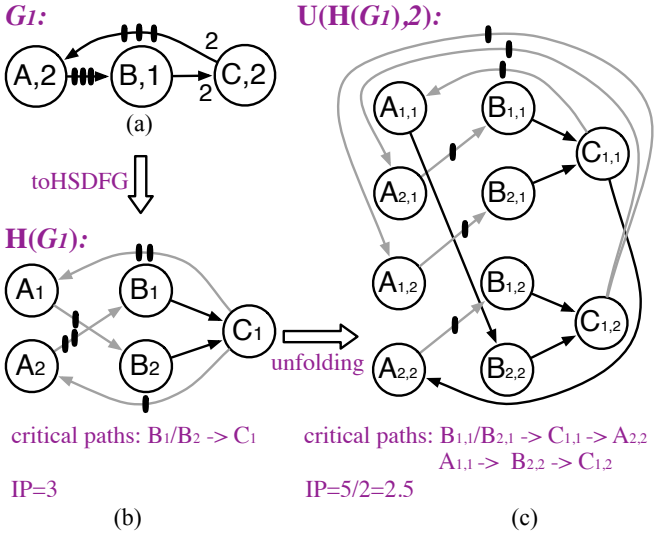

Fig. 1. (a) The SDFG G_1. The computation time of each actor is attached inside the node. An iteration of G_1 includes two firings of actor A, two firings of B and one of C. (b) The equivalent HSDFG of G_1, in which A_i means the i^{th} firing of actor A. (c) 2-unfolded graph of $H(G_1)$, in which $A_{i,j}$ means the i^{th} firing of actor A in the j^{th} iteration. Black dots on edges represent initial tokens on the channels; for clarity, the sample rates are omitted when they are 1 and the channels with initial tokens are denoted by gray lines in the HSDFGs.

Schedules of SDFGs on multiprocessor can be nonoverlapped or overlapped [5] . In a nonoverlapped schedule, firings in different iterations may not overlap, that is, any firing of an iteration starts only after completion of all firings of the previous iteration, while in an overlapped schedule, firings in different iterations may overlap. In this paper, we consider IPs in the context of nonoverlapped scheduling, the same as that in [6], [7], [8], [9].

A valid SDFG can always be converted to an equivalent HSDFG [10], which captures the data dependences among firings of actors in the original SDFG in an iteration. The graph in Fig. 1(b), for example, is the equivalent HSDFG of G_1. The IP of an SDFG equals the IP of its equivalent HSDFG, which is the length of its critical path [6]. For example, the IP of G_1 is 3, the length of path $B_1 \rightarrow C_1$.

Unfolding [5] can reduce the IP by exploiting the inter-iteration data dependencies. An unfolded SDFG with an

unfolding factor f, called f-unfolded SDFG, describes f consecutive iterations of the original graph. The HSDFG in Fig. 1(c), for example, is a 2-unfolded graph of the HSDFG in Fig. 1(b), in which, $B_{2,2}$, the second firing of actor B in the second iteration, depends on $A_{1,1}$, the first firing of actor A in the first iteration. The length of its critical path is 5. Because it includes two iterations, the IP of the 2-unfolded graph is $\frac{5}{2}$. That is, after unfolding, the IP of G_1 is reduced to $\frac{5}{2}$.

Traditionally, the computation of the IP of an unfolded SDFG is done by first converting the SDFG to its equivalent HSDFG (G_1 to $H(G_1)$ in Fig. 1), then unfolding the HSDFG with the given unfolding factor ($H(G_1)$ to $U(H(G_1), 2)$ in Fig. 1), and at last computing the length of the critical path of the unfolded HSDFG. This procedure, however, is very time-consuming when SDFGs scale up. Converting an SDFG to an HSDFG may exponentially increase the size of the graph under consideration in extreme cases [9]. Unfolding further multiplies the problem space.

Although there are efficient algorithms to compute IPs of SDFGs without converting them to HSDFGs [7], [8], these methods cannot be used on unfolded SDFGs directly. In this paper, we present an efficient algorithm to compute the IP of an unfolded SDFG. Our contributions are as follows.

1) We define the IP graph of the f-unfolded SDFG, an execution of which is equivalent to that of the f-unfolded SDFG. The IP of an f-unfolded SDFG is then computed by exploring the state space of its IP graph.
2) The proposed method can be easily extended to deal with the case when the resource constraints are considered. We present buffer bound and processor limitation as examples.
3) Combining with retiming technique [6], our method can further be applied to compute the reduced IP of unfolded SDFGs, with or without resource constraints.
4) The IP computation methods for different cases are thoroughly evaluated by experiments on four practical streaming applications and hundreds of synthetic graphs.

The proposed method uses state-space exploration (SSE) to solve problems related to unfolding, which is also used in [11], [12]. The main difference between the proposed method and that in [11], [12] is on the addressed problems. The problem that [11], [12] focuses on is: for a given SDFG, how to retime and unfold an SDFG to get a schedule that is rate-optimal? That is, finding the optimal retiming and unfolding factor of the SDFG. It is in the context of overlapped scheduling, like in [13]. The unfolding factor is an output in the methods in [11], [12]. The problem that the proposed method focuses on is: for a given SDFG and a given unfolding factor, what would be the iteration period of the unfolded SDFG (without really unfolding the SDFG)? It is in the context of non-overlapped scheduling, like in [14]. The unfolding factor is an input. To the best of our knowledge, the proposed method is the first work that uses SSE on this problem. And it does outperform the existing method much, as our experimental results show.

The remainder of this paper is organized as follows. We first introduce the related concepts and formulate the problems in Section II. An operational semantics of SDFGs is defined in Section III. Our main contributions are presented in Sections IV, V and VI. Section VII provides an experimental evaluation. Finally, Section VIII concludes.

II. Preliminaries and Problem Formulation

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. Each actor $v \in V$ is weighted with its computation time $t(v)$, a positive integer. E is the set of directed edges, modeling interconnections between functional elements. Each edge $e \in E$ is weighted with three properties: $d(e)$, 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 execution of the source actor of e; $cns(e)$, a positive integer that represents the number of tokens consumed from e by each execution of the sink actor of e. These numbers are also called the *delay*, *production rate* and *consumption rate*, respectively. If $prd(e) = cns(e) = 1$ for each $e \in E$, G is a *homogenous SDFG* (HSDFG). For technique reasons, we may allow $prd(e) = 0$ to mean that no tokens are produced on e.

The source actor and sink actor of $e \in E$ are denoted as $src(e)$ and $snk(e)$, respectively. The edge e with source actor u and sink actor v is denoted by 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 an SDFG is a vector containing delays on all edges of the SDFG G, denoted as $d(G)$. Take the SDFG G_1 in Fig. 1(a) for example. For each edge e, its $prd(e)$ and $cns(e)$ that are not equal to one and its $d(e)$ that is not equal to zero are labeled on e. The computation time vector $t = [2, 1, 2]$, corresponding to actors A, B and C. The initial delay distribution $d(G_1) = [3, 0, 3]$, corresponding to edges $\langle A, B\rangle, \langle B, C\rangle, \langle C, A\rangle$.

An SDFG G is *sample rate consistent* if and only if there exists a positive integer vector $q(V)$ satisfying *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* [1]. We use q to represent the repetition vector directly. One *iteration* of an SDFG G is an execution sequence in which each actor v in G occurs exactly $q(v)$ times. The *iteration period* (IP) of G is the minimal achievable average execution time per iteration, represented by $IP(G)$.

For example, a balance equation can be constructed for each edge of G_1 in Fig. 1(a). By solving these balance equations, we have G_1's repetition vector $q = [2, 2, 1]$. Since, there are no tokens on edge $\langle B, C\rangle$, actor C can start to fire only after B fires twice and produces two tokens on $\langle B, C\rangle$. Two firings of B can start concurrently. Therefore the time required for executing one iteration of G_1 is at least 3, that is, $IP(G_1) = 3$.

An SDFG is *sample rate inconsistent* if there is no nonzero solution for its balance equations. Any execution of an inconsistent SDFG will result in deadlock or unbounded memory. We only consider sample rate consistent and deadlock-free SDFGs, referred to as *valid SDFGs*. A valid SDFG can always

be converted an equivalent HSDFG [10]. The IP of an SDFG can be obtained by computing the length of the critical path of its equivalent HSDFG.

Unfolding is originally defined on HSDFGs [5]. An unfolded graph with an unfolding factor f, called *f-unfolded graph*, describes f consecutive iterations of the original graph. A procedure to construct unfolded HSDFGs is presented in [14]. Let H denote the transformation from an SDFG to its equivalent HSDFG, and U denote the unfolding transformation. The behavior of an f-unfolded graph of SDFG G is equivalent to the behavior of HSDFG $U(H(G), f)$.

Definition 1. *The IP of f-unfolded graph of SDFG G, $IP(G, f)$, is the IP of f-unfolded graph of its equivalent HSDFG, $IP(U(H(G), f))$.*

The problems we address is: given an SDFG G and an unfolding factor f, how to compute $IP(G, f)$ efficiently.

Let $IP_{buf}(G, f)$ be the IP under buffer bound $buf(E)$, $IP_{pro}(G, f)$ be the IP when there are pro number of processors available, and $IP_{BandP}(G, f)$ be the IP under the combination of above two constraints. Prefix re is added to the corresponding IPs of the optimally retimed SDFGs. We will also show that how the solution for the problem can be easily extended to compute $reIP$, $(re)IP_{buf}$, $(re)IP_{pro}$ and $(re)IP_{BandP}$ efficiently.

III. An Operational Semantics of SDFGs

For developing our method, we define the operational semantics of SDFGs similar to [11], [12] and restate here for the completeness. The behavior of an SDFG G in terms of a simple *transition system*, represented by $TS(G)$. A transition system includes a set of states, a set of actions, an initial state and a set of transitions which define rules on how to change states depending on different actions. Before defining the transition system of an SDFG, we introduce some notations to simplify the later illustrations.

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

A *state* of $TS(G)$ is a 3-tuple that consists of the values of $tn(E)$, $tr(V)$ and $glbClk$. In the *initial state* of $TS(G)$, s_0, $tn(E)$ is the initial delay distribution $d(G)$; no firings have been started, so each element of $tr(V)$ is empty; and the global clock $glbClk$ is zero. For example, the initial state of G_1 in Fig. 1(a) is $s_0 = ([3, 0, 3], [\{\}, \{\}, \{\}], 0)$, corresponding to edges $\langle A, B\rangle$, $\langle B, C\rangle$ and $\langle C, A\rangle$ and actors A, B and C, where $\{\}$ represents an empty queue.

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

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

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

When $readyS(v)$ is satisfied, actor v can start to fire. Actor v starting a firing, $sFiring(v)$, is to insert its computation time, $t(v)$, into queue $tr(v)$, and to consume tokens of all of its incoming edges according to the consumption rates. That is,

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

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

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

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

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

An actor v ending a firing, $eFiring(v)$, is to remove the first element from queue $tr(v)$ and to produce tokens on all of its outgoing edges according to the production rates. That is,

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

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

Time progresses as much as possible when no actor is ready to end or start. A time step, clk, reduces the remaining times of all firing actors by the minimal element of $tr(v)$ of all v, and increases the global clock by one. The largest possible time step $mS = \min_{c \in \bigcup_{v \in V} tr(v)} c$.

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

where $isEmpty(tr(v))$ tests if $tr(v)$ is empty. Notice that the delay distribution remains unchanged by a time step.

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

An *execution* of SDFG G is a sequence of states of $TS(G)$ beginning with the initial state and following by states caused by transitions from their predecessors. According the definition of time step, each firing in an execution starts as soon as possible.

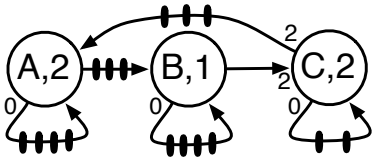

Fig. 2. IP graph of 2-unfolded graph of G_1.

IV. IP Computation

In this section, we first define the IP graph for an unfolded SDFG, then present the algorithm for computing the IP by exploring the state-space of the execution of the IP graph in Section IV-B.

A. The IP graph of the unfolded SDFG

According to the definition of unfolding, a f-unfolded graph describe f consecutive iterations of the original graph. The earliest possible completion time of the execution of the f iterations divided by f is the IP of the f-unfolded graph. Based on this observation, we construct IP graphs of f-unfolded SDFGs for IP computation.

Definition 2. *Given an SDFG $G = \langle V, E\rangle$ and an unfolding factor f, the* IP graph *of the f-unfolded graph of G is an SDFG $ipG_f = \langle V, E \cup E_f\rangle$, in which $E_f = \{\langle v, v\rangle | v \in V\}$ and for all $e' = \langle v, v\rangle \in E_f$, $prd(e') = 0$, $cns(e') = 1$ and $d(e') = f \cdot q(v)$.*

Fig. 2, for example, shows an IP graph of 2-unfolded graph of G_1. Executing an SDFG one iteration causes it to reach the initial delay distribution [1], so the execution of an SDFG is infinite. This infinity is limited by the edges in E_f of an IP graph, however. Each actor v in SDFG G fires infinite times in an execution. In ipG_f, tokens on $e' = \langle v, v\rangle$ will be exhausted after $f \cdot q(v)$ firings of v, because each firing of v consumes one token from e' but produce no token on it. Therefore, an execution of ipG_f includes exactly f iterations of G. Then we have the following property.

Property 1. *In an execution of ipG_f, there are $\sum_{v\in V} f \cdot q(v)$ firings of actors in G.*

Denote the last state of an execution of ipG_f by s_e. It is easy to see that at state s_e, each actor v has been fired exactly $f \cdot q(v)$ times. And the value of $s_e.glbClk$ is the earliest possible completion time of f iterations of G, because the execution is ASAP. Therefore we have the following property.

Property 2. *The value of $glbClk$ at state s_e is exactly the length of the critical path of $U(H(G), f)$.*

Fig. 3 (a) and (b), for example, show the executions of the IP graphs of 1-unfolded and 2-unfolded graphs of G_1, respectively. Their longest subroutines directly correspond to the critical paths of HSDFGs in Fig. 1 (b) and (c), respectively.

The following theorem can be directly deduced from Property 2.

Theorem 1. $IP(G, f) = s_e.glbClk/f$.

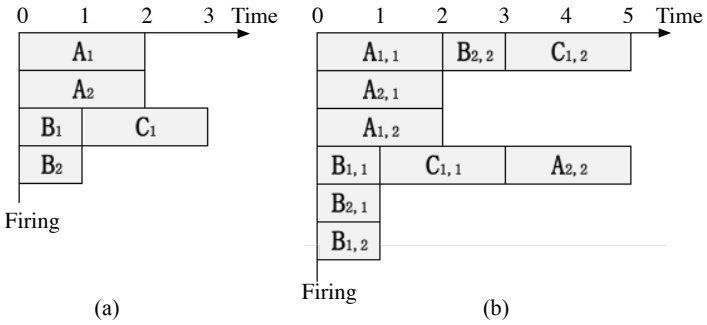

Fig. 3. Execution of the IP graph of (a) 1-unfolded graph of G_1, and (b) 2-unfolded graph of G_1.

B. Computing IP by State-Space Exploration

According to the above analysis of IP graphs, comIP, a procedure that computes the IP of an unfolded SDFG by exploring the state space of the execution of its IP graph, is outlined in Algorithm 1. The execution of the IP graph in comIP is according to macro steps. A *macro step* includes: first, starting all firings of actors that are ready to start, then one clk, and at last ending all firings of actors that are ready to end.

Algorithm 1 comIP(G, f)

Input: A valid SDFG G and an unfolding factor f
Output: The IP of the f-unfolded graph of G

```
1:  g = ipG_f // Construct the IP graph.
2:  ts = TS(g) // Construct the transition system of the IP graph.
3:  s = ts.s_0
4:  nFires = Σ_{v∈V} f · q(v)
5:  // Begin to execute the IP graph.
6:  while nFires > 0 do
7:    for all v ∈ G do
8:      while readyS(v) do
9:        sFiring(v)
10:     end while
11:   end for
12:   clk
13:   for all v ∈ G do
14:     while readyE(v) do
15:       eFiring(v)
16:       nFires = nFires − 1
17:     end while
18:   end for
19: end while
20: return s.glbClk/f // Return the IP (by Theorem 1)
```

Variable $nFires$ is set to be the remaining number of firings in the execution. Line 4 initializes it as $\sum_{v\in V} f \cdot q(v)$ (by Property 1), Line 16 decreases it by one after the end of each firing, and Line 6 checks whether all required firings are finished. Lines 6 to 19 simulate the execution of the IP graph. The termination of Algorithm 1 is guaranteed by Property 1,

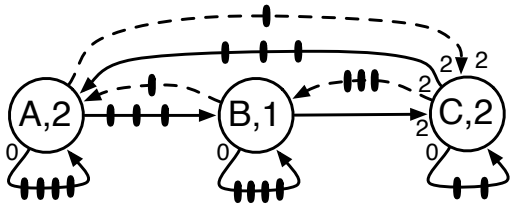

Fig. 4. IP graph of 2-unfolded graph of G_1 with buffer bound $buf = [4, 3, 3]$, corresponding to edges $\langle A, B\rangle, \langle B, C\rangle$ and $\langle C, A\rangle$. The edges modeling buffer bound are denoted by dotted lines.

and its correctness is guaranteed by Theorem 1. The number of firings of the execution is exactly the initial value of $nFires$, therefore the complexity of Algorithm 1 is $O(\sum_{v\in V} f \cdot q(v))$.

V. IP Computation with Resource Constraints

The advantage of the proposed method is that it is not only efficient but also flexible for combining different constraints into consideration.

A. With Buffer Bound

When an SDFG is constrained with buffer bound, we can also use Algorithm 1 to find the IP of its unfolded graph. The bound on buffer of an edge $\langle u, v\rangle$ of an SDFG can be modeled by adding edge $\langle v, u\rangle$ with tokens to model available storage space [13].

Definition 3. *A buffer bound SDFG of $G = (V, E)$ with buffer bound $buf(E)$ is an SDFG $G_{buf} = (V, E \cup E_{buf})$, in which $E_{buf} = \{\langle v, u\rangle | \langle u, v\rangle \in E\}$. For all $e' = \langle v, u\rangle \in E_{buf}$, there is an edge $e = \langle u, v\rangle \in E$, such that $prd(e') = cns(e)$, $cns(e') = prd(e)$ and $d(e') = buf(e) - d(e)$.*

The buffer bound SDFG is also an SDFG. The IP graph of the f-unfolded graph of the buffer bound SDFG can be constructed according to Definition 2. Fig. 4, for example, shows the IP graph of 2-unfolded SDFG of G_1 with buffer bound $buf = [4, 3, 3]$, corresponding to edges $\langle A, B\rangle, \langle B, C\rangle$ and $\langle C, A\rangle$. Then $IP_{buf}(G, f)$ can be computed by Algorithm 1.

Theorem 2. *$IP_{buf}(G, f)$ is the value returned by comIP(G_{buf}, f).*

B. With Limited Number of Processors

When there are only pro number of processors available, the number of concurrent firings in an execution of an SDFG is limited to pro. Therefore we can put the constraint on the guard of $sFiring$ actions [11]. We denote the new guard as $readyS_{pro}(v)$.

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

A procedure to obtain the IP of unfolded SDFGs with limited number of processors, called comIP$_{pro}(G, f)$, is a variation of Algorithm 1, in which $readyS$ is replaced with $readyS_{pro}$.

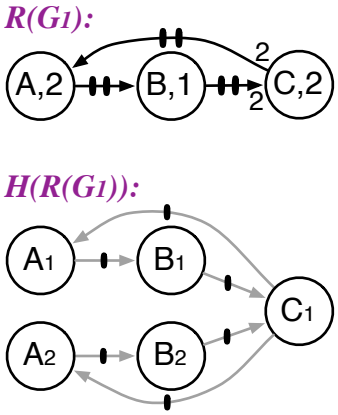

Fig. 5. A retimed graph of G_1 with retiming $R = [1, 2, 0]$ and its equivalent HSDFG.

Theorem 3. *$IP_{pro}(G, f)$ is the value returned by comIP$_{pro}(G, f)$.*

The IP of an unfolded SDFG with the combination of buffer bound and processor limitation can be computed by first constructing the buffer bound SDFG and then use the procedure comIP$_{pro}$.

Theorem 4. *$IP_{BandP}(G, f)$ is the value returned by comIP$_{pro}(G_{buf}, f)$.*

VI. Reducing IPs of Unfolded SDFGs via Retiming

The IPs of unfolded SDFGs may be further reduced by retiming. *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. 5, for example, is a retimed graph of G_1 by retiming R, which is defined as $R(A) = 1$, $R(B) = 2$ and $R(C) = 0$. The token distribution of $R(G_1)$ is led by firing A once and firing B twice of G_1. The IP of $R(G_1)$ can be easily found in its equivalent HSDFG, $H(R(G_1))$, also shown in Fig. 5. Its IP is 2, less than The IP of G_1 .

The order of retiming and unfolding is immaterial for scheduling an HSDFG [14], and retiming on an SDFG and its equivalent HSDFG is equivalent [8]. Therefore the order of retiming and unfolding is immaterial for the IP computation. That is, $IP(U(R(G), f)) = IP(R(U(G, f)))$. We can first optimally retime an SDFG to be a new SDFG, then unfold the retimed SDFG to further reduce the IP.

Let $optR$ be the optimal retiming obtained by optimal retiming algorithm in [8]. Then $reIP$, $reIP_{buf}$, $reIP_{pro}$ and $reIP_{BandP}$ can also be computed by Algorithm 1 and procedure comIP$_{pro}$.

Theorem 5. *The IP $reIP(G, f)$ is the value returned by procedure comIP($optR(G), f$), $reIP_{buf}(G, f)$ by comIP($optR(G)_{buf}, f$), $reIP_{pro}(G, f)$ by comIP$_{pro}(optR(G), f)$ and $reIP_{pro}(G, f)$ by comIP$_{pro}(optR(G)_{buf}, f)$.*

VII. Experimental Evaluation

A. Experimental Setup

We implemented Algorithm 1 and above-mentioned extensions, the IP computation algorithms in [7] and [8], and the traditional method (byHSDFG) in the open source tool SDF3 [15]. The algorithms in [7] and [8], can only be used for 1-unfolded graph. Method byHSDFG computes the IP of an unfolded graph by first converting it to an HSDFG, then unfolding the HSDFG, and at last computing the length of the critical path of the unfolded HSDFG. We performed experiments on two sets of SDFGs, running on a 2.67GHz CPU with 12MB cache. The experimental results of these two sets are shown in Tables I, II and III. All execution times are measured in milliseconds (ms) unless otherwise specified.

The first set of SDFGs consists of four practical DSP applications, including a sample rate converter (SaRate) [16], a satellite receiver (SR) [3], a maximum entropy spectrum analyzer (MaxES) [17], and a channel equalizer (CEer) [8]. Because the method in [7] can only work on strongly connected graph, we convert these models to strongly connected graphs 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.

The second set of test models consists of 200 synthetic strongly connected SDFGs generated by SDF3, mimicking real DSP applications and scaling up the models. The number of actors in an SDFG, denoted by nA, and the sum of the elements in the repetition vector, denoted by nQ, have significant impact on the performance of the various methods. We distinguish two different ranges of nA: 15-30 and 100-120; and two different ranges of nQ: 2000-3000 and 8500-11000. Then we generate SDFGs according to different combinations of nA and nQ to form 4 groups. Each group includes 50 SDFGs. The explicit difference in nA and nQ among these groups is helpful for showing how the performance of each method changes with nA and nQ.

The buffer bounds are chosen to be the smallest storage capacity that guarantees deadlock-freeness of an SDFG.

B. Experimental Results

Table I gives the information about and results for the 1-, 3- and 5-unfolded graphs of the practical DSP examples. There are four parts in Table I. The first part is the information on the DSP models, including the number of actors in an SDFG (nA), the sum of the elements in the repetition vector (nQ) and the sum of the elements in the buffer bound (buf). The second part lists the results for $(re)IP$, $(re)IP_{buf}$, $(re)IP_{pro}$ and $(re)IP_{BandP}$. The third part shows the execution times of IP, IP_{BandP}, $reIP_{BandP}$ and $byHSDFG$. We measured the memory used by IP and $byHSDFG$ on the four DSP examples using the tool Valgrind (http://valgrind.org/) and show the results in the fourth part.

The information in the first part of Table I includes the dummy actors introduced for strong connectedness.

TABLE I
The Experimental Results for DSP Examples

Graph information				
	CEer	SaRate	MaxES	SR
nA	23	7	14	23
nQ	43	613	1,289	4,516
buf	75	1,368	2,089	14,454
Returned IPs by Proposed Methods				
f	IP/$reIP$			
1	53652/47128	21/6	11528/8192	11/2
3	49302.7/47128	7/5.3	3842.7/3842.7	3.7/2
5	48432.8/47128	4.2/4.2	4611.2/4611	2.2/2
	IP_{buf}/$reIP_{buf}$			
1	53652/47328	1053/491	12293/9217	601/528
3	49302.7/47194.7	1037/490.3	10241.7/9216.3	584.3/528
5	48432.8/47168	1033.8/490.2	9831.4/9216.2	581/528
	IP_{pro}/$reIP_{pro}$ with $\#pro = 16$			
1	53652/47128	153/153	11618/8208	284/283
3	49302.7/47128	152.7/152.7	3936.7/3907	282.7/282.3
5	48432.8/47128	152.6/152.6	4704.8/4668.8	282.4/282.2
	IP_{BandP}/$reIP_{BandP}$ with $\#pro = 16$			
1	53652/47328	1053/537	12308/9232	615/542
3	49302.7/47194.7	1037/540.3	10257.3/9232	589/528
5	48432.8/47168	1033.8/538.8	9847.2/9232	583.8/528
	IP_{pro}/$reIP_{pro}$ with $\#pro = 4$			
1	53876/47128	610/610	11906/8256	1130/1129
3	49377.3/47128	610/610	4224.7/4160.7	1129/1129
5	48477.6/47128	609.8/609.8	4992.8/4890.4	1128.8/1128.8
	IP_{BandP}/$reIP_{BandP}$ with $\#pro = 4$			
1	53652/47328	1067/735	12356/9280	1207/1292
3	49302.7/47194.7	1043/724.7	10305.3/9280	1155.7/1189
5	48432.8/47168	1038.2/725.6	9895.2/9280	1145.4/1165
Execution time (ms) ($IP/IP_{BandP}/reIP_{BandP}/byHSDFG$)				
1	0/0/5/2	1/0/1/89	0/1/1/256	1/2/3/4,641
3	0/1/5/4	0/1/1/471	0/2/3/1.3s	2/6/6/50s
5	1/1/5/8	0/1/1/1,258	1/3/5/3.8s	4/9/10/158s
Memory used (MB) ($IP/byHSDFG$)				
1	0.83/0.91	0.23/1.63	0.38/3.06	0.77/11.08
3	0.83/0.99	0.23/2.96	0.39/5.52	0.93/20.70
5	0.83/1.09	0.23/4.29.	0.39/8.03.	1.07/30.90

For SaRate model, its IPs and $reIP$s are decreased with f increased. The reason is that after the optimal retiming, which explores the parallel probability in an iteration, there still remains parallel probability among inter-iteration, which are further exploited by unfolding. On the contrary, for models

CEer and SR, after retiming, there are no parallel probability for further explore among inter-iteration. Therefore, their IPs are decreased with f increased, but their $reIP$s have no change with different fs. The IPs of unfolded graph of an SDFG are not always monotonously decreased with the unfolding factor increased. For example, for MaxES model, $IP(MaxES, 5) > IP(MaxES, 3)$. But its IP will be reduced to 3842.7 again when $f = 6, 9....$ Therefore $f = 3$ is good enough for unfolding MaxES to achieve a small IP. For the cases that models with buffer bound, because the bound buf is chosen to be the smallest storage capacity that guarantees deadlock-freeness of an SDFG, there is little room left for IP optimizing. Therefore, their IP_{buf}s and $reIP_{buf}$s are improved in a very limited way by unfolding. Two numbers of processors are tested to show how the processor limitation impacts on the IPs. These examples also show that how our method can help when designing DSP algorithms. The proposed method IP is significantly faster than method byHSDFG as the execution times shown in the third part of Table I. The space efficiency of our method gets better when the sum of the elements of repetition vector of SDFGs (nQ) grows larger, e.g. the SR case (nQ=4516), as shown in the fourth part of Table I. Our execution time advantages are mainly obtained from that we do not convert an SDFG to its equivalent HSDFG and neither further convert the HSDFG to an unfolded graph. Thus, compare to byHSDFG, our method saves the time for these conversion procedures and saves the memory for storing the HSDFG and its unfolded graph.

Tables II and III give the results for the synthetic examples. Each point in the tables is an average of 50 graphs in the same group. The execution times of IP, IP_{BandP}, $reIP_{BandP}$, [8], [7] and byHSDFG are shown in Tables II. The methods in [7] and [8] are only comparable to our method IP at $f = 1$ case. In this case, the method in [7] is fastest on the average. The byHSDFG method is only comparable to our method IP without any constraints and optimization. It is already five orders of magnitude slower than IP with $f = 1$, so we do not show its execution times for $f = 2$ and $f = 3$.

Table III gives the IPs of synthetic graphs. The processor limitation is set to 16, which affects the IPs more than buffer bound does for these large graphs. The results also show that after retiming there is little room left for unfolding to improve the IPs.

TABLE II
EXECUTION TIMES (MS) FOR SYNTHETIC EXAMPLES

f	15-30	100-120	nA / nQ
$IP/IP_{BandP}/reIP_{BandP}$/[8]/[7] /byHSDFG			
1	0/1/2/0/0/1,364	5/21/42/1/1/1,307	2k-3k*
2	1/2/3/N†/N/-‡	8/44/65/N/N/-	
3	1/3/4/N/N/-	11/67/88/N/N/-	
1	2/4/5/0/0/26,007	6/74/113/1/1/24,445	8.5k-11k
2	3/8/9/N/N/-	11/168/206/N/N/-	
3	4/12/13/N/N/-	15/261/298/N/N/-	

† The methods in [7] and [8] can only be used for 1-unfolded graph, therefore they are only comparable to our method IP at $f = 1$ case.
‡ byHSDFG is already five orders of magnitude slower than IP with $f = 1$, so we do not show its execution times for $f = 2$ and $f = 3$.
* 1k=1000.

TABLE III
THE IPS OF SYNTHETIC GRAPHS

f	15-30	100-120	nA / nQ
	$IP/IP_{buf}/IP_{pro}/IP_{BandP}$		2k-3k*
1	75/101/826/840	212/227/860/875	
2	66/90/824/837	168/181/848/859	
3	63/87/824/836	154/167/844/854	
	$reIP/reIP_{buf}/reIP_{pro}/reIP_{BandP}$		
1	57/81/826/841	126/143/848/854	
2	57/80/824/838	126/140/842/849	
3	57/79/824/836	126/139/840/847	
	$IP/IP_{buf}/IP_{pro}/IP_{BandP}$		8.5k-11k
1	78/110/2957/2969	202/227/3077/3087	
2	67/99/2954/2964	160/183/3075/3082	
3	64/96/2953/2963	147/170/3074/3080	
	$reIP/reIP_{buf}/reIP_{pro}/reIP_{BandP}$		
1	57/89/2954/2964	119/143/3077/3082	
2	57/88/2953/2962	119/139/3076/3080	
3	57/88/2952/2961	119/138/3074/3079	

* 1k=1000.

VIII. CONCLUSION

In this paper, we have presented an algorithm for IP computation of unfolded SDFGs. The proposed method uses the state-space exploration technique. It works directly on SDFGs, without converting them to their equivalent HSDFGs, and without further unfolding the HSDFGs. This method can be easily extended to deal with cases when resource constraints are enforced. Buffer bound and processor limitation are used to exemplify such case. Combining with retiming technique, the reduced IP of unfolded SDFGs can be found using the proposed algorithm. Our experimental results show that the proposed method outperforms the existing method significantly.

A schedule that achieves the IP returned by the proposed method can be obtained by the information of the execution, similar to that in [11]. And, also, more constraints that may affect the IP of unfolded SDFGs can be taken into account in the framework of the proposed method. Furthermore, optimizing the memory use of models by eliminating buffers [18] or by reducing the sizes of buffers [19] are helpful for practical

use of SDFGs. These will be investigated in the future work.

IX. Acknowledgments

The author would like to thank the anonymous referees for their valuable comments and helpful suggestions.

References

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

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

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

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

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

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

[8] X.-Y. Zhu, T. Basten, M. Geilen, and S. Stuijk, "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.

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

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

[11] X.-Y. Zhu, M. Geilen, T. Basten, and S. Stuijk, "Static Rate-Optimal Scheduling of Multirate DSP Algorithms via Retiming and Unfolding," in *Proc. 18th Real-Time and Embedded Technology and Applications Symp.*, 2012, pp. 109–118.

[12] ——, "Multiconstraint static scheduling of synchronous dataflow graphs via retiming and unfolding," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 6, pp. 905–918, June 2016.

[13] 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.

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

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

[16] 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.

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

[18] Y. Ko, B. Burgstaller, and B. Scholz, "Laminarir: Compile-time queues for structured streams," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15. New York, NY, USA: ACM, 2015, pp. 121–130. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737994

[19] X.-Y. Zhu, M. Geilen, T. Basten, and S. Stuijk, "Memory-constrained static rate-optimal scheduling of synchronous dataflow graphs via retiming," in *Proc. of the 17th Design, Automation and Test in Europe (DATE)*, 2014, pp. 1–6.