Retiming Synchronous Dataflow Graphs through a State-Space Exploration

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

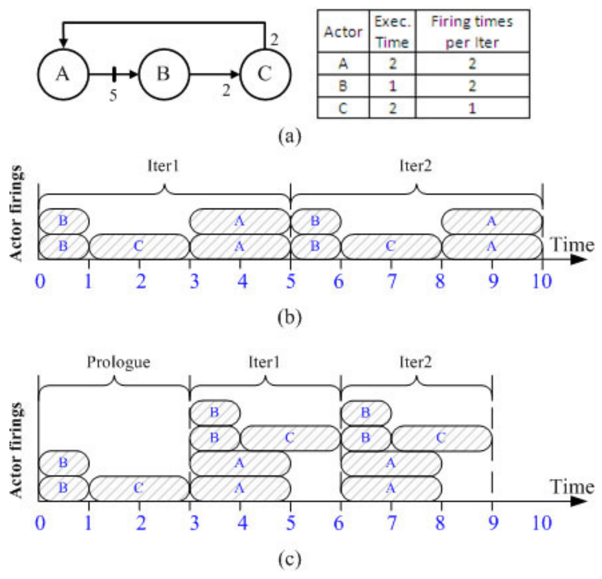
Marc Geilen, Twan Basten, and Sander Stuijk
Department of Electrical Engineering
Eindhoven University of Technology
Eindhoven, the Netherlands
{a.a.basten, m.c.w.geilen, s.stuijk}@tue.nl

ABSTRACT

Synchronous dataflow graphs (SDFGs) are widely used to model multi-rate digital signal processing (DSP) algorithms. A lower iteration period of such a model implies a faster execution of a DSP algorithm. Retiming is a simple but efficient graph transformation technique for performance optimization, which can decrease the iteration period without affecting functionality. In this paper, we deal with the iteration period minimization problem — retiming an SDFG to achieve the smallest possible iteration period. We present a heuristic method that works directly on SDFGs, without converting them to their equivalent homogeneous SDFGs. It analyzes the state-space generated by a self-timed execution of the SDFG to obtain a near-optimal retiming. Our experimental results show that in 85% of the test cases that allow actors to fire auto-concurrently, our method gets reduced iteration periods close to the optimal ones, while being ten times faster than the state-of-the-art exact method; in all the test cases in which auto-concurrent firing of actors is excluded, our method gets reduced iteration periods almost the same as the optimal ones, while being 100 times faster than the exact method. Combining parts of the exact method with our novel method, we present an improved algorithm, whose execution time is further reduced by 22%.

1. INTRODUCTION

Dataflow models of computation are widely used to represent DSP applications. Each node (also called actor) in such a model represents a computation or function and each edge models a FIFO channel. One of the most useful dataflow models for designing multi-rate DSP algorithms are *synchronous dataflow graphs* (SDFGs) [8]. The sample rates of actors of an SDFG may differ. The graph G_1 in Fig. 1(a), for example, is an SDFG model. The real multi-rate DSP algorithms modeled with SDFGs include the simplified spectrum analyzer [22], the satellite receiver [16], etc.



Actor	Exec. Time	Firing times per Iter
A	2	2
B	1	2
C	2	1

Figure 1: (a) The SDFG G_1, where the sample rates are omitted when they are 1; (b) the periodic schedule of G_1; (c) a schedule with prologue and improved iteration period.

DSP algorithms are often repetitive. Execution of all the computations for the required number of times is referred to as an *iteration*. A DSP algorithm repeats iterations periodically. An iteration of G_1 in Fig. 1(a), for example, includes two executions, often called *firings* in dataflow, of actor A, two firings of B and one firing of C. The *iteration period* is the least time required for executing one iteration of a DSP algorithm [15]. The iteration period of G_1, for example, is 5 as shown by the periodic schedule of Fig. 1(b). The data dependencies among actors B, C and A imply that the iteration period of G_1 cannot be lower than 5 no matter which periodic schedule is used.

Consider the SDFG G_1 in Fig. 1(a) again. A schedule of G_1 with

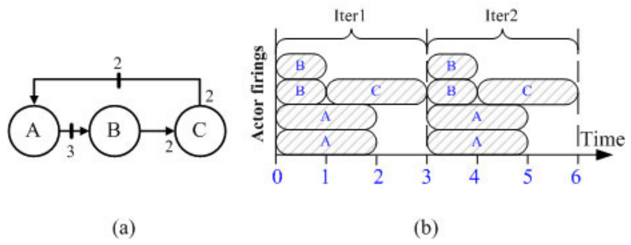


Figure 2: (a) The retimed graph of G_1; (b) its periodic schedule.

prologue, shown in Fig. 1(c), consists of some initial actor firings followed by iterations whose execution time is shorter than the iteration period of G_1. Executing the prologue of G_1, that is, firing actor B twice and firing C once, respectively, leads to the graph in Fig. 2(a), whose delays are redistributed and whose iteration period is reduced to 3, as its periodic schedule, Fig. 2(b), shows.

Fig. 2(a) is in fact a graph obtained by *retiming* G_1 in Fig. 1(a), and it is called a retimed graph of G_1. Retiming is a graph transformation technique that only changes the delay distribution of a graph, while it has no effect on its functionality [9]. The *delay distribution* indicates the numbers of initial tokens on edges of the SDFG. A *retiming function* specifies the firings required to redistribute the delays. Retiming is originally applied to reduce the iteration period of *homogeneous synchronous dataflow graphs* (HSDFGs), which is a special type of SDFGs. In an iteration of an HSDFG, each actor fires once.

As this example shows, retiming an SDFG is actually executing some of its actors. This inspired us to think about a retiming method that analyzes behaviors of SDFGs. A self-timed execution (STE) [6] results in a nicely pipelined execution [4] and a snapshot of such a pipelined execution is likely to have a low iteration period. In this paper, we present a heuristic retiming algorithm based on the state space generated by a self-timed execution of an SDFG. The purpose is to find a retiming for an SDFG such that the retimed graph has an iteration period as small as possible. Our method does not convert an SDFG to its equivalent HSDFG.

For evaluating our new method, *STE*, we implemented it and the exact optimal retiming algorithm that works directly on SDFGs [10], *LLW*, in the open source tool SDF3 [20]. The experiments were carried out on thousands of synthetic SDFGs and several realistic SDFG models. Although our method is not guaranteed to provide an optimal retiming, the experimental results show that in 85% of the test cases that allow auto-concurrent actor firing, it reaches a reduced iteration period very close to what LLW reports; and that in all the test cases that disallow auto-concurrent actor firing, our method reaches a reduced iteration period almost the same as the optimal one. Our method is ten times and 100 times faster than LLW, on the test cases with and without auto-concurrency, respectively.

Combining parts of LLW and STE, we present an improved algorithm with the same results as STE but a 22% reduced execution time.

The remainder of this paper is organized as follows. We first discuss the related work in the next section and introduce synchronous dataflow graphs in Section 3. The problem is formulated in Section 4. An operational semantics of SDFGs is defined in Section 5. Our retiming method is presented in Section 6. Section 7 provides an experimental evaluation. An improved algorithm and its experimental results are presented in Section 8. Finally, Section 9 concludes.

2. RELATED WORK

A great deal of research has been done on retiming in the context of HSDFGs [18, 3, 17]. Different from an HSDFG, however, in an iteration of an SDFG, actors may be executed more than once and a different number of times. This invalidates many useful results derived for HSDFGs, and complicates the analysis of retiming properties of SDFGs. Nevertheless, some efforts have been made. Some important properties of retiming on SDFGs, such as functional equivalence and reachability, have been proven [23, 22].

A typical solution for the retiming problem includes two main subroutines: computing the iteration period and finding a proper retiming function. The traditional way to solve retiming problems on SDFGs is to first convert the SDFG to its equivalent HSDFG and then to use the available methods for HSDFGs [7]. Theoretically, this is always possible. However, converting an SDFG to an HSDFG may increase the problem size tremendously and it is very time-consuming when SDFGs become larger. The size of the HSDFG can be exponentially larger than the original SDFG in extreme cases [23]. There exists work that tries to solve the retiming problem without or with limited conversions of the SDFG to HSDFG. [14] and [21] present methods to find a retiming of an SDFG such that the retimed SDFG has an iteration period at most a given value. The method in [14] computes the retiming function directly on SDFGs, but the computation for the iteration period is still on HSDFGs. The method in [21] computes both the iteration period and retiming function on SDFGs. Liveris et al. [10] present an optimal retiming method that computes both the iteration period and retiming function on SDFGs. To the best of our knowledge, it is the only optimal retiming method that works directly on SDFGs.

All the above-mentioned methods provide solutions from a static viewpoint — analyzing structures of SDFGs. They are explicitly based on the relationship between an SDFG and its equivalent HSDFG, which is quite complex. Our method provides a solution from a dynamic viewpoint — analyzing behaviors of SDFGs. It is intuitive and efficient.

3. SYNCHRONOUS DATAFLOW GRAPHS

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

- V is the set of actors, modeling the functional elements of the system. 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. The source actor and sink actor of $e \in E$ are denoted as $src(e)$ and $snk(e)$, respectively.

We represent the edge e with source actor u and sink actor v 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 $OutE(v)$. We use $v \in G$ to represent that v is an actor of G. 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 $t = [2, 1, 2]$. The initial delay distribution $d(G_1) = [5, 0, 0]$, corresponding to the 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 $q(V)$ such that for each edge e in G,

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

where (1) is called a *balance equation*. The smallest q is called the *repetition vector* [8]. 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 least time required for executing one iteration, represented by $IP(G)$.

For example, a balance equation can be constructed for each edge of G_1 in Fig. 1. By solving these balance equations, we have G_1's repetition vector $q = [2, 2, 1]$. Actor A can start to fire only after C fires once, while C can start to fire only after B fires twice. Therefore the time required for executing one iteration of G_1 is at least 5, that is, $IP(G_1) = 5$, as Fig. 1(b) shows.

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*.

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 [9, 22]. We therefore only consider valid SDFGs that are strongly connected, when developing our ideas.

4. PROBLEM FORMULATION

Retiming is a graph transformation technique that redistributes the graph's delays while its functionality remains unchanged [9, 22]. Retiming can be defined either in a forward fashion, by which retiming an actor once means firing this actor once [22, 14], or in a backward fashion, by which retiming an actor once means reversed firing this actor once [9, 10, 21]. We use the former one.

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

$$d_r(e) = d(e) + prd(e)r(u) - cns(e)r(v). \tag{2}$$

A retiming r of a valid SDFG is *legal* if the retimed graph $r(G)$ is a valid SDFG. Only legal retimings are meaningful. Any solution for retiming problems need to guarantee a legal retiming. It is sufficient to check if the initial delay distribution $d(r(G))$ is nonnegative to ensure that a retiming is legal [21].

We deal with the iteration period minimization problem — to find a retiming r of G such that it is legal and $IP(r(G))$ is as small as possible. We call a retiming r an *optimal retiming* if $IP(r(G))$ is the *optimal iteration period* (optIP) of G — the smallest iteration period that can be reached by retiming G. $R_1 = [0, 2, 1]$ is an optimal retiming and 3 is the optIP for G_1 in Fig. 1(a), for example, because no lower IP can be reached by any retiming of G_1. For an SDFG, there is only one optIP, but the SDFG may have more than one optimal retiming.

5. AN OPERATIONAL SEMANTICS OF SDFGS

In this section, we first define the operational semantics of SDFGs according to transition systems; then we introduce the important properties of the self-timed execution.

5.1 Transition System of SDFGs

For developing our method, we define the behavior of an SDFG G in terms of a simple *transition system* (TS), 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 the current number of delays on edge e. The SDFG is a concurrent model of computation. It allows simultaneous firings of an actor. For different concurrent firings of an actor, the one first to start is the one first to end. We use a queue $tr(v)$ to contain the remaining times of the concurrent firings of actor v. The i^{th} element of $tr(v)$ is the remaining time of the i^{th} unfinished firing of v. The number of elements of $tr(v)$, $sizeOf(tr(v))$, is the number of concurrent firings of v. For a queue qu, the queue operations we use include: $HeadQ(qu)$ to return the first element of qu; $ENQ(qu, x)$ to insert x at the end of qu; $DLQ(qu)$ to remove the first element of qu; and $isEmpty(qu)$ to test if qu is empty.

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

Our purpose is to improve the time performance of an SDFG, so we need to hold two other variables: a global clock, *glbClk*, to record the time progress, and the vector $nStarted(V)$ to record the number of started firings of each actor. We will get the IP from *glbClk* and the retiming function from $nStarted(V)$ in a certain state.

A *state* of $TS(G)$ is a 4-tuple that consists of the values of the delay distribution vector $tn(E)$, the firing status vector $tr(V)$, the vector $nStarted(V)$, and the global clock *glbClk*. We call $tn(E)$ and $tr(V)$ the *behavior elements* of states.

There is only one *initial state* of $TS(G)$, denoted as s_0. In s_0, $tn(E)$ is the initial delay distribution $d(G)$; no firings have been started, so each element of $tr(V)$ is empty and $nStarted(V)$ is the zero vector; and the global clock *glbClk* is zero. For example, the initial state of G_1 in Fig. 1(a) is $gs_0 = ([5, 0, 0], [\{\}, \{\}, \{\}], [0, 0, 0], 0)$, corresponding to the edges $\langle A, B\rangle, \langle B, C\rangle$ and $\langle C, A\rangle$ and the 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,

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

An actor v starting a firing, $sFiring(v)$, is to insert its computation time, $t(v)$, into queue $tr(v)$, to reduce delays of all its incoming edges according to the consumption rates, and to increase $nStarted(v)$ by one. That is,

$$\begin{aligned} sFiring(v) \equiv_{def} \quad & (\forall e \in InE(v) : tn'(e) = tn(e) - cns(e)) \\ & \wedge tr'(v) = ENQ((tr(v), t(v)) \\ & \wedge nStarted'(v) = nStarted(v) + 1. \end{aligned}$$

Where $tn'(e)$, $tr'(v)$ and $nStarted'(v)$ refer to the value of $tn(e)$, $tr(v)$ and $nStarted(v)$ in the new state s', respectively. For conciseness, We omit the elements of states if their values unchanged after an action. In G_1, for example, at the initial state gs_0, B is ready to start its firing; after $sFiring(B)$, the state is changed from gs_0 to $gs_1 = ([4, 0, 0], [\{\}, \{1\}, \{\}], [0, 1, 0], 0)$.

Time progresses when no actor is ready to end. A time step, clk, reduces the remaining times of all firing actors by one if there are firing actors, and increases the global clock by one. That is,

$$\begin{aligned} clk \equiv_{def} \quad & (\forall v \in V : \neg isEmpty(tr(v)) \Rightarrow (\forall x \in tr(v) : x' = x - 1)) \\ & \wedge glbClk' = glbClk + 1. \end{aligned}$$

For example, in G_1, after actor B starts firing, no actors are ready to end, then time progresses one step, leading to the state changed from gs_1 to $gs_2 = ([4, 0, 0], [\{\}, \{0\}, \{\}], [0, 1, 0], 1)$. Its enabled conditions guarantee that a clk action never leads to a negative $tr(v)$. Notice that the delay distribution remains unchanged by a time step.

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

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

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

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

In G_1, at the state gs_2, actor B is ready to end; after $eFiring(B)$, the state is changed from gs_2 to $gs_3 = ([4, 1, 0], [\{\}, \{\}, \{\}], [0, 1, 0], 1)$.

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 an SDFG G is an infinite sequence of states of $TS(G)$ beginning with the initial state and following by states caused by transitions from their predecessors. For example, states gs_0, gs_1, gs_2 and gs_3 form a prefix of an execution of G_1.

A *self-timed execution* (STE), is an execution in which the time step, clk, only occurs when no actors are ready to start [6]. That is,

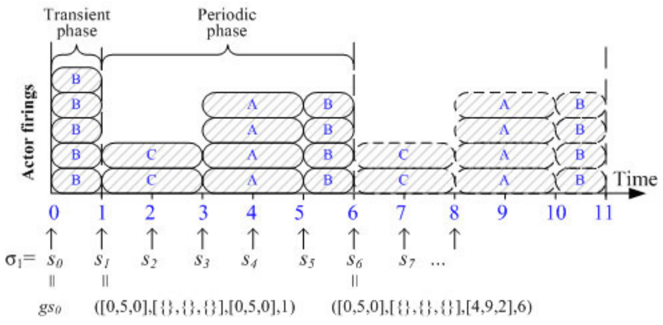


Figure 3: A self-timed execution state space of G_1.

in an STE, firings of actors start immediately if they are enalbed. The time step cannot progress if only there are actors ready to start a firing. For example, at state gs_1, actor B is still ready to start a firing; therefore state gs_2 is not a successor of gs_1 in an STE.

5.2 Properties of STEs

In an STE, between two clk actions, there can be some interleaving of simultaneous $sFiring(v)$ and/or $eFiring(v)$ actions. However, no matter what order of these actions are selected, the sequences of actor firings according to any STE of a strongly connected valid SDFG are the same. For example, Fig. 3 shows the only sequence of actor firings of all STEs of G_1. Self-timed SDFG behavior is therefore deterministic if we only consider the clk actions and its effects on the actor firing sequence [4].

When we say two states are *equal in behavior*, denoted as $s_1 =_{bh} s_2$, the values of their behavior elements, $tn(E)$ and $tr(V)$, are equal. For example, in Fig. 3, $s_1 =_{bh} s_6$.

For a strongly connected valid SDFG, self-timed execution is deterministic and the values of $tn(E)$ and $tr(V)$ are finite; therefore, an STE includes in fact finite states that are distinct in behavior. Observed only on the behavior elements, the state-space of an STE includes a finite sequence of states, called the *transient phase*, followed by an infinite sequence that is periodically repeated, called the *periodic phase* [4]. That is the following theorem.

Theorem 1. *For any STE σ of a strongly connected valid SDFG, we have $hasCycle(\sigma) = true$, where*

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

The first pair of states of σ to make $hasCycle(\sigma)$ true is the beginning state of the periodic phase, denoted as s_b, and the end state of the periodic phase, denoted as s_e. For example in σ_1 in Fig. 3, $s_b = s_1$ and $s_e = s_6$.

According to Theorem 1, although an STE is infinite, we can find it in finitely many steps of exploration: beginning with the initial state s_0 and ending at s_e. We directly call such a finite state sequence an STE in the next section.

6. RETIMING ALGORITHM

In this section, we first present a procedure for exploring the state space of an STE according to so-called macro steps, as explained below; then based on variations of this basic procedure, a precise algorithm for computing the IP and a heuristic algorithm for finding a retiming for iteration period minimization are presented.

6.1 State-Space Exploration

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: 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. Because we assume that the computation time of each actor is positive, it is guaranteed that a *clk* action in a macro step is enabled only when no actors are ready to start or to end a firing. That is, the enabled condition for *clk* in an STE is preserved. Algorithm 1 is a procedure for exploring the state space of an STE according to macro steps.

Algorithm 1 STE(G)

Input: A strongly connected valid SDFG G
Output: The STE σ of G

```
 1: $ts = TS(G)$
 2: $s = ts.s_0$
 3: while not $hasCycle(\sigma)$ do
 4:     for all $v \in G$ do
 5:         while $readyE(v)$ do
 6:             $eFiring(v)$
 7:         end while
 8:     end for
 9:     $\sigma \leftarrow s$ // store a state
10:     for all $v \in G$ do
11:         while $readyS(v)$ do
12:             $sFiring(v)$
13:         end while
14:     end for
15:     $clk$
16: end while
17: return $\sigma$
```

The termination of Algorithm 1 is guaranteed by Theorem 1. States are stored after each macro step. We can even store only the states that caused by the end firing action of a certain actor to reduce the memory and time used. If a zero computation time is allowed, we need only to repeat lines 4-14 of Algorithm 1 until no actors are ready to end a firing to get an STE. Only when all actors in a strongly connected valid SDFG are with zero computation time, this procedure may be non-terminating. But such a model is meaningless in practice.

Placed different constraints on enable conditions of the start firing action, *readyS*(v), the procedure for the state space exploration can be used straightforwardly to compute the iteration period and the retiming function for a given SDFG.

6.2 Computing the Iteration Period

When retiming is used to reduce the IP, a subroutine for computing the IP is needed. One way to compute the IP of an SDFG is to convert it to its equivalent HSDFG and then compute the IP of the HSDFG [9]. This is exactly the method that is used by the traditional retiming method and the method in [14]. It is time and space consuming due to the conversion procedure from an SDFG to an HSDFG. Another way to compute the IP is by searching the edges of the SDFG, without converting it to an HSDFG. [10] and [21] use this method. This method is much more efficient than the former one. However, because of the complex relationship between

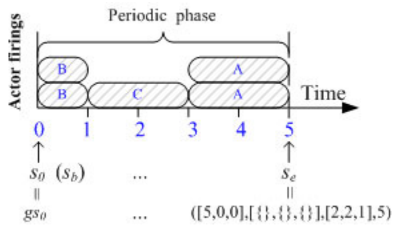


Figure 4: The ipSTE of G_1.

an SDFG and its equivalent HSDFG, the method is not intuitive. Here we present a method to compute the IP of an SDFG through a state-space exploration of an STE. The method is straightforward yet efficient. Although it is not as fast as the IP computation procedures of [10] and [21], it is much faster than that of [14].

A self-timed execution is also known as an as-soon-as-possible execution [12]. The IP of an SDFG is the earliest possible completion time of the execution of one iteration. If an STE is blocked to go only one iteration, the time passed is the IP and then the STE shutters on *clk* steps. We model this constraint by limiting the started firings of each actor v, *nStarted*(v), to $q(v)$ in an STE. Then the exploration procedure stops exactly after one iteration has been executed and the number of *clk* actions is exactly the IP of the SDFG. We put this constraint on the guard of *sFiring* action as follows.

$$readyS_{ip}(v) \equiv_{def} \quad readyS(v) \wedge (nStarted(v) < q(v)).$$

We call such an STE an *ipSTE*. Fig. 4, for example, is the ipSTE of G_1. The procedure to obtain an ipSTE of G, *ipSTE*(G), is a variation of Algorithm 1, in which *readyS*(v) is replaced with $readyS_{ip}(v)$.

It is easy to see that in ipSTE σ, $s_e =_{bh} s_0$, because executing an SDFG one iteration causes it to reach the initial delay distribution [8]; and at state s_e, each actor v has been fired exactly $q(v)$ times and the value of *glbClk* is the earliest possible completion time of G, i.e. the IP of G. That is:

$$\sigma.s_e.nStarted(V) = q(V) \text{ and}$$
$$\sigma.s_e.glbClk = IP(G).$$

The procedure to get the IP of an SDFG is shown in Algorithm 2.

Algorithm 2 getIP(G)

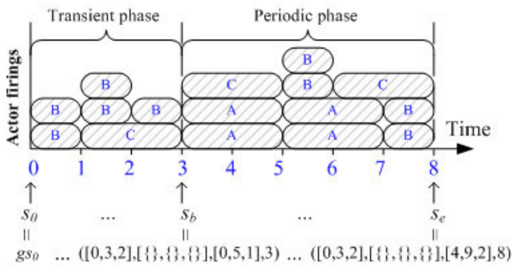
Input: A strongly connected valid SDFG G
Output: The IP of G

```
1: $\sigma = ipSTE(G)$
2: return $\sigma.s_e.glbClk$
```

6.3 Iteration Period Minimization

In either the traditional method or LLW, the procedure for finding an optimal retiming is repetitive — testing whether a legal retiming exists for a potential optimal IP until finding one for the smallest IP. Here we present a single pass heuristic method for finding a retiming that makes the IP as small as possible.

For an SDFG, there exists a lower bound on the iteration period, which can be approached by a self-timed execution [19, 5] and is determined by the period phase of an STE of this SDFG [4]. For example, the periodic phase of the STE of G_1, shown in Fig. 3, includes a pipelined execution of two iterations of G_1 and takes



Transient phase
Periodic phase
Actor firings
Time
s_0
s_b
s_e
gs_0 ... ([0,3,2],[{},{},{}],[0,5,1],3) ... ([0,3,2],[{},{},{}],[4,9,2],8)

Figure 5: The optSTE of G_1.

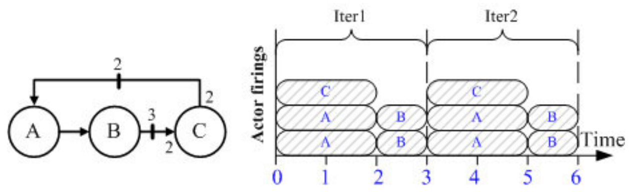


Figure 6: The retimed graph of G_1 according to $R_2 = [0, 5, 1]$ and its periodic schedule.

only 5 time units. On average, each iteration takes $\frac{5}{2}$ time units, which is the iteration period bound of G_1.

Since an STE, in which the number of concurrent firings of each actor is not limited, settles in such a nicely pipelined execution, an STE that is constrained by limiting the number of concurrent firings of each actor v to $q(v)$, is likely to lead to a periodic execution of one iteration with a low IP. We model this constraint by limiting each $sizeOf(tr(v))$ to $q(v)$ and put it on the guard of *sFiring* actions as follows.

$$readyS_{opt}(v) \equiv_{def} \quad readyS(v) \wedge (sizeOf(tr(v)) < q(v)).$$

We call such an STE an *optSTE*. Fig. 5, for example, is the optSTE of G_1. The procedure to obtain the optSTE of G, $optSTE(G)$, is a variation of Algorithm 1, in which $readyS(v)$ is replaced with $readyS_{opt}(v)$.

Executing the transient phase of the optSTE of G_1, i.e., firing 5 times of B and once C, leads to a new graph, shown in Fig. 6, with a lower IP. In the optSTE of an SDFG, the vector $nStart(V)$ of state s_b records the number of firings of each actor in the transient phase.

Based on the observation, the procedure trying to find a retiming to minimize the IP of an SDFG is shown in Algorithm 3. It computes *optSTE* of SDFG G first to get retiming function r; next, it retimes G by r to get the retimed SDFG $r(G)$; then it computes $ipSTE$ of $r(G)$ to get the IP of the retimed SDFG; at last, it returns r and $IP(r(G))$. It computes two STEs.

```
Algorithm 3 optRe(G)
Input: A strongly connected valid SDFG G
Output: An estimated retiming and IP of G
1: $\sigma = optSTE(G)$
2: $\forall v \in G$, let $r(v) = \sigma.s_b.nStarted(v)$
3: $G_r = r(G)$
4: $IP = getIP(G_r)$
5: return r and IP
```

According to the semantics we defined in Section 5, the delay distribution decreases only after an *sFiring* action. The enabled condition of *sFiring* guarantees that the delay distribution never goes negative. The retiming function returned by Algorithm 3 is an accumulation of *sFiring* actions, and therefore never leads to a retimed SDFG with negative delay distribution. That is, the retiming returned by Algorithm 3 is guaranteed to be legal.

7. EXPERIMENTAL EVALUATION

In this section we present our experimental results to evaluate our method.

7.1 Experimental Setup

We implemented our new method, *STE*, and the second algorithm from [10] (the improved version), *LLW*, in the open source tool SDF3 [20]. We performed experiments on four sets of SDFGs, running on a 2.8GHz CPU with 1MB cache. The experimental results of these four sets are shown in Tables 1, 2 and 3, and Fig. 7. All execution times are measured in milliseconds (ms).

The first set of SDFGs consists of four practical DSP applications, including a sample rate converter (Samplerate) [13], a satellite receiver (Satellite) [16], a maximum entropy spectrum analyzer (MaxES) [1], and a channel equalizer (CEer); the latter is converted from the cyclo-static dataflow model [2] in [11]. Adopting the method in [22], by introducing to each model a dummy actor with computation time zero and edges with proper rates and delays to connect the dummy actor to the actors that have no incoming edges or no outgoing edges, we convert these models to strongly connected graphs.

The second set of test models consists of 1200 synthetic strongly connected SDFGs generated by SDF3, mimicking real DSP applications and scaling up the models. The number of actors in an SDFG, denoted as *nA*, and the sum of the elements in the repetition vector, denoted as *sumQ*, have significant impact on the performance of the various methods. We distinguish three different ranges of *nA*: 15-30, 50-65, and 100-120; and four different ranges of *sumQ*: 2000-3000, 4000-6000, 8500-11000, and 18000-22000. Then we generate SDFGs according to different combinations of *nA* and *sumQ* to form 12 groups. Each group includes 100 SDFGs. The explicit difference in *nA* and *sumQ* among these groups is helpful for showing how the performance of each method changes with *nA* and *sumQ*.

There are some applications in which an actor holds internal states or has a self-loop. We model this situation by adding a self-loop with one delay to each actor of graphs in the two sets of real and synthetic graphs to form another two sets. Adding self-loops in this way excludes auto-concurrency, i.e., it disallows concurrent firings of the same actor.

Table 1 gives the information about and results for the practical DSP examples. There are three parts in Table 1. The first part is the information on the graphs, including the number of actors in an SDFG (*nA*), and the sum of the elements in the repetition vector, (*sumQ*); the second part and the third part list the results for the models without and with self-loops, respectively. Both the latter two parts include the initial iteration period (*initIP*) of the model, the best/optimal iteration periods obtained with different methods, and the execution times of different methods. The information in the first part of Table 1 includes the dummy actors introduced for strong connectedness. Fig. 7 and Tables 2 and 3 give the results for the synthetic examples. Fig. 7 gives the results for quality evalua-

Table 1: Experimental results for practical DSP examples

Graph information				
name	Samplerate	Satellite	MaxES	CEer
nA	7	23	14	23
sumQ	613	4,516	1,289	43
Without self-loops				
Initial IP and Best/optimal IP				
initIP	21	11	11,528	53,652
llwOPT	6	2	8,192	47,128
steBEST	8	4	8,192	47,128
Execution time(ms)				
LLW	6.5	61.7	9.4	1.8
STE	3.1	8.9	0.8	0.8
With self-loops				
Initial IP and Best/optimal IP				
initIP	1,000	1,314	12,293	53,652
llwOPT	960	1,056	8,192	47,128
steBEST	960	1,056	8,192	47,128
Execution time(ms)				
LLW	485.2	4,017.4	553.4	2.3
STE	8.4	9.3	3.0	0.7

tion. Tables 2 and 3 give the results for execution time evaluation. Each point in the tables is an average of 100 graphs in the same group.

7.2 Quality Evaluation

It is not guaranteed to get a precise optimal retiming from STE, since it is a heuristic method. To evaluate the quality of our method, we normalize the best IP returned from STE, *steBEST*, of each graph according to the optimal IP returned from LLW, *llwOPT*, using the following formula:

$$N = \frac{llwOPT}{steBEST}.$$

The normalized *steBEST*, N, measures how close *steBEST* is to *llwOPT*. It approaches one when *steBEST* approaches *llwOPT*.

For the four practical DSP examples, in the cases without self-loops, two of them have *steBEST* as good as *llwOPT*, as the second part of Table 1 shows; in the cases with self-loops, all have *steBEST* as good as *llwOPT*, as the third part of Table 1 shows.

The results of the synthetic examples are shown in Fig. 7. The number of graphs is depicted on the horizonal axis and the quality of our method represented by N as defined above is depicted on the vertical axis. In the cases without self-loops, from the 1200 examples, 38% have $N = 1$, that is, these *steBEST*s are as good as *llwOPT*s; 40% have $0.9 \leq N < 1$. In total, 85% of the Ns is larger than 0.8. In the cases with self-loops, for all the synthetic examples, $N > 99\%$; for 70% of them, *steBEST*s are as good as *llwOPT*s. The different ranges of *nA* and *sumQ* have no important impact on the quality of STE.

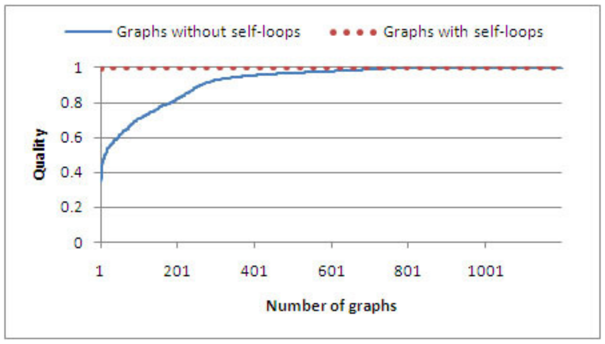


Figure 7: Quality of STE on synthetic examples.

The above results show that STE has much better quality on the SDFGs with self-loops than the SDFGs without self-loops.

7.3 Execution Time Evaluation

The execution times of STE and LLW for the practical DSP examples is shown in the second and the third parts of Table 1. Our method is more efficient than LLW for all examples, especially for Satellite. From these practical DSP examples, it seems that the larger the difference between *nA* and *sumQ*, the more efficient our method is in comparison to LLW. Our experiments on the synthetic examples also confirm this conclusion.

Table 2: Execution times (ms) for synthetic examples without self-loops

	15-30	50-65	100-120	*nA* / *sumQ*
LLW	15	24	48	2k-3k*
STE	3	4	13	
LLW	33	50	92	4k-6k
STE	4	9	15	
LLW	89	89	129	8.5k-11k
STE	9	11	20	
LLW	269	271	425	18k-22k
STE	16	20	35	

* 1k=1000.

The execution times for the synthetic examples is shown in Tables 2 and 3, for models without and with self-loops, respectively. It is clear that our method gains a greater advantage when the difference between *sumQ* and *nA* grows. For example, when *nA* is in 15-30 (the smallest) and *sumQ* is in 18000-22000 (the largest) , STE gets the greatest advantage in both kinds of models. The execution time of LLW is affected more by the growth of *sumQ* than the growth of *nA*. In the cases with self-loops, it even decreases with *nA*. There is no obvious different effect on STE.

Our method is about ten times faster than LLW in the cases without self-loops and about 100 times faster in the cases with self-loops. The difference is more significant for larger models and/or when

Table 3: Execution times (ms) for synthetic examples with self-loops

	15-30	50-65	100-120	*nA* / *sumQ*
LLW	11,278	6,645	5,945	2k-3k[*]
STE	19	69	451	
LLW	32,878	34,835	25,445	4k-6k
STE	30	140	533	
LLW	127,597	106,384	84,953	8.5k-11k
STE	210	260	1,053	
LLW	524,645	471,700	389,291	18k-22k
STE	152	655	1,884	

[*] 1k=1000.

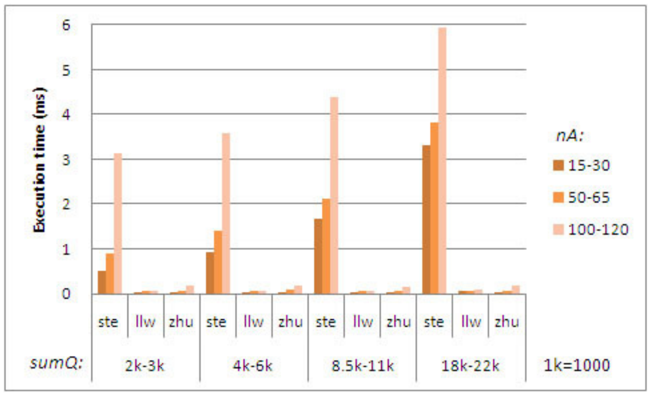


Figure 8: Execution times for the IP of synthetic examples without self-loops.

retiming is done frequently as part of an encompassing exploration process.

8. AN IMPROVED ALGORITHM

In the retiming algorithms, the procedure for computing the IP of SDFGs affects the efficiency of whole procedure. We evaluate our *getIP* procedure separately by comparing it with the procedures for the IP computation of various retiming methods in [14], [10] and [21], denoted as *nsIP*, *llwIP* and *zhuIP*, respectively. The results of the various method, except for neilIP, on the synthetic examples without self-loops are shown in Fig 8. The procedure nsIP is much slower than the other three, so we do not show it in the figure. The results are similar for the models with self-loops. The procedure llwIP is the fastest among them.

Based on this observation, we replace the procedure *getIP* in Algorithm 3 with llwIP [10] to get an improved version of our method, denoted as *STEllw*. The quality of STEllw is the same as that of STE. STEllw is about 18% faster than STE on the synthetic examples without self-loops and 27% faster on the synthetic examples with self-loops. We show the results of the graphs without self-loops in Fig 9.

9. CONCLUSION

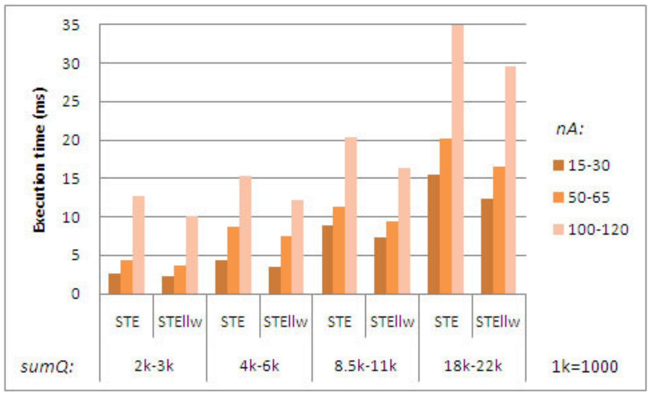


Figure 9: Execution times for the best IP of synthetic examples without self-loops.

In this paper, we have presented a heuristic retiming method trying to minimize the iteration period of an SDFG. Our method computes the iteration period and the retiming function through analyzing the state space generated by a self-timed execution of the SDFG. It works directly on the SDFG without converting it to its equivalent HSDFG.

Experimental results show that in 85% of the test cases that have no self-loops, our method reports the best iteration periods close to the optimal ones and it is ten times faster than the exact method of [10]; in all test cases that disallow auto-concurrency via self-loops, our method gets iteration periods that are almost the same as the optimal ones, while being 100 times faster than the method of [10]. By combining our method with the iteration period computation of [10], we achieve the same quality and further reduce the execution time by another 22%. Our method has a greater advantage when the models scale up.

10. REFERENCES

[1] http://ptolemy.eecs.berkeley.edu/.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cyclo-static dataflow. *IEEE Trans. on signal processing*, 44(2):397–408, 1996.

[3] G. Even, I. Y. Spillinger, and L. Stok. Retiming revisited and reversed. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 15(3):348–357, 1996.

[4] A. H. Ghamarian, M. C. W. Geilen, S. Stuijk, T. Basten, B. Theelen, M. Mousavi, A. Moonen, and M. Bekooij. Throughput analysis of synchronous data flow graphs. In *Proc. of the 6th Int. Conf. on Application of Concurrency to System Design*, page 36. IEEE, 2006.

[5] R. Govindarajan and G. R. Gao. Rate-optimal schedule for multi-rate dsp computations. *The Journal of VLSI Signal Processing*, 9(3):211–232, 1995.

[6] E. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *IEEE Global Telecommunications Conf. and Exhibition, GLOBECOM'89. Communications Technology for the 1990s and Beyond.*, pages 1279–1283, 1989.

[7] E. A. Lee. *A coupled hardware and software architecture for programmable digital signal processors*. PhD thesis, University of California, Berkeley, 1986.

[8] E. A. Lee and D. G. Messerschmitt. Static scheduling of

synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, 36(1), 1987.

[9] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[10] 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.*, pages 480–485. IEEE, 2007.

[11] 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*, pages 238–245. IEEE, 2007.

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

[13] 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*, 11(1):41–70, 1997.

[14] T. O'Neil and E. H. M. Sha. Retiming synchronous data-flow graphs to reduce execution time. *IEEE Trans. on Signal Processing*, 49(10):2397–2407, 2001.

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

[16] 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.*, pages 2651–2654. IEEE, 1995.

[17] N. Shenoy. Retiming: theory and practice. *Integration, the VLSI journal*, 22(1-2):1–21, 1997.

[18] N. Shenoy and R. Rudell. Efficient implementation of retiming. In *Proc. of the 1994 IEEE/ACM int. conf. on Computer-Aided Design*, page 233. IEEE, 1994.

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

[20] 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*, pages 276–278. IEEE, 2006. http://www.es.ele.tue.nl/sdf3/.

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

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

[23] V. Zivojnovic and R. Schoenen. On retiming of multirate DSP algorithms. In *Proc. of the Acoustics, Speech, and Signal Processing Conf.*, pages 3310–3313. IEEE, 1996.