Retiming for Synchronous Data Flow Graphs

N. Liveris† C. Lin‡ J. Wang† H. Zhou† P. Banerjee‡
†Northwestern University, Evanston IL, USA
‡University of Illinois, Chicago IL, USA

Abstract
In this paper we present a new algorithm for retiming Synchronous Dataflow (SDF) graphs. The retiming aims at minimizing the cycle length of an SDF. The algorithm is provably optimal and its execution time is improved compared to previous approaches.

1 Introduction
Synchronous Dataflow Graphs are considered a useful way to model DSP applications [1]. This is because in most cases the portions of DSP applications, where most of the execution-time is spent, can be described by processes or actors with constant rates of data consumption and production. Moreover, efficient memory and execution-time minimization algorithms have been developed for SDF graphs [8, 5].

Retiming has been used widely in the past to optimize the cycle time or resources of gate-level graph representations [4, 2, 3]. A lot of work has also been done on extending retiming on SDF graphs [6, 7]. More specifically, retiming has been proposed to facilitate vectorization [9] and to minimize the cycle length of these graphs [6].

Govindarajan et. al. have proposed an algorithm to determine a non-blocking schedule for an SDF graph for maximum throughput [5]. However, there are design cases in which a non-blocking schedule is not feasible. This happens when a part of the application’s behavior is determined dynamically at run-time, or when some of the application’s tasks are sharing resources with higher-priority tasks. These tasks are normally executed on a programmable processor, while the computationally expensive part of the application is run on dedicated resources, has predictable execution time, and is conveniently modeled as an SDF. In case there are data dependencies between the SDF actors and the tasks executed on the programmable processors, a non-blocking schedule may not be feasible. Then a blocking schedule for the SDF is necessary and the blocking schedule with the minimum cycle length will be equivalent to minimum latency of the static part of the application (Figures 1,2).

O’Neil et. al. proposed an algorithm to reduce the clock cycle of a graph below a threshold using retiming [6]. In this paper we propose an optimal algorithm for retiming SDF graphs. The purpose is to minimize the length of the complete cycle of a SDF graph. Two versions of the algorithm will be shown. Both produce better results than any existing algorithm. Moreover, the second one is orders of magnitude faster than O’Neil’s algorithm.

In Sections 2 and 3 we present the basic properties of SDF graphs. An optimal algorithm for minimizing the period of a blocking schedule for an SDF will be described in Section 3. Then in Section 4 the first version of the retiming algorithm will be presented and in Section 5 its correctness will be proven. An improved version of this algorithm will be developed in Section 6. Finally in Sections 8 and 9 the experimental results and conclusions are presented.

All proofs have been omitted due to space limitations. However, they are published in a report, which is available in our website [10].

2 Synchronous Data Flow Graphs
In this section the basic properties of the Synchronous Data Flow graphs will be summarized. For a more detailed description of the SDF properties the user should refer to the literature [1].

An SDF graph is a directed graph \( G = (V, E, p, c, w) \), in which \( d : V \rightarrow \mathbb{R}^+ \) is a function giving the execution delay of a node, and \( p, c, w : E \rightarrow N \) are functions which give the production rate, consumption rate, and initial number of tokens of each edge. Table 1 lists all these symbols with their definitions.

In this paper only live and consistent SDFs will be considered, which can execute without deadlock and with finite memory for an infinite number of times. A necessary condition for a graph to be consistent is

\[
P_0 \equiv (\forall (u,v) \in E : q_u \cdot p(u,v) \equiv q_v \cdot c(u,v))
\] (1)

Throughout this paper we will assume that computing resource constraints for a specific actor are captured by loops, i.e. edges with the same node as head and tail. The number of delays of each loop will determine the number of actor executions that can occur concurrently. The production and consumption rate of the loop will be set to 1.

Moreover, if an actor carries state there is a loop on the node representing the actor with \( p = c = 1 \). The number of delays on the node denote the distance of the dependency in terms of number of executions. As an example for an FIR filter the number of delays will be 1, since each execution of an instance depends on the previous one. Instances of the same SDF node can execute concurrently as long as they do not violate self-dependencies and other constraints imposed by the structure of the graph. The ordering of the produced output tokens and consumed input tokens is taken care by the control mechanism of the edge.

The period for a gate-level graph [4] is defined by the longest path in the graph. In that time all nodes must be executed exactly once. In a consistent SDF graph different nodes can have different average invocation rates. The solution with the minimum positive integers to the state equations gives the number of times each node needs to be executed in a system period or complete cycle of the graph. In a blocking schedule complete cycles of the graph cannot be overlapped. Therefore, the length of the complete cycle can be considered the period of the graph. In this paper we will assume only blocking schedules for SDF graphs.

3 Retiming Properties for SDF Graphs
3.1 Node r Values
In gate-level retiming [4] the \( r(v) \) value of a node \( v \) denoted the number of registers moved from each output edge to the input edges of \( v \).

Retiming in SDF is applied on instance executions of a node \( v \). Each instance execution consumes \( c(u,v) \) tokens from each incoming edge \( (u,v) \) and produces \( p(v,z) \) tokens to each outgoing edge \( (v,z) \). Increasing \( r(v) \) by one is equivalent to “canceling” one execution of one instance of \( v \). Therefore, the outgoing edges will have their weights decreased by \( p(v,z) \) and the incoming edges will have their weights increased by \( c(u,v) \). For \( (u,v) \in E \) the number of delays \( w_r(u,v) \) after each retiming step will be given by

\[
w_r(u,v) = w(u,v) + r(v) \cdot c(u,v) - r(u) \cdot p(u,v)
\] (2)
Since for any valid retiming the final number of delays on each edge must be non-negative:

\[ P1 \equiv (\forall (u,v) \in E : w_t(u,v) \geq 0) \]

must hold for any valid retiming.

It can be easily proven that any retiming solution with integer values satisfying the above properties defines a new graph which belongs to the reachable space of the initial graph [7].

### 3.2 Computing the Max-Length Path

The longest path computation in previous works was done either on the EHG (Equivalent Homogeneous Data-Flow Graph) [6] or the precedence graph [5]. In this paper we will show a way to compute the longest path by using the original SDF graph.

If the repetitions vector of a graph is \( q = [q_1, q_2, ..., q_k] \), then each system iteration (or complete cycle of the graph) will include \( q_v \) executions of SDF node \( v \). We will call these \( q_v \) instance executions of \( v \).

We know that since the edges implement FIFO channels, there exists an implicit partial order for the executions of the instances of \( v \). For each node \( v \) with \( k \in N \) and \( 1 \leq k \leq q_v \):

\[ t(i, k) \leq t(i+1, k) \] (3)

where \( t(i, k) \) is the arrival time at the inputs of the instance \( k \) of node \( v \). In order to find the maximum longest path of one complete cycle of the graph it is enough to find the \( \max_{v \in V} \{ t((v, q_v)) + d(v) \} \).

A recursion equation we can use for this purpose is

\[ t(i, k) = \max_{u \in E} \left( t(u, l) + d(u) \right) \] (4)

where the \( l \) instance of node \( u \) is given by

\[ l = \left\lfloor \frac{k \cdot (u, t(u, v)) - w_s(u, v)}{p(u, v)} \right\rfloor \] (5)

The above equations define an ASAP scheduling. Instance \( k \) of node \( v \) executed immediately after all the necessary tokens are present in the input FIFO channels. The instances, on which \( k \) depends on, are found for each edge incoming to \( v \) by Equation 5. For the \( k \)th instance to be executed \( k \cdot c(u, v) \) tokens must be available on each channel \( (u, v) \in E \). The \( l \)th instance of node \( u \) is the first instance that guarantees that the \( w_s(u,v) \) already present tokens together with the \( l \cdot p(u,v) \) produced in the current complete cycle reach this number.

In Equation 5, \( l \) can be less than or equal to zero. This means that the \( k \)th instance of \( v \) node depends on the \( q_v + l \) instance of the previous complete cycle. We will define \( \forall v \in V : l \leq q_v \cdot t(u, l) + d(u) = 0 \). This property makes the scheduling difficult. As instance \( k \) cannot start execution before time 0, when the complete cycle begins, this property prevents complete cycles from overlapping.

Equation 4 can be made weaker by replacing equality. Then the following property needs to hold

\[ P2 \equiv (\forall v \in V : \forall (u,v) \in E : \forall k \in N : \]

\[ (1 \leq k \leq q_v) \Rightarrow (l = \left\lfloor \frac{k \cdot (u, t(u, v)) - w_s(u, v)}{p(u, v)} \right\rfloor \land t(i, k) \geq t(u, l) + d(u)) \]

The blocking schedule property is equivalent to

\[ P3 \equiv (\forall v \in V : \forall k \in N : (k < 1) \Rightarrow (t(i, k) = d(v))) \]

\( P2 \) and \( P3 \) are more general and hold for any valid blocking scheduling instead of an ASAP blocking scheduling of the instance nodes.

### 3.3 Optimal Period

For the period \( T \) of a blocking schedule of an SDF graph, it must hold

\[ (\forall v : \forall k \leq q_v : t(i, k) + d(v) \leq T) \]

Because of Relation 3 the following property is necessary and sufficient

\[ P4 \equiv (\forall v : t(i, q_v) + d(v) \leq T) \]

for \( T \) to be the period of the schedule. The period can be considered as a function of the retiming vector \( r = [r_1, r_2, ..., r_k] \) of \( T \). For the optimal period \( T(r) \) of a blocking schedule the following property holds

\[ P5 \equiv (\forall r : T(r) \leq T(r')) \]

### 3.4 Problem Formulation

Given a consistent SDF graph \( G = (V, E, d, p, c, w) \) find a retiming \( r \) and minimum complete cycle length \( T(r) \) that satisfies properties \( P1 \)-\( P5 \).

### 4 Retiming Algorithm

In this section the retiming algorithm will be described. The algorithm can be seen in Figures 3.4, and 5.

The algorithm uses procedures \( \text{get}\_\text{u} \) and \( \text{init}\_\text{u} \) to find the arrival times. From \( P4 \) we note that only the last \( (q_v) \)th instance of each node is important to find the period of the complete cycle. Therefore, it is sufficient to compute \( \forall v \in V : \forall (u, v) \) and the arrival times of their dependencies. Procedure \( \text{get}\_\text{u} \) achieves that by recursively calling itself on the dependencies of an instance node. Therefore, the procedure will avoid computing the arrival nodes of instance nodes that cannot change the arrival time of \( (v, q_v) \) for any \( v \in V \).

Moreover, the procedure avoids recomputation of the arrival times of the same instance nodes by maintaining an array \( W \). This array holds the arrival times of the nodes already computed. Initially, the entries of this array are set to -1 (could be any illegal number) by procedure \( \text{init}\_\text{u} \). Any computed arrival time is stored in the array. Arrival times are only computed if the value in the array is -1, or else the already computed value is returned.

Property \( P3 \) is preserved by the first two lines of procedure \( \text{get}\_\text{u} \). By implementing \( \text{get}\_\text{u} \) as a memory function working directly on the SDF, the expensive construction of an EHG or a precedence graph is avoided.

Furthermore, restricting the computation of the arrival times to only those instances that can affect the period has an effect on the properties discussed above. More specifically it is equivalent to re-laxing \( P2 \) to be valid only for the \( q_v \)th instance of each node and its dependencies. It is easy to show that for any result using these arrival times we can obtain arrival times for all node instances that validate \( P2 \) using an ASAP algorithm. For efficiency reasons, however, the algorithm will not compute the arrival times for all nodes in each iteration. Predicate \( P2 \) can be replaced by a weaker predicate \( P2' \). \( P2' \) will be true, whenever for the arrival times obtained there exists an algorithm 5 to compute the rest of the node instance arrival times, such that \( P2 \) can be validated

\[ P2' \equiv (\exists r : P2) \]

With the arrival times obtained by \( \text{get}\_\text{u} \), predicate \( P2' \) is true.

The algorithm in Figure 5 starts by initializing the memory function elements for all arrival times to -1. Then it sets \( w_{t, t}(v) = 0 \) and computes the arrival times for all \( (v, q_v) \). After finding the \( \max = \max(t((v, q_v)) + d(v)) \), it sets \( t_{\max} = \max(t) \) and enters the while loop. In each iteration of the while loop, \( r(v) \) is increased by 1, where \( v \) is
the node for which \( \text{max}_t = t(v_n, q_n) + d(l_n) \) in the previous iteration. If \( \text{max}_t < T_{\text{step}} \), then \( T_{\text{step}} \) becomes equal to \( \text{max}_t \) and the algorithm tries to find another \( r \) with \( T(r) < T_{\text{step}} \).

Each time an r-value changes the algorithm recomputes the arrival times using the memory function. This way after each \( r \) change the algorithm keeps predicates \( P^2 \) and \( P^3 \) invariant. \( P^4 \) is always satisfied by \( \text{max}_t \) and the current iteration's \( r \). Therefore, it is satisfied by \( (r^0, T_{\text{step}}) \) when the algorithm exits.

In order, to understand the reason \( P^1 \) is kept invariant as well, we have to refer to Equation 5. An edge \((v_n, q)\) can have \( w_f(v_n, q) < 0 \) if before the change of \( r(v_n) \) to \( r(v_n) + 1 \), there were \( w_s(v_n, q) < p(v_n, q) \) tokens. But in that case

\[
\begin{align*}
    l_n &= \left\lfloor \frac{q \cdot (v_n, q) - w_s(v_n, q)}{p(v_n, q)} \right\rfloor, \\
     l_n &= \left\lfloor \frac{q \cdot (v_n, q) - w_s(v_n, q)}{p(v_n, q)} \right\rfloor - q_n. \\
\end{align*}
\]

But that means that \((v_n, q)\) instance can only start after \((v_n, q_n)\) has completed execution and, therefore,

\[
t(v_n, q) + d(v) > t(v_n, q_n) + d(v_n) = \text{max}_t
\]

which is a contradiction. \( P^3 \) is also an invariant of the algorithm. Only property \( P^5 \) may not be true after initialization and will become true upon termination of the while loop algorithm, as proven in the next section.

```plaintext
proc get_t(v, k)
    if (k < 1) then
        return -d(v);
    fi;
    if (t[v][k] < -1) then
        return -1;
    fi;
    max_t ← -1;
    for each \((u, v) \in E\) do
        t ← \([l(v)]_{u \to v} + d(u)\);
        if (max_t ≤ l_t) then
            max_t ← l_t;
        fi;
    endfor;
    return t[v][k];
endproc
```

Figure 3: Procedure for initializing the arrival times.

Figure 4: Pseudocode of getting the arrival time.

5 Algorithm Correctness

In this section the correctness of the algorithm will be proven. Our analysis will be restricted to strongly connected graphs. In Section 7 it will be shown how to extend the approach to graphs with input/output channels, sources and sinks.

5.1 Analysis

In this section we will analyze the properties of strongly connected SDF graphs. By using the ordered pair \((v, l)\) we will denote a node \( v \) labeled with the instance number \( l \), for which \( 1 \leq l \leq q_v \).

**Definition** A dependence walk

\[ W = (v_0, l_0) \rightarrow (v_1, l_1) \rightarrow \ldots \rightarrow (v_n, l_n) \]

is a walk in the SDF graph \( G \) in which the execution of \((v_i, l_i)\) can only start after the execution of \((v_{i-1}, l_{i-1})\) has been completed for all \( i, 0 \leq i < n \).

**Algorithm SDF Retiming**

**Input:** An SDF graph \( G = (V, E, d, p, r, w) \).

**Output:** A pair \((r, T_{\text{min}})\) which represents an optimal retiming \( r \) satisfying minimum complete cycle execution time \( T_{\text{min}} \).

\[
\begin{align*}
    \text{max}_t &← 0; \\
    \text{init}_r(); \\
    \text{for each } v \in V \text{ do} \\
        r(v) &← 0; r(v, 0) ← \text{get}_r(v, q_v); \\
        \text{if } t(v, q_v) + d(v) > \text{max}_t \text{ then} \\
            \text{max}_t &← t(v, q_v) + d(v); t(v) ← v; \\
        \text{fi}; \\
    \text{endfor}; \\
    T_{\text{step}} &← \text{max}_t; \\
    \text{while } (\exists v : r(v) < q_v \text{ and } \forall v : r(v) > 2 \cdot q_v) \text{ do} \\
        r(v) &← r(v) + 1; \\
        \text{init}_r(); \\
    \text{for each } v \in V \text{ do} \\
        r(v, q_v) &← \text{get}_r(v, q_v); \\
        \text{if } t(v, q_v) + d(v) > \text{max}_t \text{ then} \\
            \text{max}_t &← t(v, q_v) + d(v); t(v) ← v; \\
        \text{fi}; \\
    \text{endfor}; \\
    \text{if } \text{max}_t < T_{\text{step}} \text{ then} \\
        r^* &← r; \\
        T_{\text{step}} &← \text{max}_t; \\
        \text{fi}; \\
    \text{endwhile}; \\
    \text{return } (r^*, T_{\text{step}}); \\
\end{align*}
\]

Figure 5: Pseudocode of retiming algorithm.

Figure 6: An example of a dependency walk.

From Equation 5, if \( l_{i-1} = \left\lceil \frac{l(v_{i-1}, l_{i-1}) - w_s(v_{i-1}, l_{i-1})}{d(v_{i-1})} \right\rceil \) with \( 1 \leq l_{i-1} \leq q_{v_{i-1}} \) and \((v_{i-1}, v_i) \in E\), then there is a dependency relation between node instances \((v_{i-1}, l_{i-1})\) and \((v_i, l_i)\).

For each \((v_i, l_i)\) it holds that \( t(v_i, l_i) \geq t(v_{i-1}, l_{i-1}) + d(v_{i-1}) \).

Also note that in \( W^* \) there can be multiple appearances of the same SDF node with a different label each time (Figure 6). That means that there could be \( v_1, v_j \) with \( i \neq j \) and \( v_i = v_j \). Moreover, in \( W^* \) an SDF edge may be used multiple times to define a dependency. From now the term walk will denote a dependence walk in the SDF graph.

**Definition** A critical walk is a walk for which

\[ (v_i : 1 \leq i \leq n) \Rightarrow (t(v_i, l_i) = t(v_{i-1}, l_{i-1}) + d(v_{i-1})) \]

and \( t(v_0, l_0) = 0 \).

For a critical walk the first node starts exactly at time 0, which is the beginning of the complete cycle. All other nodes start exactly at the time their predecessor in the walk has completed execution.

**Lemma** Suppose \( W = (v_0, 0) \rightarrow \ldots \rightarrow (v_n, l_n) \) is a critical walk and \( r(v_0, q_0) + d(v_0) = T(r) \) for a retiming vector \( r \), then for any retiming vector \( r' \) for which a dependence walk \( W' = (v_0', 0) \rightarrow \ldots \rightarrow (v_{n}', l_{n}') \) exists, it will hold \( T(r') \geq T(r) \).
Algorithm SDF Retiming: Improved
Input: An SDF graph $G=(V,E,p_i,p_f,r,w)$.
Output: A pair $(r,T_{\text{min}})$ which represents an optimal retiming $r$ satisfying minimum complete cycle execution time $T_{\text{min}}$.
1. max $\gets 0$; $Q1 \gets \emptyset$; $Q2 \gets \emptyset$;
2. init (;);
3. for each $v$ in $V$ do
   4. $r(v) \gets 0$; $t(v,q_t) \gets \text{get}(v,q_t)$;
   5. if $t(v,q_t)+d(v) > \text{max}$ then
      6. max $\gets t(v,q_t)+d(v)$; $v_0 \gets v$;
   7. fi;
   8. endfor;
9. $T_{\text{step}} \gets$ max;
10. while ($Q1 \neq \emptyset$) do
11. $v \gets \text{dequeue}(Q1)$;
12. $r(v) \gets r(v)+1$;
13. foreach $(v_0,v) \in E$ do
14. if $(w(v_0,v),u) < 0$ then
15. $Q2 \gets (v_0,v)$;
16. fi;
17. endfor;
18. endwhile;
19. while ($Q2 \neq \emptyset$) do
20. $(v_0,v) \gets \text{dequeue}(Q2)$;
21. $\Delta r(v) \gets -[\frac{\Delta}(v)-\frac{\Delta r(v)}{2}]$;
22. if $\Delta r(v) > 0$ then
23. $r(v) \gets \Delta r(v)+r(v)$;
24. foreach $(v_0,v_2) \in E$ do
25. if $(w(v_0,v_2),u) < 0$ then
26. $Q2 \gets (v_0,v_2)$;
27. fi;
28. endfor;
29. endwhile;
30. endwhile;
31. fi;
32. endwhile;
33. fi;
34. for each $v$ in $V$ do
35. $t(v,q_t) \gets \text{get}(v,q_t)$;
36. if $t(v,q_t)+d(v) > \text{max}$ then
37. max $\gets t(v,q_t)+d(v)$; $v_0 \gets v$;
38. fi;
39. if $t(v,q_t)+d(v) \geq T_{\text{step}}$ then
40. $Q1\gets \text{enqueue}(v)$;
41. fi;
42. endfor;
43. if max $< T_{\text{step}}$ then
44. $r^* \gets r$;
45. $T_{\text{step}} \gets$ max;
46. $Q1\gets \text{enqueue}(v)$;
47. fi;
48. endwhile;
49. Return $(r^*,T_{\text{step}})$;

Figure 8: Pseudocode of the improved retiming algorithm.

Lemma 2 Suppose $W = (v_3,l_0) \rightarrow \ldots \rightarrow (v_{n-1},l_n)$ is a dependency walk. Then by increasing the r value of any node u with $u \notin W$, $W = (v_0,l_0) \rightarrow \ldots \rightarrow (v_{n-1},l_n)$ remains a dependency walk in the graph. □

Lemma 3 Suppose $W = (v_3,l_0) \rightarrow \ldots \rightarrow (v_{n-1},l_n)$ is a dependency walk and the r-value of node $u$, with $u \notin W$ but $u \neq v_{i-1}$, is increased by $\Delta r_u$. Then another dependency walk is obtained $W' = (v_3,l_0') \rightarrow \ldots \rightarrow (v_{n-1},l_n')$ with
$$l'_i = \begin{cases} l + \Delta r_u & : v_i = u \\ l_i & : v_i \neq u \end{cases}$$

Theorem 1 Suppose for a retiming $r$ that $r(v_a,q_a)+d(v_a) = T(r)$. If $\exists r'$ such that $T(r') < T(r)$ and $\forall r', r'(v) \geq r(v)$, then $r'(v_a) > r(v_a)$ □

Lemma 4 If $r$ is a retiming solution such that $\forall v \in V_r(v_q_r) \leq T(r)$, then $r' = [r_1+k \cdot q_1, r_2+k \cdot q_2, \ldots, r_{|l|}+k \cdot q_{|l|}] \forall k \in Z$, is also a solution with $T(r') = T(r)$.

Therefore, if one retiming solution exists for $T$ then infinite solutions exist. However, from the following equation
$$r' = [r_1+k \cdot q_1, r_2+k \cdot q_2, \ldots, r_{|l|}+k \cdot q_{|l|}]$$

it can be shown that $\exists k$ such that $\forall v \in V _r(v_q) \geq 0$ and $\forall u, r(u) < q_u$. The retiming solutions for the minimum $T_{\text{min}}$ will be called optimal solutions. The solutions with $r(v) \geq 0, \forall v$ and at least one $u$ such that $r(u) < q_u$ and $T(r) = T_{\text{min}}$ will be called the basic optimal solutions. It can be proven that the algorithm will always produce a basic optimal solution.

5.2 First Termination Condition

Lemma 5 After initialization and at each iteration of the algorithm of Figure 5, if $(\exists r: T(r) < T_{\text{step}})$, then the following property holds $(\exists r: r(u) < q_u)$.

Lemma 5 also specifies a property on the existence of $T_{\text{min}}$. In each iteration of the algorithm $T_{\text{step}}$, which is the minimum period found so far, is kept constant and $T(r)$ is the target for reduction until $T(r) < T_{\text{step}}$. Based on Lemma 5, if

$$(\exists r: T(r) < T_{\text{step}}) \Leftrightarrow (\exists r: (\forall v: r(v) < q_v))$$

The above property makes $(\forall v: r(v) \geq q_v)$ a termination condition for the algorithm. If it is true $T_{\text{min}} = T_{\text{step}}$ and the r-vector $(r : T(r) = T_{\text{step}})$ as found in the previous iterations of the algorithm is one basic optimal solution.

5.3 Second Termination Condition

Lemma 6 If $(\exists r: r(v) \geq q_v)$ the algorithm exits with one basic optimal condition. □

In the section below the second termination condition will be discussed.

5.4 Algorithm's Complexity

From the second termination condition a bound can be derived for the number of iterations of the while loop. The sum of the r values can be
$$\sum_{v \in V} r(v) \leq \sum_{v \in E[v]} (2 \cdot |V| \cdot q_v + (q_v - 1) + 1)$$

In the worst case only node $u$ will have $r(u) < q_v$ keeping the first termination condition falsified. The r values of the rest of the nodes form the first term and 1 more r value increase is needed to terminate the algorithm.

If $q_{ave} = \frac{1}{|V|} \sum_{v \in V} q_v$, we represent the average q value over all nodes then the sum is upper bounded by
$$\sum_{v \in V} r(v) \leq 2 \cdot |V|^2 \cdot q_{ave}$$

483
Since in each iteration of the while loop the sum on the left side will change by 1, the number of iterations is bounded by $2 \cdot |V| \cdot \Delta_{\text{dave}}$.

In each iteration the necessary arrival times are computed. In the worst case the arrival computation will take

$$\sum_{(v,s) \in E} q_v = |E| \cdot |V| \cdot \Delta_{\text{dave}}.$$ 

Therefore, the total worst case complexity will be $O(|V| \cdot |E| \cdot \Delta_{\text{dave}})$.

6 Improved Version of the Retiming Algorithm

The running time of the algorithm can be improved if we relax P1 not to be valid after each step of the algorithm, but be valid upon termination. That will allow the algorithm to do multiple r value changes without having to find the arrival times of the node instances.

Moreover, two more conclusions can be drawn from the previous section. Firstly, from Theorem 1 we observe that the order in which we change the r values, while approaching a basic optimal solution, is not important. If there exists a critical walk in the graph and for the last node $v_s$ of the walk $T_{\text{step}} \leq t(v_s q_v) + d(v_s)$, then for any $r'$ for which $T(r') < T_{\text{step}}$, the r value of v will be $r(v) > r(v)$.

Secondly, from Lemmas 1-3 we see that by increasing the r value of a node for which $t(v_s q_v) + d(v_s) \geq T_{\text{step}}$ cannot improve the arrival time of nodes $v_m$ for which $T_{\text{step}} < t(v_s q_v) + d(v_m)$ Therefore, if before the r value change, $t(v_s q_v) + d(v_s) = T_{\text{step}}$ was valid, after the change $t(v_s q_v) + d(v_s) \geq T_{\text{step}}$ remains valid.

Using these conclusions, the algorithm can be modified to store all nodes, which have $t(v_s q_v) + d(v_m) \geq T_{\text{step}}$, each time the arrival times are computed. Then modify their r values and then compute the arrival times again. That way, it is not guaranteed that P1 will remain invariant. Therefore, after each change all edges, which have their weight reduced, will be checked for P1. If P1 does not hold the necessary r change will be done to validate P1. The change is correct, as long as it is minimum, because in the basic optimal solution P1 must hold for all edges.

The necessary change to make the number of delays of an edge positive is

$$w(u,v) + r(v) - c(u,v) + \Delta r(v) \cdot c(u,v) - r(u) \cdot p(u,v) \geq 0$$

$$\Rightarrow \Delta r(v) \geq \frac{r(u) \cdot p(u,v) - w(u,v)}{c(u,v)} - r(v)$$

Since $r(v)$ is less than or equal to $r'(v)$, $r'(v)$ must be greater than or equal to $r(v) + \Delta r(v)$, otherwise condition P1 will not hold for the basic optimal solution, which is a contradiction. In the algorithm of Figure 2 two queues are maintained. The first queue (Q1) holds the nodes for which it is known that their values must be increased for $T_{\text{step}}$ to be reduced. The while loop with condition Q1 $\neq \emptyset$ increases the value of each of these nodes. The queue does not contain double entries, since when filled each node is checked only once (done by the for-loops of the algorithm).

The second queue (Q2) stores the edges for which P1 has been invalidated. For those edges, the r value of the head node is increased to restore the validity of P1, if needed. Note that although Q2 does not contain double entries, the head node of two or more edges may be the same in some cases. Therefore, before restoring P1, it is necessary to check how large the increase of $\Delta r(u)$ should be. The check for $\Delta r(u) > 0$ in the while loop with condition Q2 $\neq \emptyset$ does exactly this.

At the end of each iteration the r values of all nodes in Q1 have been increased, and P1 has been validated for all edges, before the computation of the arrival times starts again, which generates new entries in Q1. In the case $\text{max} \leq T_{\text{step}}$, Q1’s unique entry is the node v for which $t(v_s q_v) + d(v_s) = \text{max}$. Otherwise, all nodes for which $t(v_s q_v) + d(v_s) \geq T_{\text{step}}$ enter the queue.

Both theorems for the termination condition are still valid.

The worst-case complexity of the algorithm remains the same. However, its practical efficiency is improved, as verified by the experimental results presented in Section 8.

7 Source, Sink Nodes - Input Output Channels

The analysis presented in this paper is based on strongly connected graphs. If a graph has source and sink nodes, then it can be easily transformed to a strongly connected graph by introducing a new node I with $q_I = 1$ and $d(I) = 0$. Then for each source $s$ of the graph an edge $(I, s)$ will be included in E with $c(I, s) = 1$, $p(I, s) = q_s$, and $w(I, s) = 0$. Moreover, for each sink t an edge $(t, I)$ will be included in E with $p(t, I) = 1$ and $c(t, I) = q_t$. The number of weights on these edges can be considered as a very large number W. It is easy to prove that P0 is still valid after this transformation and the graph is consistent.

As shown in Figure 1, there are SDF graphs which include input and output channels. These channels model the system’s communication with its environment. Input and output channels are represented by edges, whose tail and head node, respectively, are missing from the graph. The head and tail of these edges are nodes that do not belong to the system under consideration, as opposed to sources and sinks of the graphs. We will assume that if r is an input channel incident to node v, all $q_v \cdot c(e)$ tokens needed by v for the current complete cycle are available at time 0.

For retiming graphs with input/output channels two nodes will be added I and O. All output edges will be connected to O and all input edges will become incident from I. The two nodes I and O will be connected with an edge $(O, I)$ with $p(O, I) = c(O, I) = w(O, I) = 1$. Moreover, $q_I = q_O = 1$ for the new nodes. Each output edge e incident from node v will be replaced by $(v, O)$ with $p(v, O) = c(v, O) = c(e) \cdot q_v$, and $w(v, O) = w(e)$. In a similar way, every input edge e incident to node v will be replaced by $(I, v)$ with $c(I, v) = c(e), p(I, v) = p(e) \cdot q_v$, and $w(I, v) = w(e)$. The delays of the two nodes with be $d(I) = d(O) = 0$.

These modifications on the graph have two important implications. Firstly, since $d(I) = d(O) = 0$ the assumption that for each $v \in D(v) > 0$ does not hold anymore. This assumption was used to prove that P1 is an invariant (Inequality 7). After the addition of the new nodes the correctness of the first algorithm cannot be proven anymore. This is not a problem though for the improved version, since P1 is relaxed and validated again by using the Q2 queue. Secondly, the newly added edge $(O, I)$ represents the dependence of the inputs of the next cycle on the outputs of the current cycle (Figure 1). Initially, $w(O, I) = 1$ and the weight of this edge should not become 0, since that would mean that the inputs for a complete cycle can be produced instantly during the complete cycle, which is not a correct model of the environment of the system. In this case, the improved version of the algorithm can make $P6 \equiv (w(O, I) \geq 1)$ hold upon termination, the same way as it ensures P1. Edge $(O, I)$ is entered in Q2 after an r change if $w(O, I) < 1$ and it is adjusted accordingly during the execution of the loop that empties Q2. The way this type of constraints can be handled by O’Neil’s algorithm [6] is not known.
8.1 Experimental Setup

The graphs were obtained from the ISCAS89 benchmarks. For the delay a random integer was assigned between 1 and 30. The weight of each node was also selected randomly between 1 and the value \( q_{\text{max}} \). For each graph, the weights of each edge were chosen in such a way, so that the graph would be consistent. More specifically, for each edge for specific \( q_0 \), the values of the edges were chosen randomly between 1 and the value \( q_{\text{max}} \). Three values (4, 16, 32) have been used for \( q_{\text{max}} \) to observe how the performance of the algorithms scales with this parameter. After the \( q \) value of each node was assigned, the \( q \) and \( c \) values of every edge were chosen in such a way, so that the graph would be consistent. More specifically, \( p(u,v) = \frac{q_0}{\gcd(a + p)} \) and \( c(u,v) = \frac{q_0}{\gcd(a + p)} \).

This method creates the minimum consumption and production rates for each edge for specific \( q \) values of the graph.

Two additional nodes \( I \) and \( O \) were included with \( q_I = q_O = 1 \). It was connected to all primary inputs of the graph and \( O \) to all primary outputs. Edge \((O,I)\) was included with \( w(O,I) = 1 \). For graphs with no zero delay nodes \( d(O) \) and \( d(I) \) were chosen randomly as integers from \([1,30]\). These values were used in the first set of experiments. In the second set \( d(O) = d(I) = 0 \).

Initially, non-zero weights were assigned to 50% of the total edges in the graph. The value of an edge weight was a random integer in \([1,q_{\text{max}}]\). Then the graph was checked for liveness and if a deadlock was detected, the weights of each input channel \((u,v)\) of a node that could not execute were increased by \( c(u,v) \). This process was repeated until the graph was live.

O’Neil’s algorithm applies retiming to reduce the cycle length below a constraint given as an input. If the algorithm is used to find the minimum cycle length a linear search must be performed on the possible cycle length values, which are integers. Binary search cannot be performed, since it is not guaranteed that if the algorithm returns a retiming for cycle length \( T_i \), it will not return false for cycle length \( T > T_i \). We implemented O’Neil’s algorithm to compare it with the two new algorithms.

8.2 Strongly Connected Graphs with No Zero-Delay Nodes

On strongly connected graphs with no zero delay nodes all three algorithms are applicable. Tables 2, 3, and 4 summarize the results in terms of running time and period. The first and improved algorithm always produce the same period \( T \), since both of them find the optimal solution for a specific graph. The period found by these two algorithms is in all cases at least as good as the period found by O’Neil’s algorithm. The difference depends on the the randomly generated graph. In some cases it is 0 and in other cases it can be more than 20%. The execution time of the improved version is much faster than the other two algorithms, especially for larger graphs. As \( q_{\text{ave}} \) grows the running time of the three algorithms increases. However, the impact of that parameter is more significant for the running time of O’Neil’s algorithm. The reason is that the size of the EHG and the complexity of the algorithms working on it depend on \( q_{\text{ave}} \). For large graphs the EHG is larger, so the complexity of the algorithms is higher.

8.3 Strongly Connected Graphs with Additional Constraints

In this section the performance of the improved retiming algorithm will be shown for strongly connected graphs with the additional constraint that \( r(I) > r(O) \), which represents the most realistic scenario for the purpose of minimizing the cycle length of SDF graphs.

Table 5 shows the execution time and resulting cycle length for the improved algorithm for graphs generated with \( q_{\text{max}} = 32 \). The other two algorithms cannot be applied on these graphs. For both of them it is not known how they can handle constraints like \( P(6) \). Moreover, the correctness of the first retiming algorithm does not hold when zero delay nodes are present.

9 Conclusions

In this paper two optimal algorithms were presented for minimum cycle length retiming of SDF graphs. Both work on strongly connected graphs. The improved version can be applied on graphs with input and output channels, is faster, and can handle additional constraints. The experimental results show that the improved version is orders of magnitude faster than existing approaches [6] and produces better results.

References