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.
Retiming for Synchronous Data Flow Graphs

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 (Figure 1).

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 2, 3).

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.

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].
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

\[
w_r(u, v) = w(u, v) + r(v) \cdot c(u, v) - r(u) \cdot p(u, v)
\]

Since for any valid retiming the final number of delays on each edge must be non-negative

\[P_1 \equiv (\forall(v, u) \in E : w_r(u, v) \geq 0)\]

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 \( q = [q_1, q_2, \ldots, q_p] \), 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 \( 1 \leq k \leq q_v \):

\[t(v, k - 1) \leq t(v, k)\]

(3)

where \( t(v, k) \) 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 maximum \( q_v \cdot t(v, q_v) + (v) \) of all delays.

A recursion equation we can use for this purpose is

\[t(v, k) = \max_{\forall(u, v) \in E} (t(u, l) + (v))\]

(4)

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

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

(5)

The above equations define an ASAP scheduling. Instance \( k \) of node \( v \) executes 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 have been available on each channel \((u, v) \in E \). The \( k \)th instance of \( v \) node is the first instance that guarantees that the \( w_r(u, v) \) 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 \( q_v + 1 \) instance of the previous complete cycle. We will define \( \forall u \in V, l \leq 0, t(u, l) + (u) = 0 \). This property makes the scheduling blocking. As instance \( k \) cannot start execution before time 0, when the current 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

\[P_2 \equiv (\forall v \in V, \forall [u, v] \in E, \forall k \in N : t(v, k) \leq t(v, k + 1) - (v))\]

Table 1: Definition of Commonly Used Parameters

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_v )</td>
<td>number of executions (instances) of node ( v ) in one complete cycle</td>
</tr>
<tr>
<td>( d(v) )</td>
<td>execution time for each instance of node ( v )</td>
</tr>
<tr>
<td>( r(v) )</td>
<td>number of tokens produced on edge ([u, v] ) as a result of an execution of node ( u )</td>
</tr>
<tr>
<td>( c(u, v) )</td>
<td>number of tokens of edge ([u, v] ) consumed as a result of an execution of node ( v )</td>
</tr>
<tr>
<td>( w_r(u, v) )</td>
<td>number of initial tokens (delays) on edge ((u, v) ) in the input graph</td>
</tr>
<tr>
<td>( r(v) )</td>
<td>retiming value for node ( v )</td>
</tr>
<tr>
<td>( r )</td>
<td>vector ( l \times</td>
</tr>
<tr>
<td>( w_r(u, v) )</td>
<td>number of delays on edge ((u, v) ) after ( r ) has been applied to the graph</td>
</tr>
<tr>
<td>( t(v, k) )</td>
<td>arrival time for the instance ( k ) of node ( v ), the time when the tokens for the ( k )th instance are available ( \forall [u, v] \in E )</td>
</tr>
<tr>
<td>( T )</td>
<td>latency of a complete cycle of the SDF graph, equals the period of a blocking schedule</td>
</tr>
</tbody>
</table>

The blocking schedule property is equivalent to

\[P_3 \equiv (\forall v \in V, \forall k \in N : (k < 1) \Rightarrow (t(v, k) = -(v)))\]

(6)

The blocking schedule property is equivalent to

\[P_3 \equiv (\forall v \in V, \forall k \in N : (k < 1) \Rightarrow (t(v, k) = -(v)))\]

(6)

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

\[(\forall k \in N : (1 \leq k \leq q_v : t(v, k) + (v) \leq T)\]

Because of Relation 3 the following property is necessary and sufficient

\[P_4 \equiv (\forall v : t(v, q_v) + (v) \leq T)\]

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

\[P_5 \equiv (\forall r^\prime : T(r) \leq T(r^\prime))\]

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 4.5, and 6.

The algorithm uses procedures \( get_r \) and \( init_r \) to find the arrival times. From \( P_4 \) we note that only the last \( q(v) \) instance of each node is important to find the period of the complete cycle. Therefore, it is sufficient to compute \( \forall v \in V, t(v, q_v) \) and the arrival times of their dependencies. Procedure \( get_r \) 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 \( v \). This property makes the scheduling blocking. As instance \( k \) cannot start execution before time 0, when the current 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

\[P_2 \equiv (\forall v \in V, \forall [u, v] \in E, \forall k \in N : t(v, k) \leq t(v, k + 1) - (v))\]

By implementing \( get_r \) 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 relaxation P2 to be valid only for the qth 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 S to compute the rest of the node instance arrival times, such that P2 can be validated

\[ P2' = (S : P2) \]

With the arrival times obtained by get\_\_l\_\_pred P2' is true.

The algorithm in Figure 6 starts by initializing the memory function elements for all arrival times to -1. Then it sets \( V; r(v) = 0 \) and computes the arrival times for all \( (v, q_v) \). After finding the \( \text{max} = \max(r(v, q_v) + d(v)) \), it sets \( T_{\text{step}} = \text{max} \) and enters the while loop. Each iteration of the while loop, \( r(v) \) is increased by 1, where \( v_n \) is the node for which \( \text{max} = r(v_n, q_v) + d(v_n) \) in the previous iteration. If \( \text{max} < T_{\text{step}} \), then \( T_{\text{step}} \) becomes equal to \( \text{max} \) 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 change the algorithm keeps predicates P2' and P3 invariant. P4 is always satisfied by \( \text{max} \) 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 P1 is kept invariant as well, we have to refer to Equation 5. An edge \( (v_n, z) \) can have \( w_f(v_n, z) < 0 \) if before the change of \( r(v_n) \) to \( r(v_n) + 1 \), there were \( w_i(v_n, z) < tr(v_n, z) \) tokens. But in that case

\[
\begin{align*}
\lambda_{v_n} & = \left[ \frac{q_v \cdot [v_n, z] - w_i(v_n, z) - w_f(v_n, z)}{p(v_n, z)} \right] \geq \left[ \frac{q_v \cdot [v_n, z] - w_f(v_n, z)}{p(v_n, z)} \right] \\
\lambda_{v_n} & = q_v \\
\end{align*}
\]

But that means that \((z, q_v)\) instance can only start after \((v_n, q_{v_n})\) has completed execution and, therefore,

\[
t(z, q_v) + d(z) > t(v_n, q_v) \geq t(v_n, q_{v_n}) + d(v_n) = \text{max}
\]

which is a contradiction. P1 is also an invariant of the algorithm. Only property P5 may not be true after initialization and will become true upon termination of the while loop algorithm, as proven in the next section.

![Figure 4: Procedure for initializing the arrival times.](image)

**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

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

![Figure 7: An example of a dependency walk](image)
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 $i, 0 \leq i < n$.

From Equation 5, if $l_{i-1} = \left\lfloor \frac{t(v_{i-1}, l_{i-1}) - t(v_i, l_i)}{p(v_{i-1}, v_i)} \right\rfloor$ with $1 \leq l_{i-1} \leq q_{v_{i-1}}$, and $(v_{i-1}, v_i) \in E$, then there is a dependence 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 7). That means that there could be $v_i, v_j$ 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

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

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 1** Suppose $W = (v_0, l_0) \rightarrow \ldots \rightarrow (v_n, l_n)$ is a critical walk and $t(v_0, q_0) + d(v_0) = T(r)$ for a retiming vector $r$, then for any retiming vector $r'$ for which a dependence walk $W' = (v_0, l'_0) \rightarrow \ldots \rightarrow (v_n, l'_n)$ exists, it will hold $T(r') \geq T(r)$.

**Proof** Since $W$ is still a valid walk in the graph, each edge in it specifies a dependence. Therefore, the following property holds

$$(\forall i : (0 \leq i < n) \Rightarrow (t'(v_{i+1}, l_{i+1}) \geq t'(v_i, l_i) + d(v_i)))$$

for the arrival times.

For the first instance of the node walk $t'(v_0, l'_0) \geq 0 = t(v_0, l_0)$. If for $k, t(v_k, l_k) \geq t(v_k, l_k)$ holds, then for $k + 1$

$t'(v_{k+1}, l_{k+1}) \geq t'(v_k, l_k) + d(v_k) \geq t(v_k, l_k) + d(v_k) = t(v_{k+1}, l_{k+1})$

follows. So, by induction $t'(v_n, l_n) \geq t(v_n, l_n)$, which implies $T(r') \geq T(r)$. □

**Lemma 2** Suppose $W = (v_0, l_0) \rightarrow \ldots \rightarrow (v_n, l_n)$ is a dependence walk. Then by increasing the $r$-value of any node $u$ with $u \notin W$, $W' = (v_0, l_0) \rightarrow \ldots \rightarrow (v_n, l_n)$ remains a dependence walk in the graph.

**Proof** If $r(u)$ changes value, the number of weights only on edges, which are incoming or outgoing to $u$, will change. Since $(\forall (v_i, v_{i+1}) \in W' : u \neq v_i \land u \neq v_{i+1})$ holds, none of the edges of $W'$ will have their $w(v_i, v_{i+1})$ modified. Therefore, from Equation 5, $W' = (v_0, l_0) \rightarrow \ldots \rightarrow (v_n, l_n)$ is still a dependence walk in the graph. □

**Lemma 3** Suppose $W = (v_0, l_0) \rightarrow \ldots \rightarrow (v_n, l_n)$ is a dependency walk and the $r$-value of node $u$, with $u \in W$ but $u \neq v_n$, is increased by $\Delta r_u$. Then another dependency walk is obtained $W' = (v_0, l'_0) \rightarrow \ldots \rightarrow (v_n, l'_n)$ with

$$l'_i = \begin{cases} l_i + \Delta r_u & : v_i = u \\ l_i & : v_i \neq u \end{cases}$$

(8)

**Proof** We are going to start from node instance $(v_n, l_n)$ and walk backwards by induction to prove that $W'$ exists. For $v_n$ the above relation holds, since $v_n \neq u$ and $l'_n = l_n$ by the way instance node $(v_n, l'_n)$ was chosen.

Suppose that it holds for node instance $(v_i, l'_i)$, then it will be proven that it will hold for node instance $(v_{i+1}, l'_{i+1})$, for which $v_{i+1}$ is the node before $v_i$ in $W$ and $l'_{i+1}$ is instance of node $v_{i+1}$, on which $(v_i, l'_i)$ depends, as given by Equation 5.

For $v_{i-1}$ and $v_i$ there are 4 cases:

**Case 1**: $v_{i-1} \neq u \land v_i \neq u$.

In this case $l'_i = l_i$ by the induction assumption and $w'_j(i-1, i) = w_j(i-1, i)$, since $r'(v_{i-1}) = r(v_{i-1})$ and $r'(v_i) = r(v_i)$. Because of Equation 5, that implies $l'_{i+1} = l_{i+1}$.

**Case 2**: $v_{i-1} = u \land v_i \neq u$.

In this case $l'_i = l_i$ by the induction assumption and $w'_j(i-1, i) = w_j(i-1, i) - \Delta r_u \cdot p(i-1, i)$.

**Case 3**: $v_{i-1} \neq u \land v_i = u$.

In this case $l'_i = l_i + \Delta r_u$ by the induction assumption and $w'_j(i-1, i) = w_j(i-1, i) + \Delta r_u \cdot c(i-1, i)$.

**Case 4**: $v_{i-1} = u \land v_i = u$.

In this case $l'_i = l_i + \Delta r_u$ by the induction assumption. The edge in the SDF graph that models this dependency is a loop $(u, u)$. Since the production rate of such an edge is equal to the consumption rate, it holds

$$w_1'(i-1, i) = w_1(i-1, i) - p(i-1, i) \cdot \Delta r_u + c(i-1, i) \cdot \Delta r_u = w_1(i-1, i)$$

Then

$$l'_{i-1} = \left\lfloor \frac{l'_i - w'_j(i-1, i)}{p(i-1, i)} \right\rfloor \in [l_i, l_i + \Delta r_u]$$

As we note in all cases, the $l'_{i-1}$ as given by Equation 5 satisfies Equation 8. For $0 \leq i \leq n$, $l'_{i-1} \geq l_{i-1} \geq 1$ holds. Since the increase of the $r(u)$ value should not violate $P_1, (\forall (v_i, v_{i+1}) \in W : w(v_i, v_{i+1}) \geq 0) \Rightarrow (\forall (u, v) \in E : (u \neq v) \Rightarrow (w(v, u) \geq p(u, v) \cdot \Delta r_u))$.

Because of Equation 5, this implies

$$(\forall i : (0 \leq i < n) \land v_i = u \land v_{i+1} \neq u) \Rightarrow (l_i \leq q_u - \Delta r_u) \Rightarrow (\forall (v_i, v_{i+1}) \in W : w(v_i, v_{i+1}) \geq 0) \Rightarrow (\forall (u, v) \in E : (u \neq v) \Rightarrow (w(v, u) \geq p(u, v) \cdot \Delta r_u))$$

For the case $v_i = u$ and $v_{i+1} = u$, since it holds that $v_n \neq u$, there exists node $v_j$ such that for $j \leq k < u, v_i \neq u$. Suppose $v_{i-1} = u$, then $l'_{i-1} \leq q_u$. That implies that $(\forall i < j : v_i = u : l'_i \leq q_u)$, which proves that for each $v_i = u, 1 \leq l'_i \leq q_u$, and $W'$ exists. □

**Theorem 1** Suppose for a retiming $r$ that $t(v_0, q_0) + d(v_0) = T(r)$. If $\exists r'$ such that $T(r') < T(r)$ and $\forall v \in r'V : r'(v) \geq r(v)$, then $r'(v_0) > r(v_0)$.

**Proof** Since $(t(v_0, q_0) + d(v_0) = T(r))$, there exists a critical walk $W = (v_0, l_0) \rightarrow \ldots \rightarrow (v_n, l_n)$ with $(t(v_0, l_0) = 0)$. Suppose $r'(v_0) = r(v_0)$. The transition from $r$ to $r'$ on the graph can be done by a sequence of transformation increasing the value of one node $u \in V - \{v_n\}$ at a time. However, after each of these transformations a walk $W' = (v_0, l'_0) \rightarrow \ldots \rightarrow (v_n, l'_n)$ will exist in the graph, as shown in Lemmas 2 and 3. Therefore, and because of Lemma 1 $T(r') \geq T(r)$ which is a contradiction. □
Lemma 4 If $r$ is a retiming solution such that $\forall v \in V, r(v, q_v) \leq T(r)$, then $r^* = [r_1 + k \cdot q_1, r_2 + k \cdot q_2, \ldots, r|V| + k \cdot q_{|V|}]$, $\forall k \in Z$, is also a solution with $T(r^*) = T(r)$.

Proof For $r^*$ the number of delays on edge $(u, v)$ is
\[
w'(u, v) = w(u, v) + r'(v) \cdot c(u, v) - r'(u) \cdot p(u, v)
\]
It can be shown that $\exists k$ such that $\forall v \in V, r(v) \geq 0$ and $\exists u, r(u) < q_u$. The retiming solutions for the minimum $T_{\min}$ 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 $T(r) = T_{\min}$ will be called the basic optimal solution. 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 6, if $\exists r : T(r) < T_{\text{step}}$, then the following property holds ($\exists u : r(u) < q_u$).

Proof After initialization, for all nodes $r_0(v) = 0 < q_v$. So, the Lemma holds by the definition of implication. If $T(r_0) = T_{\min}$ then $r_0$ is a basic optimal solution.

Otherwise, $T(r_0) = T_{\text{step}} > T_{\min}$. That means that $\exists r$ such that $T(r) < T_{\text{step}}$. Suppose $r^*$ is a basic optimal solution with $T_{\min} = T(r^*) < T_{\text{step}} = T(r_0)$. For all $v$, $r_0(v) = 0 \leq r'(v)$ and because of Lemma 1, $r_0(v, q_v) + d(v_0) = T(r_0) \Rightarrow r_0(v) < r'(v) \Rightarrow r_0(v) + 1 \leq r'(v)$. Therefore, the invariant $\forall v : r(v) \leq r'(v)$ is preserved after each iteration of the algorithm, and because of that the predicate ($\exists u : r(u) < r'(u) < q_u$) holds.

Each iteration of the algorithm $T_{\text{step}}$, which is a 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
\[
(\forall v : r(v) \geq q_v) \Rightarrow (T(r) < T_{\text{step}})
\]
its (i) $\exists u : r(u) < q_u \Rightarrow (T_{\min} < T_{\text{step}})$

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

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

Proof Follows from the discussion above.

5.3 Second Termination Condition

Lemma 7 After initialization and at each iteration of the algorithm of Figure 6, as long as $\exists r$ such that $T(r) < T_{\text{step}}$, for each node $v$ there exists node $u \neq v$, such that $(u, v) \in E$ and $[r'(v) - r(v)] - [r(u) / q_u] \leq 2$

Proof After initialization the property holds because $\forall v : r(v) = 0$. Suppose that it holds after $k$ iterations of the algorithm. Then at the $k$-th iteration, let $v_0$ be the node with $T(r(v_0, q_{v_0}) + d(v_0) \geq T_{\text{step}}$. The new $r$ value of $v_0$ is $r'(v_0) = r(v_0) + 1$. If the critical v of $v_0$ is composed only of nodes $v_0$ then $2r$ such that $T(r) < T_{\text{step}}$ and the algorithm will exit. Otherwise, we can prove that even after $r(v)$ is increased there exist $u \neq v$ with $(u, v) \in E$, such that $[r'(u) - r(u)] - [r(v) / q_v] \leq 2$.

Moreover, if before the change there existed $z \neq v$ with $(v, z) \in E$ and $[r'(z) - r(z)] - [r(v) / q_v] \leq 2$, this will continue to hold after the change, as well.

Let $u$ be the last node in the critical walk with $u \neq v_0$. If $u$ is the $k$-th node of the walk, the step $v_0 = u$ and $v_{k-1} = v$. Then in order for the walk to be valid for the successive elements $(u, l)$ and $(v_{k-1}, 1)$, it must hold $1 \leq l_i$. A necessary condition for $1 \leq l_i$, because of Equation 5, is that
\[
\frac{1}{q_v} c(u, v) - \frac{1}{q_u} p(u, v) + r'(v) - r(v) \geq 0
\]

Suppose there was node $z$ in the graph, with $(v, z) \in E$, for which $[r'(z) - r(z)] - [r(v) / q_v] \leq 2$. Then after the change, since $r'(v) > r(v), [r'(z) - r(z)] - [r(v) / q_v] \leq 2$.

Therefore, by induction the property holds.

Lemma 8 After initialization and at each iteration of the algorithm of Figure 6, if $\exists r : T(r) < T_{\text{step}}$, then the following property holds $(\forall v : r(v) \leq 2 \cdot q_v \cdot |V|)$.

Proof After initialization and at each iteration of the algorithm, if $\exists r : T(r) < T_{\text{step}}$, then $(\exists u : r(u) < q_u)$ (Lemma 5). For $u$ the value $[r'(u) - r(u)] / q_u = 0$.

Suppose that $\exists v$ such that $r(v) > 2 \cdot q_v \cdot |V|$. This implies that $[r'(v) - r(v)] / q_v \geq 2 \cdot |V|$. That means that $\exists u \neq v$ such that $[r'(v) - r(v)] / q_v \geq 2 \cdot |V| - [r(u) / q_u] \leq 2$.

Continuing like that for the rest $|V| - 1$ nodes of the graph, it can be proven that the minimum $r$ value of the last node $v_{|V| - 1}$ of this sequence will be

$2 \cdot |V| - (|V| - 1) \leq |V| - [r'(v) - r(v)] / q_v \Rightarrow 2 \cdot q_v \leq r'(v)$ - If that holds, then $(\exists u : r(u) < q_u)$ which is a contradiction.
can be
\[ \sum_{v \in V} r(v) \leq \sum_{v \in V - \{u\}} (2 \cdot |V| \cdot q_v) + (q_u - 1) + 1 \]

The node \( u \) is assumed to be the one for which the redundant optimal solution holds. The \( r \) values of the rest of the nodes form the first term and 1 more move is needed to terminate the algorithm.

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

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 \( 2 \cdot |V|^2 \cdot q_{ave} \).

In each iteration the necessary arrival times are computed. In the worst case the arrival computation will take
\[ \sum_{v \in V} q_v = |E| \cdot |V| \cdot q_{ave} \]

Therefore, the total worst case complexity will be \( O(|V|^3, |E|, q_{ave}) \).

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 \( v_0 \) of the walk \( T_{step} \leq t(v_0, \lambda_0) + d(v_0) \), then for any \( v' \) for which \( T(v') < T_{step} \), the \( r \) value of \( v \) will be \( r(v') < r(v) \).

Secondly, from Lemmas 1-3 we see that by increasing the \( r(v_0) \) value of a node for which \( t(v_0, q_{a0}) + d(v_0) \geq T_{step} \) cannot improve the arrival time of nodes \( v_n \neq v_0 \). Therefore, if before the \( r(v_0) \) change, \( t(v_0, q_{a0}) + d(v_0) \geq T_{step} \) was valid, after the change \( t(v_0, q_{a0}) + d(v_0) \geq T_{step} \) remains valid.

Using these conclusions, the algorithm can be modified to store all nodes, which have \( t(v_m, q_{a0}) + d(v_m) \geq T_{step} \), each time the arrival times are computed. Then modify their \( r \) values and then compute the arrival times again. That way though, 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
\[ w(a, v) + r(v) \cdot c(a, v) + \Delta r(v) \cdot c(a, v) - r(v) \cdot p(a, v) \geq 0 \]
\[ \Rightarrow \Delta r(v) \geq \frac{[r(a) \cdot p(a, v) + w(a, v)] - r(v)}{c(a, v)} \]

Since \( r(v) \) is less than or equal to \( r^*(u) \), \( r^*(v) \) 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 9 two queues are maintained. The first queue (Q1) holds the nodes for which it is known that their values must be increased for \( T_{step} \) 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 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

```plaintext
Algorithm SDF_Retiming_Improved
Input: An SDF graph \( G = [V, E, d, p, c, w] \).
Output: A pair \([r, T_{min}]\) which represents an optimal retiming \( r \) satisfying minimum complete cycle execution time \( T_{min} \).
1. maxt \( \leftarrow 0 \);
2. Q1 \( \leftarrow 0 \);
3. Q2 \( \leftarrow 0 \);
4. while \((\exists v: r(v) < q_v \& \& \exists v: r(v) > 2q_v \cdot |V|)\) do
   5. init() ;
   6. for each \( v \) in \( V \) do
      7. \( r(v) \leftarrow r(v) + 1 \);
      8. foreach \( \{v, u\} \in E \) do
         9. if \((w(v, u) < 0)\) then
            10. \( Q2 \leftarrow (v, u) \);
            11. fi;
         12. endfor;
   13. endwhile;
   14. while \((Q2 \neq \emptyset)\) do
      15. \( x, u \leftarrow Q2.dequeue() \);
      16. \( \Delta r(u) \leftarrow \frac{[r(x) \cdot p(x, u) + w(x, u)] - r(u)}{c(x, u)} \);
      17. if \((\Delta r(u) > 0)\) then
         18. \( r(u) \leftarrow r(u) + \Delta r(u) \);
         19. foreach \( \{u, z\} \in E \) do
            20. if \((w(u, z) < 0)\) then
               21. \( Q2 \leftarrow (u, z) \);
               22. fi;
            23. endif;
         24. endif;
   25. endwhile;
   26. init();
   27. for each \( v \) in \( V \) do
      28. \( (v, q_v) \leftarrow get_\lambda(v, q_v) \);
      29. if \((t(v, q_v) + d(v)) > maxt\) then
         30. \( maxt \leftarrow t(v, q_v) + d(v) \);
         31. fi;
   32. endfor;
   33. return \([r, T_{min}]\);
Figure 9: Pseudocode of the improved retiming algorithm.
```
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 \( \text{max} < T_{\text{stop}} \), \( Q1 \)'s unique entry is the node \( v \) for which \( r(v_0, q_0) + d(v_0) = \text{max} \). Otherwise, all nodes for which \( r(v, q) + d(v) \geq T_{\text{stop}} \) 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 Channel

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 \( l \) with \( q_l = 1 \) and \( d(l) = 0 \). Then for each source \( s \) of the graph an
Table 2: Results for graphs generated with $q_{max} = 4$.

<table>
<thead>
<tr>
<th>Graph</th>
<th>O’Neil’s First</th>
<th>Improved</th>
<th>O’Neil’s First</th>
<th>Improved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{27}$</td>
<td>104</td>
<td>104</td>
<td>0.014</td>
<td>0.006</td>
</tr>
<tr>
<td>$s_{208.1}$</td>
<td>185</td>
<td>152</td>
<td>0.162</td>
<td>0.049</td>
</tr>
<tr>
<td>$s_{298}$</td>
<td>174</td>
<td>174</td>
<td>0.425</td>
<td>0.086</td>
</tr>
<tr>
<td>$s_{244}$</td>
<td>259</td>
<td>180</td>
<td>0.242</td>
<td>0.140</td>
</tr>
<tr>
<td>$s_{349}$</td>
<td>310</td>
<td>255</td>
<td>0.893</td>
<td>0.153</td>
</tr>
<tr>
<td>$s_{352}$</td>
<td>414</td>
<td>414</td>
<td>2.612</td>
<td>0.112</td>
</tr>
<tr>
<td>$s_{386}$</td>
<td>275</td>
<td>275</td>
<td>0.495</td>
<td>0.140</td>
</tr>
<tr>
<td>$s_{444}$</td>
<td>202</td>
<td>202</td>
<td>0.310</td>
<td>0.123</td>
</tr>
<tr>
<td>$s_{526}$</td>
<td>632</td>
<td>604</td>
<td>0.859</td>
<td>0.314</td>
</tr>
<tr>
<td>$s_{641}$</td>
<td>234</td>
<td>226</td>
<td>0.340</td>
<td>1.193</td>
</tr>
<tr>
<td>$s_{820}$</td>
<td>256</td>
<td>247</td>
<td>1.034</td>
<td>0.473</td>
</tr>
<tr>
<td>$s_{953}$</td>
<td>430</td>
<td>430</td>
<td>2.388</td>
<td>1.127</td>
</tr>
</tbody>
</table>

Table 3: Results for graphs generated with $q_{max} = 16$.

<table>
<thead>
<tr>
<th>Graph</th>
<th>O’Neil’s First</th>
<th>Improved</th>
<th>O’Neil’s First</th>
<th>Improved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{27}$</td>
<td>129</td>
<td>104</td>
<td>104</td>
<td>0.084</td>
</tr>
<tr>
<td>$s_{208.1}$</td>
<td>538</td>
<td>538</td>
<td>538</td>
<td>29.014</td>
</tr>
<tr>
<td>$s_{298}$</td>
<td>765</td>
<td>704</td>
<td>704</td>
<td>2m:18.526</td>
</tr>
<tr>
<td>$s_{344}$</td>
<td>975</td>
<td>905</td>
<td>905</td>
<td>6m:29.149</td>
</tr>
<tr>
<td>$s_{349}$</td>
<td>1124</td>
<td>907</td>
<td>907</td>
<td>2m:01.058</td>
</tr>
<tr>
<td>$s_{386}$</td>
<td>795</td>
<td>701</td>
<td>701</td>
<td>1m:04.163</td>
</tr>
<tr>
<td>$s_{444}$</td>
<td>1140</td>
<td>840</td>
<td>840</td>
<td>36.331</td>
</tr>
<tr>
<td>$s_{526}$</td>
<td>1528</td>
<td>1498</td>
<td>1498</td>
<td>15m:05.460</td>
</tr>
<tr>
<td>$s_{641}$</td>
<td>897</td>
<td>624</td>
<td>624</td>
<td>19.648</td>
</tr>
<tr>
<td>$s_{820}$</td>
<td>895</td>
<td>816</td>
<td>816</td>
<td>30.478</td>
</tr>
<tr>
<td>$s_{953}$</td>
<td>819</td>
<td>773</td>
<td>773</td>
<td>26.242</td>
</tr>
</tbody>
</table>

Table 4: Results for graphs generated with $q_{max} = 32$.

<table>
<thead>
<tr>
<th>Graph</th>
<th>O’Neil’s First</th>
<th>Improved</th>
<th>O’Neil’s First</th>
<th>Improved</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{27}$</td>
<td>459</td>
<td>416</td>
<td>416</td>
<td>1.924</td>
</tr>
<tr>
<td>$s_{208.1}$</td>
<td>834</td>
<td>834</td>
<td>834</td>
<td>2m:50.537</td>
</tr>
<tr>
<td>$s_{298}$</td>
<td>1083</td>
<td>1027</td>
<td>1027</td>
<td>5m:30.897</td>
</tr>
<tr>
<td>$s_{344}$</td>
<td>2534</td>
<td>2408</td>
<td>2408</td>
<td>70m:29.472</td>
</tr>
<tr>
<td>$s_{349}$</td>
<td>1503</td>
<td>1415</td>
<td>1415</td>
<td>8m:18.381</td>
</tr>
<tr>
<td>$s_{386}$</td>
<td>1312</td>
<td>1273</td>
<td>1273</td>
<td>19m:29.061</td>
</tr>
<tr>
<td>$s_{444}$</td>
<td>938</td>
<td>806</td>
<td>806</td>
<td>1m:40.775</td>
</tr>
<tr>
<td>$s_{526}$</td>
<td>1185</td>
<td>888</td>
<td>888</td>
<td>48m:18.215</td>
</tr>
<tr>
<td>$s_{641}$</td>
<td>2161</td>
<td>2007</td>
<td>2007</td>
<td>120m:00.000</td>
</tr>
<tr>
<td>$s_{820}$</td>
<td>690</td>
<td>610</td>
<td>610</td>
<td>54.758</td>
</tr>
<tr>
<td>$s_{953}$</td>
<td>1594</td>
<td>1573</td>
<td>1573</td>
<td>46m:26.437</td>
</tr>
<tr>
<td>$s_{1776}$</td>
<td>1776</td>
<td>1776</td>
<td>5m:26.620</td>
<td>16.680</td>
</tr>
</tbody>
</table>

Table 5: Results for zero-delay node graphs generated with $q_{max} = 32$.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Initial</th>
<th>Final</th>
<th>Execution Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_{27}$</td>
<td>368</td>
<td>351</td>
<td>0.005</td>
</tr>
<tr>
<td>$s_{208.1}$</td>
<td>1035</td>
<td>852</td>
<td>0.020</td>
</tr>
<tr>
<td>$s_{344}$</td>
<td>1062</td>
<td>928</td>
<td>0.164</td>
</tr>
<tr>
<td>$s_{349}$</td>
<td>933</td>
<td>833</td>
<td>0.016</td>
</tr>
<tr>
<td>$s_{386}$</td>
<td>951</td>
<td>908</td>
<td>0.021</td>
</tr>
<tr>
<td>$s_{444}$</td>
<td>902</td>
<td>882</td>
<td>0.027</td>
</tr>
<tr>
<td>$s_{526}$</td>
<td>1690</td>
<td>1690</td>
<td>0.009</td>
</tr>
<tr>
<td>$s_{641}$</td>
<td>694</td>
<td>665</td>
<td>0.011</td>
</tr>
<tr>
<td>$s_{820}$</td>
<td>1264</td>
<td>1219</td>
<td>0.002</td>
</tr>
<tr>
<td>$s_{953}$</td>
<td>1758</td>
<td>1558</td>
<td>0.010</td>
</tr>
</tbody>
</table>

8 Experimental Results

In this Section we present the experimental results obtained by applying the retiming algorithms on a number of graphs. Firstly, the
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 q value of each node was also selected randomly between 1 and the value q_{max}. Three values (4, 16, 32) have been used for q_{max} to observe how the performance of the algorithms scales with this parameter. After the q value of each node was assigned, the p 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_i}{\gcd(q_i, q_j)} and c(u, v) = \frac{q_j}{\gcd(q_i, q_j)}.

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. I 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_{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_1, it will not return false for cycle length T_2 > T_1. 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 on the randomly generated graph. In some cases it is 0 and in other cases it can be more then 20%. The execution time of the improved version is much faster than the other two algorithms, especially for larger graphs. As q_{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_{ave} [5].

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) \geq 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_{max} = 32. The other two algorithms cannot be applied on this problem instance. For O’Neil’s algorithm it is not known how it can handle constraints like P6. Moreover, the first version of the retiming algorithm cannot handle constraints like P6 and under presence of zero delay nodes its correctness does not hold.

9 Conclusions

In this paper two optimal algorithms were presented for minimum cycle length retiming of SDF graphs. The first is an optimal algorithm for retiming strongly connected graphs, whereas the second works on any graph including 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


