Retiming Revisited and Reversed

Guy Even, *Member, IEEE*, Ilan Y. Spillinger, *Senior Member, IEEE*, and Leon Stok, *Senior Member, IEEE*

***Abstract*— Retiming is a very promising transformation of circuits which preserves functionality and improves performance. Its benefits are especially promising in automatic synthesis of circuits from higher-level descriptions. However, retiming has not been widely included in current design tools and methodologies. One of the main obstacles is the problem of finding an equivalent initial state for the retimed circuit. In this paper, we introduce a simple modification of the retiming algorithm of Leiserson and Saxe. The modified algorithm helps minimize the effort required to find equivalent initial states and reduces the chance that the network needs to be modified in order to find an equivalent initial state. This algorithm is the kernel of a new efficient retiming method, which searches for optimal retimings while preserving the initial state condition. The paper also presents an improved method to perform the initial state calculation.**

I. Introduction

RETIMING is a transformation which improves circuits by relocating registers in sequential circuits. It has been shown in [6] and [7] that (under certain restrictions) this transformation preserves the functionality of circuits. Retiming may be applied for several optimization goals, e.g., minimizing the cycle time, minimizing the area, minimizing the number of registers, or improving testability.

An example of retiming a circuit is shown in Fig. 1(a). The initial circuit has three registers $r1$, $r2$, and $r3$, and a maximum combinatorial delay of three units through gates $g2$, $g3$, and $g4$. (Assuming unit delay model, no fan-in fan-out delays). In order to reduce the cycle time to two units and minimize the registers to two, we can retime gate $g4$. The two registers $r2$ and $r3$ at its outputs are moved to the input and replaced by register $r4$. The retimed circuit is shown in Fig. 1(b).

A retiming can be described by an integer function $L()$ (called the lag) of all nodes in the network. This function represents the number of registers that are to be moved from each output of node v to each of its inputs. In Fig. 1, one register is removed from each output and one register is inserted in the input. Therefore, the lag of gate $g4$ equals one, $L(g4) = 1$. Note that a positive lag causes registers to move backward in the network, while a negative lag moves them forward, where the forward direction is defined as the direction in which the data flows through the circuit.

Manuscript received July 11, 1994; revised March 22, 1995 and December 7, 1995. This work was done while the authors were working in the BooleDozer Logic Synthesis Group of the IBM T. J. Watson Research Center, Yorktown Heights, NY. This paper was recommended by Associate Editor K. Keutzer.

G. Even is with the University of Saarland, Saarbruecken, Germany.

I. Spillinger is with Intel Israel, Haifa 61015, Israel.

L. Stok is with IBM T. J. Watson Research Center, Yorktown Heights, NY 10598 USA.

Publisher Item Identifier S 0278-0070(96)01847-7.

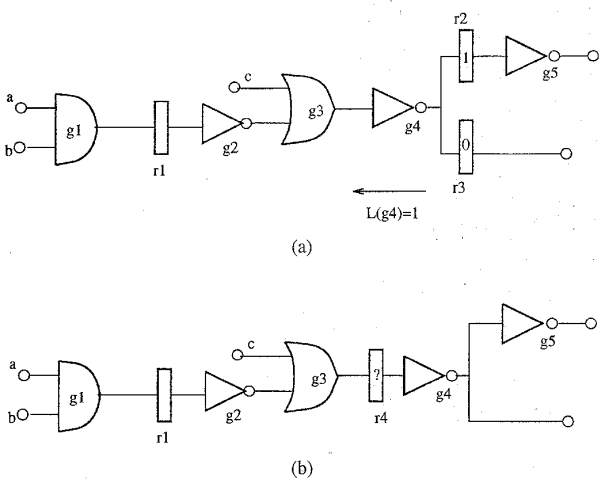


Fig. 1. (a) Original circuit. (b) Retimed circuit.

The initial state of a circuit is determined by the initial values of the registers in the circuit. For a limited set of applications, e.g., for the data path circuitry in DSP type circuits, the initial state is not important and retiming without additional constraints can be applied [10]. However, in many microprocessor and controller type applications, the initial state is an integral part of the behavior of the machine.

Whenever the initial state of the sequential circuit is meaningful, it is necessary to find an equivalent initial state for the retimed circuit. However, it is not always possible to find the initial state of the retimed circuit. For example, let us assume that in Fig. 1 the initial value of register $r2$ is 1 and the initial value of register $r3$ is 0. The retimed circuit cannot be initialized to have the same behavior as the original circuit and, in particular, one cannot find an initial value for the new register $r4$.

A retimed circuit has an initial state equivalent to an initial state in the original circuit if for any input sequence applied to both circuits (one circuit started in the initial state, the other in the equivalent one) the same sequence of outputs is produced.

One way to assure that a corresponding initial state can be found in the retimed network is to only move registers forward in the network [4]. Let us define this as *simple forward retiming*. In this case, the initial state can be propagated to the new register positions by a simple simulation (i.e., forward implication) of the values in the network.

Let us define *forward retiming* as a generalization of simple forward retiming. In forward retiming registers are not only allowed to move forward, but registers can also be removed

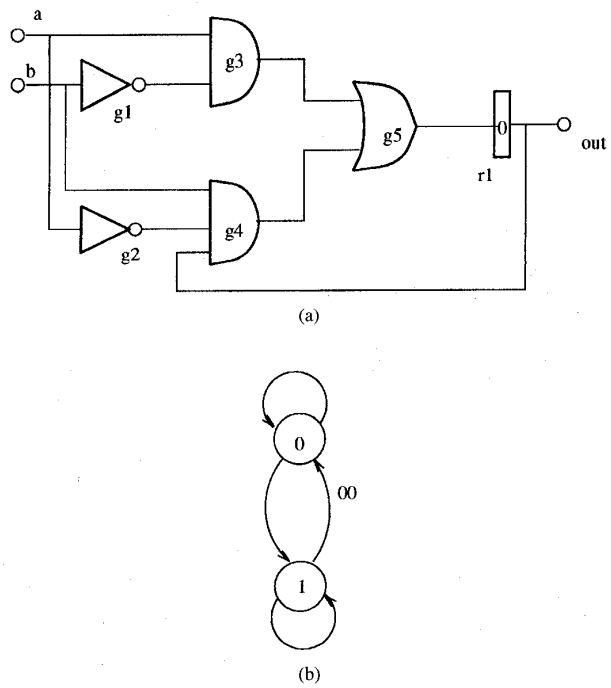


Fig. 2. (a) Circuit with no simple forward retiming. (b) State machine.

at the primary outputs and reinserted at the primary inputs. Forward retiming through the primary outputs/primary inputs (O–I's) removes a register from each path that ends in a primary output and inserts one on each path that starts from a primary input [9]. Notice that the number of registers along each path from a primary input to a primary output is unchanged by retiming [8]. The restriction of simple forward retiming is motivated by the ability to track initial values since retiming through O–I's is not allowed. As shown in Fig. 2, this imposes a significant restriction on the retiming and excludes various retimings. The circuit in Fig. 2 cannot be retimed to obtain a delay of two units by allowing only simple forward moves, although a clock period of two units is obtainable via retiming without this restriction.

Eliminating the constraint on retiming through the O–I's makes forward retiming a general retiming. Every backward retiming can be obtained by applying a sequence of forward retimings through O–I's.

The basic problem is to determine the initial values for the registers inserted in the primary input paths. For example, the register $r1$ in Fig. 2 can be duplicated. One of the duplicates is directly connected to the last input of gate $g4$, the other directly feeds the primary output. Removing the register from the output and inserting two new registers at the inputs a and b produces a valid retiming. But we can not easily calculate the initial values for these new registers by forward implication.

Touati and Brayton [11] describe a method which finds a sequence of input values to be inserted at the inputs to find the appropriate initial values of the registers in the retimed circuit. Given a particular legal retiming, one can derive the number of forward moves through the O–I's necessary to modify the network to obtain this retiming. Let us call this number of forward O–I moves k. A sequence of k input values is needed, which prescribes the values inserted in each O–I move. This sequence can be obtained by inspecting the state machine extracted from the circuit. In this state machine, a sequence of k transitions must be found which will bring the machine into the initial state. Any state may be the starting point of such a sequence. Each time the inputs are retimed (i.e., registers are inserted in the input paths), they are initialized with the values from this input sequence.

Let us apply this method to the example of Fig. 2. The state machine for this network has two states. Assume that we want to find an initial state for the retiming of gate $g5$ by one, $L(g5) = +1$, by repeated forward moves. This requires one forward O–I move. Therefore, the length of the input sequence to initialize the new registers at the inputs equals one, $k = 1$. In the partial state diagram of Fig. 2, we have to search for a sequence of one transition that leads to state 0. The transition $(a = 0, b = 0)$ brings us from state 1 to state 0 and can be used. When the register is removed from the output *out*, two new registers are inserted at the inputs a (initialized with a 0) and b (initialized with a 0). These new registers can be moved forward through gates $g1, \cdots, g4$ by simple forward moves to their final positions at the inputs of $g5$.

To be able to execute this method to find an initialization, the state machine is required to have a sequence of state transitions of length k leading to the initial state. If this is not the case, the circuit must be *modified* to include such a sequence.

A circuit which cannot be initialized after retiming is shown in Fig. 3, which shows both the circuit and the state diagrams. Since there is no transition that leads to the initial state, a transition needs to be added. A state is searched that has the minimum Hamming distance to the initial state 10. State 00 is selected and a new transition is added. The new transition from state 00 to state 10 has input values $a = -$, $b = 1$, $c = 0$ and $reset = 1$. The new state diagram is shown in Fig. 4(b). All other transitions have a $reset = 0$ (not shown) attached to them. In the initial state calculation, these values are used when the latches are moved across the O–I boundaries. This leads to the initial values in the registers as shown in Fig. 4(a).

Unfortunately, this modified circuit has a maximum delay of two due to the additional (constant) reset input and the OR-gate needed to merge this into the path. This example shows that modifying the logic according to the approach in [11] might ruin the advantage obtained by retiming in the first place, and require additional hardware.

However, another retiming may exist which enables one to find an initial state without requiring modifications to the network. An example of a retiming for this circuit is shown in Fig. 5. This is the only retiming with cycle time one which has an equivalent initial state without modifications to the combinational logic. The retimed circuit has four registers and their initial values are as shown in the figure. Interestingly, the final circuit contains an unshared register at the output of the inverter $g1$. This is sometimes needed to obtain an initializable retiming with a minimal cycle time.

The major contribution of this paper is that we explore the existence of retimings, which require no combinational circuit

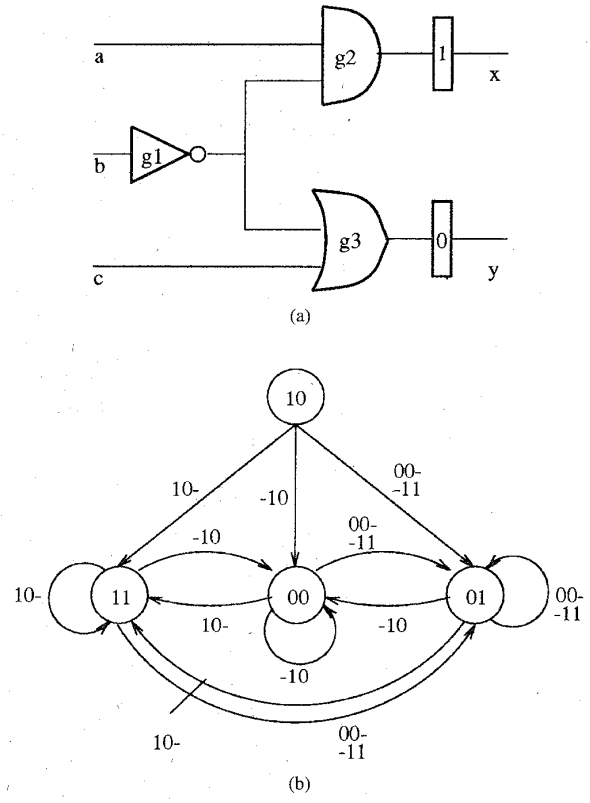


Fig. 3. (a) Circuit and (b) corresponding state machine.

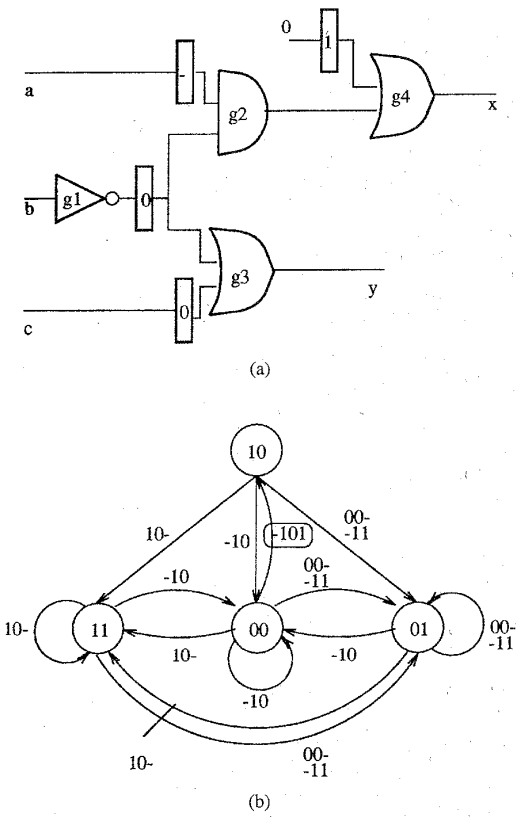


Fig. 4. (a) Circuit resulted from (b) modified state machine.

modifications. The *reverse retiming* is a variation of the best known retiming algorithm (FEAS) [7], and therefore has the same complexity. The algorithm finds a retiming such that the number of registers that move backward through a single combinational block is the minimized among all possible retimings. Whenever there exists a simple forward retiming, the modified algorithm finds it.

The rest of this paper consists of two main parts. Section II describes the reverse retiming algorithm and shows why this modification to retiming is crucial in order to obtain better initializable retimed circuits. Section III explains a new procedure to calculate the initial states.

In Section IV, the two previous sections are combined in a retiming method which bounds lags of specific combinational logic blocks. Section V classifies circuits with respect to retiming and initialization and describes why our method is applicable to a larger set of circuits than earlier methods. Finally, Section VI shows that the results of our experiments are in accordance with the claims on the retiming method and illustrates the different categories of circuits that actually exist in the common benchmarks.

II. Reverse Retiming

As explained in the Introduction, backward retiming steps should be avoided as much as possible. The basic difficulty with backward moves is the existence of a mapping for the initial state. Finding this mapping is an NP-hard problem, as it is similar to the phase of justification in the process of automatic test pattern generation [5]. So whenever possible, a good criterion is to minimize the number of registers that move backward through each functional (combinational) block.

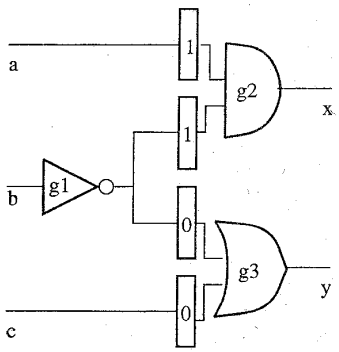


Fig. 5. Retimed circuit with cycle time one (Reversed Retiming).

A circuit graph $G = (V, E, w, d)$ consists of a directed graph (V, E) with nonnegative integer weights $w(e)$ on the edges and nonnegative real delays $d(v)$ on the vertices. The weights on the edges model the number of registers along the edge, the delays of the vertices model the propagation

delay of the nodes. Given a path $p = [v_0, \cdots, v_k]$, its weight is defined as $w(p) = \sum_{i=0}^{k-1} w(e_i)$, where e_i is the edge from v_i to v_{i+1}. The delay is defined as $d(p) = \sum_{i=0}^{k} d(v_i)$. The minimum feasible clock period of G, $\Phi(G)$ is defined as $\Phi(G) = \max\{d(p)|w(p) = 0\}$.

A circuit graph can be derived from an actual circuit $C(B, R, N)$ by replacing the combinational blocks B by nodes V, the nets N by edges E between the nodes and the registers R by weights w on these edges [7]. In the graph model, a special node with zero delay called the *host* and denoted by h is added. For every primary output of the network, an edge with zero weight is inserted from the output to the host. For every primary input an edge is added from the host to the input.

Let $L(v)$ be the lag function for all nodes $v \in V$, $e(u, v)$ an edge from u to v, $w(e)$ the weight of the edges before retiming and $w_L(e)$ the weights of the edges after retiming. w_L is determined from w and the lags by: $w_L(e) = w(e) + L(v) - L(u)$.

The normalized lag $L^*(v)$ of a node v is defined as the difference between the lag of the node itself and the lag of the host, i.e., $L^*(v) = L(v) - L(h)$. L^* is the maximum $L^*(v)$ over all nodes in the circuit. Since the retimed weight $w_L(e)$ is only dependent on the difference of the lags between two nodes and not on the absolute value of the lags, retiming with the normalized lags will result in the same circuit as retiming with the original lags.

For each node v, a required time $t_r(v)$ is defined. For the simplicity of the discussion, we assume that all primary outputs and registers are synchronized with the clock and that their required time equals the required cycle time c. The required time for the host equals the cycle time c. For any other node v the required time is defined as the difference between the smallest required time for its successors and the propagation delay of the node itself $t_r(v) = \min_{(v,u)\in E}(t_r(u)) - d(v)$. Notice, the reverse_retiming algorithm can be easily extended to handle different required times for the outputs and registers.

The reverse_retiming algorithm, described in algorithm 1, first sets the lags of all nodes to zero. In the outer loop, it retimes the circuit according to the lags L and recomputes the set M of nodes whose outputs are not in time to meet a required time. The lags of these nodes are decreased by one. If none of the nodes violates a constraint, the iteration can be stopped. If after $|V|$ iterations M is nonempty, then no solution that meets the requirements is possible for G and the algorithm terminates.

```
Algorithm 1. reverse_retiming(G, c)
foreach v ∈ V do
  L(v) = 0;
k = 1;
do
  Compute retimed circuit G_L;
  Compute t_r(v) for every vertex v ∈ V;
  M = {v|t_r(v) < 0}
  foreach v ∈ M do
    L(v) = L(v) − 1;
  k = k + 1;
while k ≤ |V| and M ≠ ∅;
```

All claims for the retiming algorithms FEAS in [7] and retime in [9] hold. In addition, reverse retiming finds the retiming with the minimum normalized lag, as expressed in the following theorem.

Theorem 1: Let $G = (V, E, w, d)$ be a circuit and c a required clock cycle. Let $L(v)$, denote the retiming computed by the algorithm reverse_retiming(G, c). Then

1) The algorithm reverse_retiming(G, c) finds a retiming L such that $\Phi(G_L) \leq c$, if such a retiming exists.
2) If all nodes are reachable from the host and if $\Phi(G_L) \leq c$, then for every retiming L' for which $\Phi(G'_L) \leq c$ the following holds

$$\max_{v\in V} L(v) - L(h) \leq \max_{v\in V} L'(v) - L'(h).$$

A sketch of the proof of the theorem appears in the Appendix. The first implication in the theorem shows that a retiming for a given cycle time will be found if one exists, similar to the FEAS algorithm. The second result expresses that the retiming that is found achieves the smallest maximum normalized lag value.

The following example shows that the difference in the maximum normalized lag between the FEAS algorithm and the reverse retiming might be as large as $|V|$. The circuit graph shown in Fig. 6(a) has n registers at its input, n registers at its output, and n combinational nodes each with a unit delay. The FEAS algorithm (described in the Appendix), applied to this example (see Fig. 6(b)), assigns a lag of $L(v_i) = i$ to each node $v_i, 0 \leq i \leq n$ and the lag of the host equals $L(h) = 0$. This retiming therefore results in a maximum normalized lag of n, i.e., $L^*(V_n) = L(V_n) - L(h) = n - 0 = n$.

The reverse_retiming algorithm (see Fig. 6(c)) assigns a lag $L(v_i) = -(n - i)$ to each node $v_i, 0 \leq i \leq n$ and the lag of the host $L(h) = 0$. Since the maximum normalized lag is zero, the reverse_retiming result can be obtained by only simple forward moves.

In other words, reverse retiming will always find a retiming with the minimum number of forward O–I moves. Note that if a retiming with a maximum normalized lag value of zero exists (i.e., a simple forward retiming), reverse retiming finds such a solution. However, there remain circuits that do not have a retiming with a maximum normalized lag of zero. For these circuits, initial state calculation is not possible by simple forward simulation and a more advanced method is required. The next section will describe such a method.

III. Initial State Calculation

Given a retiming for a circuit graph $G(V, E, w, d)$, a retiming function can be defined for the circuit $C(B, R, N)$. By construction there is a one-to-one correspondence between a node $v \in V$ and a combinational block $b \in B$. The lag of a block b is defined equal to the lag of the corresponding node v. A retimed circuit C' for a lag function $L()$ is constructed using the *Update_Registers* algorithm described in algorithm 2. This algorithm simultaneously calculates the new positions and the initial values for the registers, such that the initial state of the retimed circuit is equivalent to the initial state of the original circuit.

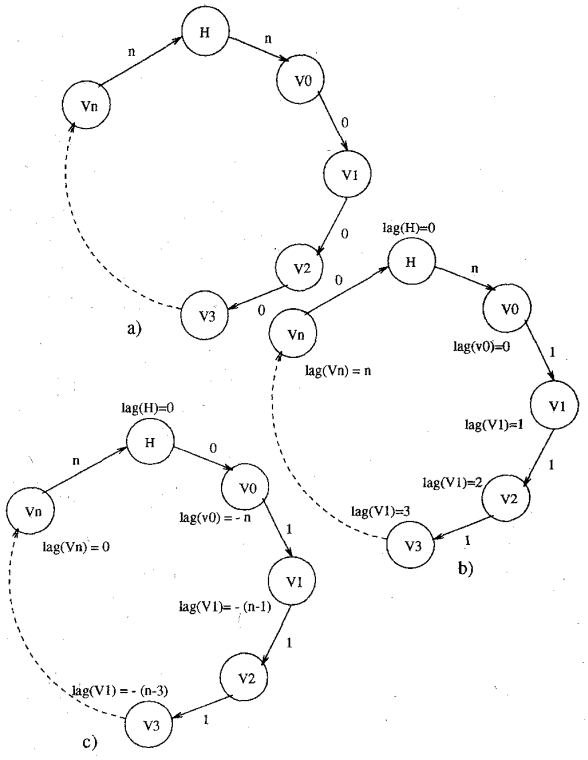


Fig. 6. (a) Cyclic circuit graph. (b) Retimed using FEAS. (c) Retimed using reverse retiming.

All registers in the original circuit C have contents zero, one, or don't care as the initial state. The *Update_Registers* algorithm iterates over all registers in the design. For each register r, it is checked for each of its outputs if it was already visited. If one of the outputs of r was visited, r is added to the implication set I. Otherwise it is checked if the lag at this output is negative. If true, this register r is inserted in the implication set I. Also, the design is traversed forward (toward the primary outputs). If it reaches a node u which has not been visited in this iteration, and its lag value is negative then its lag is incremented by one, and the traversal in the forward direction continued. Whenever it reaches a visited node, the traversal is stopped. Whenever it reaches a node u with lag $L(u) \geq 0$ or it reaches a register pr, a new register nr is introduced before u or pr and the traversal in this direction is stopped. In Fig. 7, register $r1$ is inserted in list I and the lag of $g1$ is incremented. The forward cone stops at gate $g1$, so a new register $nr1$ is inserted after $g1$.

In a similar way, for each register in the design, it is checked if one of its unvisited inputs has a positive lag. If so, traverse the design backward, decrement the lags of the appropriate nodes, insert new registers, and update the justification list J. In the example (Fig. 7), J will contain register $r2$ and two new registers $nr2$ and $nr3$ are inserted in front of gate $g2$. All new registers inserted in the circuit have a don't care initial value.

```
Algorithm 2. Update_Registers C(B,R,N),L())
do
I = J = ∅;
foreach u ∈ B do visited(u) = FALSE;
foreach r ∈ R do
  foreach b ∈ {b|∃(r,b) ∈ N} do
  if (visited(b) == TRUE) then
     I = I + r;
  else if (L(b) < 0) then
     I = I + r;
     Build the forward cone FC from
     and including b;
     Stop when you reach a node u such
     that visited(u) == TRUE;
     Also stop when you reach a node u such
     that L(u) ≥ 0 or reach a register pr then
       insert a new register nr with
       value don't care before u or pr;
     foreach u ∈ FC do
       L(u) = L(u) + 1;
       visited(u) = TRUE;
     endforeach
     endif
   endif
   endforeach
   foreach b ∈ {b|∃(b,r) ∈ N } do
   if (visited(b) == TRUE) then
     J = J + r;
   else if (L(b) > 0) then
     J = J + r;
     Build the backward cone BC from and
     including b;
     Stop when you reach a node u such that
     visited(u) == TRUE;
     Also stop when you reach a node u such
     that L(u) ≤ 0 or reach a register pr then
       insert a new register nr with value
       don't care after u or pr;
     foreach u ∈ BC do
       L(u) = L(u) − 1;
       visited(u) = TRUE;
     endforeach
   endif
   endforeach
endforeach
if (I ∪ J = ∅) then return (success);
Forward implication for all values of
registers in I and
justification of all values in
J using justify [5];
if (justify fails) then
   return (failure);
else
   Remove all registers r ∈ I ∪ J from C;
endif
while TRUE;
```

Only if I and/or J are nonempty, retiming moves have been done. For all values of registers in I, a *forward implication*

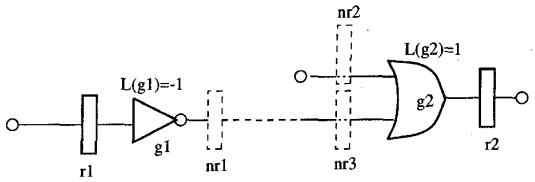


Fig. 7. Register update procedure.

procedure is called. For all values of registers in J, a *backward justification* is used. Both of these procedures are described in [5]. If justification fails, there is no possible equivalent retimed initial state. If it succeeds, the appropriate register values are updated. All registers in the lists I and J (they have been replaced by new registers) can be deleted, and the $Update\_Registers$ algorithm proceeds to the next iteration. The number of iterations of the outer loop of the algorithm that is required for a successful computation of the retimed equivalent initial state is the maximum absolute value of the normalized lags. Within this iteration a justification step is done which in theory is NP-complete. Since this justification is done only on a very small portion of the design the run time in practice is small.

Reverse retiming and the initial state calculation can be combined in a method which is the topic of the next section.

IV. Bounding the Lags of Specific Blocks

Suppose that during the update of registers, one fails in the justification process, e.g., the backtracing computation for a block b causes a conflict. In such a case, it may be helpful to bound the normalized lag of block b, to avoid the need to backtrace its computation so many times.

```
Algorithm 3. Reverse Retiming Method
1) Build circuit graph G for circuit C.
2) Apply algorithm reverse_retiming on
(G, c).
3) If such a retiming does not exist
then stop.
4) Normalize the lags obtained for G.
5) Copy the resulting lags from G to C.
6) Apply algorithm Update_Registers on C.
7) If successful, stop.
8) Insert the additional edges in G
to form G^m.
9) G ≈ G^m.
10) Goto step (2).
```

For every block b, that could not be backtraced during the ith iteration of the $Update\_Registers$ algorithm, the lag is bounded by $i-1$. This is accomplished by adding an edge of weight $i-1$ from the node v_b, which corresponds to the block b, to the host h. Let G^m denote the new circuit graph. The optimality of applying reverse retiming on the graph G^m is summarized in the following claim.

Claim: Reverse retiming on the graph G^m with required clock period c finds a feasible retiming if such a retiming exists that minimizes the maximum normalized lag. In addition, it satisfies the constraints $L(v_b) - L(h) \leq i-1$.

Proof: Any feasible retiming, L', of G^m must satisfy

$$\begin{aligned} w_{L'}(v_b, h) &= w(v_b, h) - L'(v_b) + L'(h) \\ &= i - 1 - L'(v_b) + L'(h) \\ &\geq 0 \end{aligned}$$

therefore, $L'(v_b) - L'(h) \leq i-1$, as required.

Theorem 1 guarantees that reverse retiming finds a feasible retiming of G^m if such a retiming exists. Hence, reverse retiming on G^m finds a feasible retiming of G which satisfies the additional constraints of the form $L(v_b) - L(h) \leq i-1$. Moreover, any feasible retiming of G which satisfies the additional constraints is a feasible retiming of G^m. By Theorem 1, such a retiming has a maximum normalized lag that is not less than the maximum normalized lag found by reverse retiming on the graph G^m. The claim follows. □

An iterative method of setting constraints of the form $v_b \leq i-1$ is summarized in Algorithm 3. The method retimes a circuit C (with G being its companion circuit graph) for a desired cycle time c, with an equivalent initial state.

V. Circuit Characterization for Retiming

Recall, the maximum normalized lag (L^*) for a given retiming of a circuit is the maximum difference between the lag of a vertex and the lag of the host. The minimum maximum normalized lag ($L^*_{\min}$) is defined as the smallest L^* of all feasible retimings of the circuit. This is a circuit property, which can be used to classify the circuits. Another property is the reachability of the initial state of the circuit's state diagram from another state by a sequence of L^* transitions.

We classify all circuits into four classes, according to $L^*_{\min}$ and the reachability of the initial state by a sequence of L^* transitions.

I) $L^*_{\min} \leq 0$ and initial state reachable.
II) $L^*_{\min} \leq 0$ and initial state unreachable.
III) $L^*_{\min} > 0$ and initial state reachable. For example, the circuit in Fig. 2 does have a reachable initial state and has a positive $L^*_{\min}$.
IV) $L^*_{\min} > 0$ and initial state unreachable. For example, the circuit in Fig. 3 does not have a reachable initial state and has a positive $L^*_{\min}$.

This circuit classification will be used to discuss the applicability of FEAS and reverse retiming. It will also be used to compare initial state computation of [11] with the one presented in this paper.

It is obvious that reverse retiming should always be used for class I and II circuits. By theorem 1, reverse retiming will find the retiming with $L^*_{\min} \leq 0$. Initialization is trivial and can be done by forward simulation. In the following, we argue that reverse retiming is also beneficial for the initialization of class III and IV circuits.

The maximum normalized lag found by reverse retiming is never greater then the maximum normalized lag found by FEAS. The length of the input sequence needed by [11] equals the maximum normalized lag. Hence, reverse retiming

can help in shortening the required input sequence. Finding a longer sequence requires partial state enumeration of a larger set of states, which, in turn requires a larger effort. In [11], the length of the sequence also implies the amount of registers to which reset logic needs to be added. Since a sequence of state transitions is searched with a minimal Hamming distance, adding a transition to this sequence will increase or keep equal (but never decrease) the number of state bits that need modification. Therefore, there are never more registers that need modification when reverse retiming is used. This shows that reverse retiming is advantageous for the initial state computation of [11].

Reverse retiming produces a result for which the justification process in *Update_Registers* is more likely to succeed, since less calls to the justification procedure have to be done. Note that for circuits, which have $L^*_{\min} > 0$ and for which *Update_Registers* fails on the retiming produced by reverse retiming, another retiming may exist on which *Update_Registers* succeeds. It may very well be that this is the retiming accidentally produced by FEAS [7]. However, in the iterative method described in Algorithm 3, it is guaranteed that the reverse retiming method (eventually) finds this solution as well.

As far as the the comparison between [11] and *Update_Registers* is concerned, only class III and IV circuits have to be discussed. In class III circuits, both initialization methods produce identical results (except maybe for some initialization values) when they complete. The *Update_Registers* algorithm does not require the partial implicit enumeration of the reachable states, a process which may be infeasible for many designs (for examples see the next section). In class IV circuits, the method in [11] incurs the danger of adding reset logic which might deteriorate both area and delay (as shown in the example of Fig. 4) and violate the minimal cycle time again. In case our justification fails, our method will bound the lags of the specific blocks and attempt to find another retiming with a reachable initial state.

There might still be cases where no retiming exists for a required cycle time c, for which an equivalent initial state can be found without modifying the circuit. In these cases, the required cycle time has to be increased or circuit modifications allowed. In the following section this discussion is justified with some results.

VI. Experimental Results

The retiming method as described in the previous sections is implemented within the BooleDozer [1] logic synthesis system. We applied the reverse_retiming algorithm to 33 sequential multilevel circuits in the MCNC benchmark set [2]. The EDIF files from the MCNC (LGSynth93) benchmark directory have been used as the input for our experiments. A unit delay model is assumed. Each gate has a propagation delay of one unit, no delays on the registers and no delays due to fan-in or fan-out were used. The minimal cycle time for these circuits is found by running FEAS (or reverse retiming) and doing a binary search on the cycle time. The initial state of all registers equal to zero is assumed. For all examples, both FEAS and reverse retiming (since they have the same complexity and a similar implementation) run in under 30 s on an IBM RS6000, model 390. For fourteen of those circuits listed in Table I, retiming could not improve their cycle time. For the remaining 19 designs, retiming reduced their cycle time up to 40%.

TABLE I
Retiming Did Not Improve Cycle Time

Circuit	Cycle Time
s27	4
s208	4
s298	4
s386	4
s420	4
s510	4
s641	75
s713	84
s820	4
s832	4
s1196	32
s1488	4
s1494	4
s35932	19

TABLE II
No Difference Between FEAS and Reverse Retiming

Circuit	*CT*	*RCT*	*L**	*Runtime*	*InitSeq*	*Class*
s208.1	15	14	0	1	1	I
s420.1	17	16	0	1	1	I
s838	93	92	0	2	1	I
s838.1	21	20	0	3	1	I
s5378	32	27	0	14	NA	I/II
s9234.1	55	51	0	22	NA	I/II
s1238	31	30	1	1	NR	IV
s1423	66	59	1	2	NR	IV
s15850	47	26	1	12	NA	III/IV
s38584.1	69	61	1	29	NA	III/IV

CT: cycle time, *RCT* : retimed cycle time
L^*: maximum normalized lag, *Runtime* : run time in seconds

Our main interest in these experiments is the comparison between the retimed circuit obtained by the FEAS algorithm [7] versus the one obtained by the reverse_retiming algorithm. Notice that the minimal feasible cycle time achieved by both algorithms is the *same*, and the difference is the retiming function which reflects the feasibility to find an equivalent initial state for the retimed design. For ten designs listed in Table II, the maximal value of the normalized lag produced by both algorithms was the same being 0 or +1. For four circuits (s1238, s1423, s15850, and s38584.1) reverse retiming was not able to find a solution with L^*=0, because it simply does not exist. In most cases, this is due to a path from a primary input to a register which is too long to meet the optimal cycle time. The only way to solve this is moving registers forward through the O–I's or moving registers backward.

For the remaining nine circuits listed in Table III, a difference is found between reverse retiming and FEAS. The

TABLE III
IMPROVEMENT FROM REVERSE RETIMING OVER FEAS

	Original			FEAS		Reverse Retiming				
Circuit	*CT*	*Reg*	*RCT*	L^*	*Reg*	L^*	*Reg*	*Runtime*	*InitSeq*	*Class*
s344	28	15	19	1	21	0	27	1	NR	II
s349	28	15	19	1	21	0	28	1	NR	II
s382	17	21	12	1	39	0	33	1	NR	II
s400	17	21	12	1	41	0	34	1	NR	II
s444	20	21	13	1	44	0	35	1	NR	II
s526	14	21	11	1	33	0	42	1	45	I
s526n	14	21	11	1	28	0	33	1	45	I
s953	27	29	23	1	39	0	33	2	8	I
s38417	65	1465	49	1	1477	0	1504	71	NA	I/II

CT: cycle time, *Reg*: number of registers
RCT: retimed cycle time, L^*: maximum normalized lag
Runtime: run time in seconds

retiming function generated by FEAS has maximum normalized lag value of +1. The better retiming function achieved by reverse retiming has a maximum normalized lag of 0. Only forward implication moves are necessary to find the equivalent initial state.

Using the technique of [11], finding an initial state requires state enumeration and may require the addition of logic, which affects both area and delay. To evaluate the properties of the benchmark machines, we did a breadth-first state enumeration [3] starting from the initial state until the first time the initial state was reached again. The length of this sequence is recorded in the column $InitSeq$ in Tables II and III. If the initial state is nonreachable from any other state, an NR is entered in the column. If partial state machine enumeration was infeasible within 10 h run time (on a RS6000/390, 134 mips, 256MB) a NA (Not Available) appears in the table.

For the four circuits in Table II with $L^* = 1$, s1238 and s1423 need modifications (NR), and s15850 and s38584.1 (NA) may be intractable with the initial state computation of [11]. The initial state calculation described in this paper found equivalent initial states without extra logic in all four cases in less than one minute run time.

From Table III, we conclude that if FEAS retiming is done and the method of [11] is used for initialization, five circuits require addition of logic (NR). Three circuits can be initialized since the length of the sequence is greater than the maximum lag (InitSeq $= 8(or 45) > L^* = 1$) and circuit s38417 (NA) is likely not to be solvable since the partial state enumeration is highly complex.

VII. CONCLUSION

The new reverse retiming algorithm finds a retiming for a given cycle time which requires only forward moves if such a retiming exists. Otherwise, it calculates a retiming which is beneficial to the method of initial state computation described in [11]. Reverse retiming is a variation of the retiming algorithm FEAS [7] and therefore has the same complexity.

An iterative new method to update the network and find an equivalent initial state is described. This method does not require the (partial) implicit enumeration of the state machine, but is based on an efficient justification procedure applied to only small portions of the network. When successful, the new method does not require the addition of logic, which may deteriorate both area and delay of the retimed circuit. In all MCNC benchmark circuits, the new initial state calculation produced results in very short running times while earlier approaches in certain instances failed to complete the initial state computation.

Reverse retiming minimizes the maximum normalized lag. Although this is only a heuristic measure for the amount of work to be done in the justification procedure of the initial state calculation step, our results show that this measure works very well in practice.

The observations made in this paper can be used in other formulations of retiming problems. For example, the linear program formulation of the register minimization problem [7] during retiming can be extended to include the initial state preservation constraints.

APPENDIX: PROOF OF THEOREM 1

This appendix contains a sketch of the proof for the reverse retiming Theorem 1. For completeness the theorem is restated here.

Theorem 1: Let $G = (V, E, w, d)$ be a circuit and c a required clock cycle. Let $L(v), v \in V$ denote the retiming computed by the algorithm reverse_retiming(G, c). Then

1) The algorithm reverse_retiming(G, c) finds a retiming L such that $\Phi(G_L) \leq c$, if such a retiming exists.
2) If all nodes are reachable from the host and if $\Phi(G_L) \leq c$, then for every retiming L' for which $\Phi(G'_L) \leq c$ the following holds

$$\max_{v \in V} L(v) - L(h) \leq \max_{v \in V} L'(v) - L'(h).$$

Proof: The proof of the theorem is based on the observation that reverse retiming can be implemented by the FEAS algorithm on the reversed graph. The FEAS [7] algorithm can be described as follows. Define the *valid-time*, $t_v(u)$, of a node

u to be the maximum delay of a registerless path ending in u. Formally

$$t_v(u) = \begin{cases} d(u) & \text{if } u \text{ is preceeded only by registers or primary inputs} \\ \max_{(x,u)\in E} t_v(x) + d(u) & \text{otherwise.} \end{cases}$$

Algorithm FEAS, described in Algorithm 4 uses valid-times contrary to reverse retiming which uses required times.

```
Algorithm 4. FEAS(G,c)}
foreach v ∈ V do r(v) = 0
k = 1
do
 Compute G_r.
 Compute t_v(u) for every vertex u.
 M = {u : t_v(u) > c}.
 Foreach u ∈ M do r(u) = r(u) + 1.
 k = k + 1
while k < |V| and M ≠ ∅.
```

One can compute the retiming found by the reverse_retiming algorithm as follows. a) Reverse the directions of the edges. b) Execute the FEAS algorithm on the reversed graph with the same required clock period. c) Set $L(u) = -r(u)$ for every node u. Negation of lags induces a one-to-one correspondence between retimings of a graph and its reversed graph. Moreover, the delay and weight of paths are invariant under reversal of edges. Since FEAS is guaranteed to find a feasible retiming if one exists [7], then part 1 of the theorem follows.

The proof of part 2 is more evolved. Leiserson and Saxe shortcut *remote* pairs of nodes with *constraint edges*. Vertex v is considered to be remote from u if there exists a shortest weight path from u to v whose delay is greater than the required clock period c. A constraint edge (u, v) is added in such a case, and is assigned weight $W(u, v) - 1$, where $W(u, v)$ equals the weight of a shortest weight path from u to v. We call the graph augmented with the constraint edges the *constraints graph*.

The crux of the proof lies in the claim that, if algorithm $FEAS$ finds a feasible retiming, then, for every node, $r(v)$ satisfies:

$$r(v) = \max\{-w_c(p) : p \text{ is a path ending in } v \text{ in the constraints graph}\} \quad (1)$$

where $w_c(p)$ denotes the weight of path p in the constraints graph.

The proof of (1) is divided into two parts. First we show that $r(v) \geq -w_c(p)$ for every path p ending in v in the constraints graph. We proceed by showing that there exists a path p ending in v in the constraints graph for which $r(v) = -w_c(p)$.

Suppose that there exists a path, p, in the circuit ending in v which contains k disjoint segments whose delay exceeds c and which contains ℓ registers. In such a case, since FEAS only increments lags, it follows that, in order to add $k - \ell$ registers along p, algorithm FEAS must lag node v at least $k - \ell$ times. The addition of constraint edges introduces a mechanism for counting segments whose delay exceeds c. The weight of a path in the constraints graph equals the weight of a corresponding path in the circuit minus the number of disjoint segments along it whose delay exceeds c. This proves that $r(v) \geq -w_c(p)$, for every path p which ends in v in the constraints graph.

We prove that, for every node v, there exists a path p ending in v in the constraints graph for which $r(v) = -w_c(p)$ by induction on the iterations of FEAS as follows. At the beginning, each node has zero lag, and the path which consists of the node itself has zero weight, therefore, the induction basis follows. Consider a node whose lag is incremented in the ith iteration. Suppose that p is a path from u to v in the circuit which causes the lag of v to be incremented. Let $w_i(p)$ denote the weight of p at the beginning of the ith iteration, and $r_i(v)$ denote the lag of v at the beginning of the ith iteration. By the definition of p, it follows that $w_i(p) = 0$ and $d(p) > c$. By the induction's hypothesis, there exists a path p' that ends in u in the constraints graph for which $r_i(u) = -w_c(p')$. Since $w_i(p) = w(p) - r_i(u) + r_i(v)$, and since $w_i(p) = 0$, it follows that $r_i(u) = w(p) + r_i(v)$. Consider the path p'' obtained by augmenting the path p' with the constraint edge (u, v)

$$\begin{aligned} w_c(p'') &= w_c(p') + w_c(u, v) \\ &= -r_i(u) + w(p) - 1 \\ &= -r_i(v) - 1 \\ &= -r_{i+1}(v) \end{aligned}$$

and the induction step follows.

The outcome, L of the reverse_retiming algorithm equals $-r$, and hence, (1) can be reformulated by

$$L(v) = \min\{w_c(p) : p \text{ is a path starting in } v \text{ in the constraints graph}\} \quad (2)$$

Note, that the FEAS algorithm is invoked on the reversed constraints graph, and therefore, we now consider paths that start in v rather than paths that end in v.

Let p denote a path in the constraints graph that starts in the host and ends in node v_0 for which $L(host) = w_c(p)$. Since L' is a feasible retiming, every edge in the constraint graph satisfies $w_c(u, v) - L'(u) + L'(v_0) \geq 0$. Consider the edges of path p, and sum up these inequalities to obtain $w_c(p) - L'(host) + L'(v_0) \geq 0$.

Hence,

$$L'(v_0) - L'(host) \geq -w_c(p) = -L(host).$$

However,

$$\max_v L'(v) - L'(host) \geq L'(v_0) - L'(host)$$

and since $L(v) \leq 0$ it follows that

$$-L(host) \geq \max_v L(v) - L(host).$$

We proved that

$$\max_v L'(v) - L'(host) \geq \max_v L(v) - L(host)$$

and the theorem follows. □

ACKNOWLEDGMENT

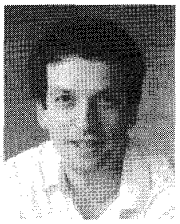
The authors wish to thank A. Mets for providing us with the tool for state enumeration and helping us with the experiments on the state machines. The authors also wish to acknowledge our reviewers for their constructive remarks.

REFERENCES

[1] D. Brand, R. Damiano, L. van Ginneken, and A. Drumm, "In the driver's seat of BooleDozer," in *Proc. IEEE Int. Conf. Computer Design*, Oct. 1994, pp. 518–521.
[2] F. Brglez, D. Bryan, and K. Kozminski, "Combinational profile of sequential benchmark circuits," in *Proc. Int. Symp. Circuits Syst.*, May 1989, pp. 1929–1934.
[3] J. Burch, E. Clarke, D. Long, K. McMillan, and D. L. Dill, "Symbolic model checking for sequential circuit verification," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 401–424, Apr. 1994.
[4] S. Dey, M. Potkonjak, and S.G. Rotweiler, "Performance optimization of sequential circuits by eliminating retiming bottlenecks," in *Tech. Dig. Papers Int. Conf. Computer-Aided Design*, Santa Clara, CA, Nov. 1992, pp. 504–509.
[5] S. Kundu, L. Huisman, I. Nair, V. Iyengar, and L. Reddy, "A small test generator for large designs," in *Proc. Int. Test Conf.*, Sept. 1992, pp. 30–40.
[6] C. Leiserson and J. Saxe, "Optimizing synchronous systems," *J. VLSI Computer Syst.*, vol. 1, no. 1, pp. 41–67, 1983.
[7] ——, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, no. 1, pp. 5–35, 1991.
[8] S. Malik, E. Sentovich, R. Brayton, and A. Sangiovanni-Vincentelli, "Retiming and resynthesis: Optimizing sequential networks with combinational techniques," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 74–84, Jan. 1991.
[9] G. D. Micheli, "Synchronous logic synthesis: Algorithms for cycle-time minimization," *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 63–73, Jan. 1991.
[10] V. Singhal, C. Pixley, R. Rudell, and R. Brayton, "The validity of retiming sequential circuits," in *Proc. 32nd Design Automation Conf.*, San Fransisco, CA, June 12–16, 1995, pp. 316–321.
[11] H. Touati and R. Brayton, "Computing the initial states of retimed circuits," *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 157–162, Jan. 1993.

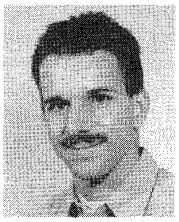
Guy Even (S'92–M'95) received the B.Sc. degree in mathematics and computer science from the Hebrew University, Jerusalem, Israel, in 1988, and the M.Sc. and D.Sc. degrees in computer science from the Technion, Haifa, Israel, in 1991 and 1994, respectively. Currently, he is a post-Doctoral student under Professor W. Paul, University of the Saarland, Saarbruecken, Germany.

The results reported in this paper are part of his Doctoral dissertation that dealt with the design of VLSI circuits using retiming. His current areas of research deal with approximation algorithms for NP-complete problems on graphs, design of systolic arrays, and computer architecture. His homepage URL is: http://www-wjp.cs.uni-sb.de/~guy.



Ilan Y. Spillinger (S'85–M'88–SM'95) received the B.Sc. degree in computer engineering, and the M.Sc. and D.Sc. degrees in electrical engineering from the Technion, Haifa, Israel, in 1982, 1984, and 1987, respectively.

From 1987 to 1992 he was with the Israeli Defense Forces, and the adjunct faculty of the Department of Electrical Engineering, Technion, Israel. From 1992 to 1995 he was with the Logic Synthesis Group of the IBM T. J. Watson Research Center, Yorktown Heights, NY. He is currently with the Architecture Research Group at the Intel Design Center in Haifa, Israel.



Leon Stok (S'88–M'91–SM'95) received the M.Sc. degree (with honors) and the Ph.D. degree in electrical engineering from Eindhoven University, The Netherlands, in 1986 and 1991, respectively.

He is the Manager of the Logic Synthesis Group with the IBM T. J. Watson Research Center, Yorktown Heights, NY, where he has been a Research Staff Member since 1991. Prior to this, he was with the Unternehmensbereich Kommunikations-und Datentechnik of Siemens AG, Munich, Germany, in 1985, and with the Mathematical Sciences Department of the IBM T. J. Watson Research Center during 1989 and 1990. He has published several papers on the various aspects of logic, high level and architectural synthesis and on the automatic placement and routing for schematic diagrams.