Optimal Retiming for Initial State Computation

Peichen Pan*
Strategic CAD Labs
Intel Corporation
Hillsboro, OR 97124
peichen@ichips.intel.com

Guohua Chen
Department of ECE
Clarkson University
Potsdam, NY 13699
cheng@sun.soe.clarkson.edu

Abstract

Retiming is a transformation that optimizes a sequential circuit by relocating the registers. When the circuit has an initial state, one must compute an equivalent initial state for the retimed circuit. In this paper we propose a new efficient retiming algorithm for performance optimization. The retiming determined by the algorithm is the best with respect to initial state computation. It is the easiest retiming for finding an equivalent initial state, and if logic modification is required, it incurs the minimum amount of modification.

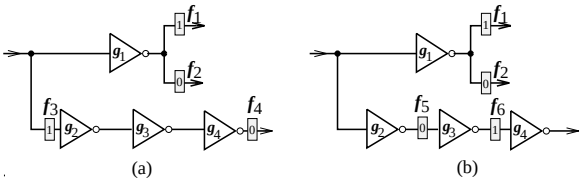

Figure 1. Retiming: (a) original circuit, (b) retimed circuit.

1 Introduction

Retiming is a transformation that relocates the registers in a (sequential) circuit while preserving functionality [1, 2]. Retiming has been used to optimize performance, area, and power [1, 3, 4]. It has also been combined with other ways of design optimization [5, 6, 7]. Retiming for level-sensitive latches has also been addressed [8, 9].

Fig. 1 shows an example of retiming. The circuit in Fig. 1(b) is obtained from the one in (a) by retiming gates g_2 and g_4. For g_2, the register at its input is moved *forward* to its output; for g_4 the register at its output is moved *backward* to its input. Assuming each gate has one unit of delay, the retimed circuit has a cycle time equal to one as opposed to three in the original circuit. The *cycle time* of a circuit is the maximum combinational path delay in the circuit.

A retiming of a circuit can be represented by a mapping r from the nodes (gates, PIs or POs) to integers, where $r(v)$ denotes the number of registers moved from each output of node v to each input of v. For the example in Fig. 1, $r(g_4) = 1$ while $r(g_2) = -1$. For all other nodes, the values are zero. In general, when $r(v)$ is positive, registers are moved backward across v, and registers are moved forward across v when $r(v)$ is negative. $r(v)$ is also called the *lag* at v.

The *initial state* of a circuit consists of the initial values of the registers in the circuit. When the initial state is an integral part of the behavior, it is necessary to find an equivalent initial state for the retimed circuit. An *equivalent initial state* of a retimed circuit is a state such that for any given input sequence, the circuit and the retimed one produce the same output sequence if both circuits start from their respective initial states. For the example in Fig. 1, the initial states are specified by the values in the registers and they are equivalent.

For forward retiming across a node, the initial values in the registers can be propagated to the new registers by evaluating the function of the node using the initial values, i.e., forward logic simulation. For the example in Fig. 1, when f_3 is moved forward across the inverter g_2 to f_5, the initial value of f_3 is inverted and the resulting value is assigned to f_5 as its initial value. On the other hand, for backward retiming across a node, determining the initial values for the new registers requires backward justification. Backward justification is NP-hard. Moreover, a solution may not exist. If this occurs, circuit modification is required to ensure the existence of an equivalent initial state. As an example, Fig. 2(a) shows another retimed circuit of the one in Fig. 1(a). This retimed circuit also has a cycle time one, but it cannot be initialized to have the same behavior as the

*This work was done while this author was with ECE Dept., Clarkson University, Potsdam, NY. His research was partially supported by a grant from Intel Corporation.

original circuit. This is because f_{12} is a result of moving both f_1 and f_2 backward, but their initial values cannot be both propagated to f_{12}. Fig. 2(b) is a modified circuit with an equivalent state. Here a reset AND gate is added to force the correct value at one of the POs in the first clock cycle. Note that the cycle time of the modified circuit is increased back to two again, in addition to the extra gate and register.

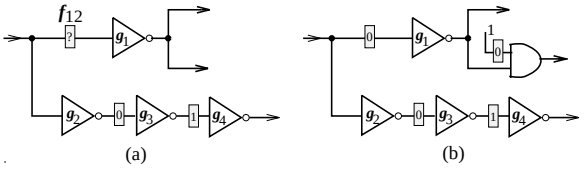

Figure 2. Maintaining initial state with circuit modification.

A method for computing an equivalent initial state when given a retiming was proposed in [10]. Suppose k is the maximum lag of the retiming. The method first extracts the state machine of the original circuit to find a sequence of k transitions that can lead to the initial state. If such a sequence exists, the method finds an equivalent initial state for the retimed circuit by a sequence of forward logic simulation. If there is no such sequence, the circuit is modified to ensure such a sequence exists. Existence of such sequence is only a sufficient condition. For the example in Fig. 1, the maximum lag is one. It is also obvious that the initial state cannot be reached from another state since registers f_1 and f_2 always have the same value except at the very beginning. The retimed circuit, however, does have an equivalent initial state as shown in Fig. 1(b), without circuit modification.

Recently, a retiming algorithm called *reverse retiming* was proposed that tries to reduce the chance of circuit modification by finding a retiming with minimum maximum lag [11]. As we just showed, the maximum lag of a retiming does not correlate well with the existence of an equivalent initial state. In fact, reverse retiming may find a retiming that does not admit an equivalent state even when such a retiming exists. For example, for the circuit in Fig. 1(a) with the target cycle time equal to one, reverse retiming produces exactly the retimed circuit in Fig. 2, which, as mentioned earlier, does not admit an equivalent initial state.

In this paper, we propose an efficient new retiming algorithm. The algorithm finds a retiming such that the retimed circuit meets a given cycle time, if such a retiming exists. In addition, the retiming has minimum lag at every node. As will be shown in Section 2, such a retiming is the *best* with respect to initial state computation. It is the easiest retiming for finding an equivalent initial state. It has minimum chance of requiring logic modification, and if logic modification cannot be avoided, it requires the minimum amount of logic modification. We point out that the authors of [11] also noticed the drawbacks of their algorithm and proposed an iterative retiming method to selectively bound individual lags, when the retiming obtained by their algorithm does not admit an equivalent initial state. The method is enumerative and has exponential time cost. One the other hand, the algorithm proposed in this paper is polynomial and has the same complexity as the best known retiming algorithm FEAS [1].

Finally, we introduce a few notations. We represent a synchronous sequential circuit as a directed graph. Each node in the graph represents either a primary input (PI), primary output (PO) or a gate, and each edge $u \xrightarrow{e} v$ represents an interconnection from node u to node v. An edge e has a weight, $w(e)$, which is the number of registers on the interconnection. Each node v has a delay $d(v)$. The circuit obtained after applying retiming r to a circuit N will be denoted by N_r. The number of registers on edge $u \xrightarrow{e} v$ in N_r, $w_r(e) = w(e) + r(v) - r(u)$. The rest of the paper is organized as follows: In Section 2, we present a method to determine the new locations of a register after retiming. Our algorithm for finding a retiming that meets a given cycle time and has minimum lags is given in Section 3. We present our experimental results in Section 4 and conclude the paper in Section 5.

2 Determining register locations after retiming

In this section, we present a method to determine the new locations of a register after retiming. From this method, we will see why a retiming with minimum lag at every node is the best retiming for initial state computation.

Let f be a register on edge $v_0 \xrightarrow{e_0} v$ (see Fig. 3). Let W_0 denote the number of registers to the left of f on e_0, including f. If $r(v_0) \geq W_0$, then f is moved backward out of e_0 by the retiming since $r(v_0)$ registers are moved backward across v_0. Similarly, if $r(v) < -w(e_0) + W_0$, f is moved forward out of e_0 by the retiming since there are $w(e_0) - W_0$ registers to the right of f on e_0, and $-r(v)$ registers are moved forward across v by the retiming. Note that only one of these cases can happen since $w_r(e_0) = w(e_0) + r(v) - r(v_0)$ must be non-negative. If neither case happens, f stays on e_0 in N_r.

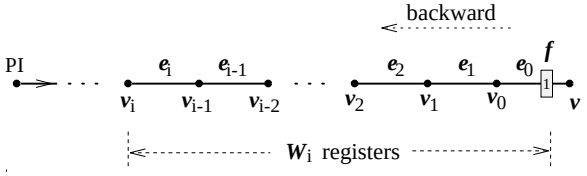

Figure 3. Retiming a register along a path.

Since forward retiming moves are not a concern in initial state computation, we only consider the case that f is moved backward out of e_0, i.e., $r(v_0) \geq W_0$. Consider a path p extending out of v_0 to a PI as shown in Fig. 3. Let W_i be the number of registers on the path from v_i to f, including f, namely, $W_1 = W_0 + w(e_1)$, $W_2 = W_1 + w(e_2)$, $\cdots$, $W_i = W_{i-1} + w(e_i)$, $\cdots$. If $r(v_i) \geq W_i$ then f is moved backward out of v_i. Let t be the minimum integer i such that $r(v_i) < W_i$. Then, f is moved backward out of v_{t-1}, but not v_t. Hence in the retimed circuit, f is right on edge e_t. Since the lag of each PI is zero, such a t always exists.

Now suppose r' is another retiming of N such that $r(u) \leq r'(u)$ for every node u in the path p. Then, $r(v_i) \geq W_i$ implies $r'(v_i) \geq W_i$ for any v_i in p. In other words, if f is moved backward across v_i in N_r, it is also in $N_{r'}$. Thus, we have the following result:

Theorem 1 *Let r and r' be two retimings of N such that $r(v) \leq r'(v)$ for every node v in N, then r moves every register backward along any path by a smaller or equal distance than r'.*

As mentioned before, propagating the initial value of f that is retimed backward to the registers at its new locations is a backward justification problem. The logic used in the justification process consists of all the nodes between f and its new locations. We will refer this logic as the *justification cone* of f. The edges going into the cone are the new locations of f in the retimed circuit, and the edge coming out of the cone is where f is, in the original circuit.

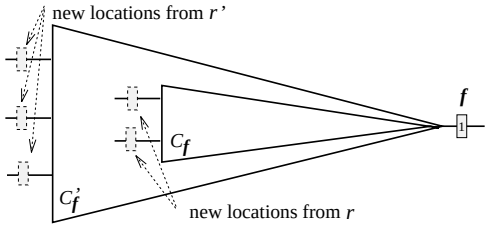

Figure 4. Relation between justification cones.

Let r and r' be two retimings of N such that $r(v) \leq r'(v)$ for each node v in N. From Theorem 1, the justification cone of any register f from r, C_f, *is contained* by the justification cone of f from r', C'_f, as indicated in Fig. 4. On the way to propagate the initial value of f to the edges going into C'_f, the edges going into C_f are also assigned values. Therefore, if the initial value of f can be propagated to the registers in the new locations for r' without logic modification, it is also the case for r. Suppose logic modification is required and C'_f is modified by the standard technique of adding reset logic [10, 12]. Since C_f is a subset of C'_f, if C_f is modified the same way as in C'_f, the initial value of f is also propagated to the new locations from r. As a result, the cost of modification required by r is always smaller than or equal to that required by r', and if no modification is required for r', then it is not needed for r either. Thus, r is a better retiming than r' as far as equivalent initial state computation is concerned.

3 Retiming for minimum lags

As the discussions in the last section conclude, the smaller the lags of a retiming are, the easier the backward justification problem will be. In this section, we present a retiming algorithm that produces a retiming with minimum lag at every node while achieving a target cycle time ϕ. We will refer to such retiming as the *min-lag retiming* (for the given cycle time ϕ).

We first examine existing retiming algorithms to see why they fail to minimize the lags. The best known retiming algorithm FEAS is iterative [1]. At each iteration, it computes the maximum combinational path delay ending at each node. Then it moves registers backward across each node with a path delay exceeding the target cycle time. The problem with FEAS is that some of backward retiming moves are actually not necessary. For example, for the simple circuit in Fig. 5, to achieve a cycle time of one, FEAS moves f_2 backward across g_2 since the combinational path delay ending at g_2 is two, exceeding the target cycle time. However, the move is not necessary since the path delay at g_2 can also be reduced by moving f_1 forward across g_1.

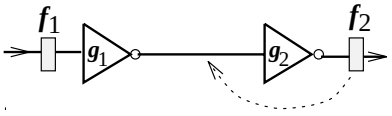

Figure 5. Unnecessary backward retiming in FEAS.

On the other hand, the reverse retiming algorithm proposed recently only move registers forward [11]. Backward retiming is done by moving registers forward from the POs to the PIs in a wrap-around fashion. In reverse retiming, registers are moved forward only when path delays exceed the target cycle time. This may cause problem when registers are moved forward from the POs to the PIs. Moving registers forward from the POs to the PIs has the same effect of moving registers at the POs all the way backward to the PIs. Reverse retiming tries to leave those registers moved to the PIs right there. This can be illustrated by considering the circuit in Fig. 1(a). For this circuit, to realize a cycle time of one, reverse retiming first moves f_3 forward across

g_2 and g_3. Then, it moves f_1, f_2 and f_4 at the POs to the PI. The register at the PI will be moved further forward on the path containing g_2 to reduce the path delay. However, on the paths containing g_1 the register at the PI will not be moved forward as the path delays already meet the cycle time. This results in exactly the circuit in Fig. 2. The actual effect is that f_1 and f_2 are moved backward to f_{12} and the resulting circuit cannot be initialized.

We now describe the details of our algorithm. Let L_v denote the minimum number of registers on the paths from the PIs to the node v. The path delays (or arrival times) of the nodes in the circuit are defined as follows:

$$a(v) = \begin{cases} 0, & v \text{ is a PI or register output} \\ d(v) + \max_{u \xrightarrow{e} v, w(e)=0} a(u), & \text{otherwise,} \end{cases}$$

here we assume a dummy node is introduced at the output of each register.

Our algorithm will be referred to as MINLAG. MINLAG maintains a *label* at each node, which intuitively is a lower bound on the minimum lag at the node. It then iteratively improves the labels until they reach the respective minimum lags. An outline of MINLAG is given in Fig. 6, where the label of node v is $l(v)$. Note that l is NOT a valid retiming in that it may assign a non-zero value to a PO. Nevertheless, we treat l as it were a retiming and use it to re-weight the circuit. Using the same notation for retiming, we denote the circuit "retimed" according to l by N_l.

MINLAG initializes the label of each node v to $-L_v$. It then increments the labels at those nodes whose arrival times in N_l exceed the target cycle time ϕ. This process is repeated until all arrival times are within the target cycle time, at which time we generate the min-lag retiming from the labels. If the number of iterations exceeds the number of nodes in the circuit, the target cycle time is not achievable. MINLAG also stops and reports FAILURE if there is a PO with a label larger than 0.

```
MINLAG(N, φ)
  foreach node v in N do
    l(v) = −L_v;
  i = 0;
  do
    i = i + 1;
    Calculate the arrival time a(v) for each node v in N_l;
    if (∀v a(v) ≤ φ) then return SUCCESS;
    foreach v such that a(v) > φ do
      l(v) = l(v) + 1;
      if (v is a PO and l(v) > 0) then return FAILURE;
  while (i is less than the number of nodes in N)
  return FAILURE;
```

Figure 6. Computing the labels.

One way to understand MINLAG is that it separates forward retiming from backward retiming. The initial values of the labels represent a forward retiming (note that $-L_v \leq 0$ for every node v), and this is the only forward retiming ever performed by MINLAG. After that, only backward retiming is performed as MINLAG only increments labels. The initial forward retiming ensures all later backward retiming moves are necessary ones.

If MINLAG stops with SUCCESS, we derive a retiming r from the final labels as follows: $r(v) = l(v)$ for each internal node v, and if v is a PO or PI, $r(v)$ is zero. The correctness of MINLAG is summarized in the following result:

Theorem 2 **(i)** *If* MINLAG *returns SUCCESS, N_r has a cycle time of ϕ.*

(ii) *If* MINLAG *returns FAILURE, then N cannot be retimed to a cycle time of ϕ.*

(iii) *If* MINLAG *returns SUCCESS, then for any retiming r' such that $N_{r'}$ has a cycle time of ϕ, $r(v) \leq r'(v)$ for each v.*

Statements (i) and (ii) guarantee that MINLAG determines a retiming that meets the cycle time ϕ whenever such a retiming exists. Statement (iii) says that if there is a retiming that meets the cycle time ϕ, MINLAG will find the one with minimum lag at every node, the best retiming for initial state computation. The proof of Theorem 2 is omitted here due to space limitation.

Let n and m denote the number of nodes and the number of edges in N, respectively. We now estimate the time complexity of MINLAG. We first note that all L_v can be determined by a single-source shortest path algorithm with edge weight $w(e)$. Since $w(e) \geq 0$ for every e, Dijkstra's shortest path algorithm can be used [13]. Thus, all L_v can be determined in $O(m+n\log n)$ time. The **do** loop has at most n iterations. In each iteration the main task is to find the arrival times of the nodes, which is a single source longest path problem in a DAG and can be done in $O(n+m)$ time. Thus, the time complexity of MINLAG is $O(n^2+nm)$, the same as FEAS's. In practice, our algorithm is faster as our experiments show. The main reason is that MINLAG can detect an infeasible target cycle time much faster because of the infeasibility test within the **do** loop.

4 Experimental results

We implemented a program that minimizes the cycle time of a sequential circuit under the condition that a equivalent initial state can be found without logic modification. The core of the program is MINLAG presented in this paper. The program carries out binary search on the target cycle time. For each target cycle time, MINLAG is called to find

the min-lag retiming that meets the target cycle time. If we cannot find an equivalent initial state for the retimed circuit without logic modification, we increase the target cycle time. Backward justification in the program is done by a PODEM-like algorithm used in test pattern generation [14].

We tested the program on all sequential benchmark circuits in ISCAS89 suite (including Addendum93) from MCNC. In the experiment, the unit delay model is assumed. For each circuit, we tried two initial states: one with all initial values of the register being zero (*all-zero*), and the other with all values being one (*all-one*). In this section, we describe our experiments and summarize the results.

Among all the benchmarks (43 of them), 36 circuits have their cycle times reduced by retiming. For each of these circuits, our program is able to achieve the the same optimal cycle time that can be obtained without considering initial states, for both *all-zero* and *all-one* initial states. The results are reported in Table 1. In the table, we list the number of gates, the number of registers and the cycle time of each circuit. For each optimized circuit from our program, we list the cycle time and the number of registers. Also listed in the table are the CPU time of our program in seconds, on an UltraSPARC 2 workstation. The CPU times for finding equivalent initial states are listed under *init*, and the times for searching the optimal cycle time are listed under *min*. As can be seen, our program is very efficient. The CPU times for computing equivalent initial states are extremely small. MINLAG may also contribute to such small CPU times as the backward justification effort is minimum.

For comparison purpose, we also tested the benchmarks on FEAS [1] and the recent reverse retiming algorithm (denoted REVE) [11]. Our main interest in the comparison lies in the minimum cycle time retimings found by these algorithms. Although all retimings achieve the same minimum cycle time, they differ in terms of the feasibility and cost for finding equivalent initial states. As we have shown, the retiming from MINLAG is the best in this regard. To show that FEAS and REVE may miss the best retiming, we count the number of nodes with positive retiming values in each retiming. The results are also listed in Table 1. Note that a positive value at a node means backward retiming is performed at the node. For the table it is evident that for quite a number of circuits, the retiming from FEAS has more positive nodes than that from MINLAG. There are some circuits for which backward retiming can be totally avoided for the minimum cycle time, but FEAS fails to find such retimings.

REVE is able to avoid introducing positive lags whenever they can be avoided completely, as it minimizes the maximum lag. However, when backward retiming is necessary, REVE performed considerably worse than even FEAS. For example, for circuit s9234.1, both MINLAG and FEAS introduce only 5 nodes with positive lags, but the retiming from REVE has 4427 nodes with positive lags (about four-fifth of the total number of nodes in the circuit), and the justification logic has a depth of 38 as opposed to a mere 5 from MINLAG.

The logic synthesis tool SIS [15] has a routine for computing equivalent initial states. This routine is based on the method in [10]. The routine has a very time-consuming step to check whether the initial state can be reached again. This is done through state machine extraction. As a result, it is impractical to apply the routine to even moderate-sized circuits. We were able to run the routine for the benchmark circuits down to s1269 in the table. Among the 15 circuit with backward retiming from FEAS, the routine found equivalent initial states for only two of them: s991 and s1269, for the *all-one* initial states. On the other hand, our program successfully found the equivalent initial states for all benchmark circuits for both *all-zero* and *all-one* initial states.

5 Conclusions

We have proposed a new retiming algorithm. For a given cycle time, the algorithm finds a retiming that meets the cycle time. Moreover, the retiming produced by the algorithm has minimum lag at every node, so it is the best retiming as far as equivalent initial state computation is concerned. The algorithm has the same time complexity as the best retiming algorithm that does not consider initial states.

References

[1] C. E. Leiserson and J. B. Saxe, "Retiming synchronous circuitry," *Algorithmica*, vol. 6, pp. 5–35, 1991.

[2] V. Singhal, C. Pixley, R. Rudell, and R. Brayton, "The validity of retiming sequential circuits," in *ACM/IEEE Design Automation Conf. (DAC)*, pp. 316–321, 1995.

[3] J. Monteiro, S. Devadas, and A. Ghosh, "Retiming sequential circuits for low power," in *Intl. Conf. on Computer-Aided Design (ICCAD)*, pp. 398–402, 1993.

[4] R. B. Deokar and S. S. Sapatnekar, "A fresh look at retiming via clock skew optimization," in *ACM/IEEE Design Automation Conf. (DAC)*, pp. 310–315, 1995.

[5] S. Malik, E. M. Sentovich, and R. Brayton, "Retiming and resynthesis: optimizing sequential networks with combinational techniques," *IEEE Trans. on Computer-Aided Design*, vol. 10, pp. 74–84, 1991.

[6] S. Dey, M. Potkonjak, and S. G. Rothweiler, "Performance optimization of sequential circuits by eliminating retiming bottlenecks," in *Intl. Conf. on Computer-Aided Design (ICCAD)*, pp. 504–509, 1992.

benchmark		original		optimed		CPU time		positive gates		
name	gate	ϕ	reg	ϕ	reg	min	init	minlag	feas	reverse
s208.1	104	11	8	10	9	0.00	0.01	0	0	0
s298	119	9	14	6	22	0.00	0.01	6	12	73
s344	160	20	15	14	23	0.01	0.00	0	13	0
s349	161	20	15	14	23	0.01	0.01	0	13	0
s382	158	9	21	7	24	0.00	0.00	2	17	82
s400	165	9	21	7	24	0.00	0.00	2	17	88
s420.1	218	13	16	12	17	0.00	0.00	0	0	0
s444	181	11	21	7	40	0.00	0.01	9	42	101
s499	152	12	22	11	28	0.01	0.00	0	14	0
s510	211	12	6	11	7	0.01	0.00	0	1	0
s526	193	9	21	6	31	0.01	0.01	6	12	94
s526n	194	9	21	6	31	0.01	0.00	6	12	94
s635	286	127	32	66	51	0.01	0.02	105	105	168
s838.1	446	17	32	16	33	0.01	0.00	0	0	0
s938	446	17	32	16	33	0.00	0.01	0	0	0
s953	395	16	29	13	34	0.02	0.01	0	18	0
s967	394	14	29	12	35	0.02	0.01	4	9	379
s991	519	59	19	54	22	0.01	0.01	3	3	198
s1269	569	35	37	19	94	0.03	0.02	119	119	370
s1423	657	59	74	53	79	0.03	0.01	19	19	465
s1488	653	17	6	16	7	0.01	0.01	0	0	0
s1494	647	17	6	16	7	0.01	0.01	0	0	0
s1512	780	30	57	23	72	0.05	0.00	14	14	652
s3271	1572	28	116	15	188	0.13	0.03	189	189	861
s3330	1789	29	132	14	165	0.06	0.07	42	42	278
s3384	1685	60	183	27	209	0.09	0.05	0	16	0
s4863	2342	58	88	30	207	0.24	0.08	321	359	1393
s5378	2779	25	164	21	192	0.07	0.07	0	0	0
s6669	3080	93	231	26	490	0.31	0.17	507	507	1308
s9234.1	5597	58	211	38	239	0.38	0.09	10	10	4427
prolog	1601	26	85	13	164	0.04	0.05	19	19	381
s13207.1	7951	59	638	51	640	0.27	0.09	13	13	5381
s15850.1	9772	82	534	63	572	0.42	0.11	175	175	7310
s35932	16065	29	1728	27	1729	1.51	1.40	576	576	6768
s38417	22179	47	1636	32	1659	40.87	0.80	0	282	0
s38584.1	19253	56	1426	48	1428	1.39	0.32	8	8	15095

Table 1. Experimental Results.

[7] P. Pan and C. L. Liu, "Optimal clock period FPGA technology mapping for sequential circuits," in *ACM/IEEE Design Automation Conf. (DAC)*, pp. 720–725, 1996.

[8] B. Lockyear and C. Ebeling, "Optimal retiming of multi-phase level-clocked circuits," in *Advanced Research in VLSI*, pp. 265–280, 1992.

[9] A. T. Ishii, C. E. Leiserson, and M. C. Papaefthymiou, "Optimizing two-phase, level-clocked circuitry," in *Advanced Research in VLSI*, pp. 245–264, 1992.

[10] H. Touati and R. Brayton, "Computing the initial states of retimed circuits," *IEEE Trans. on Computer-Aided Design*, vol. 12, pp. 157–162, 1993.

[11] G. Even, I. Spillinger, and L. Stok, "Retiming revisited and reversed," *IEEE Trans. on Computer-Aided Design*, vol. 15, pp. 348–357, 1996.

[12] V. Singhal, S. Malik, and R. Brayton, "The case for retiming with explicit reset circuitry," in *Intl. Conf. on Computer-Aided Design (ICCAD)*, pp. 618–625, 1996.

[13] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. New York: McGraw-Hill Book Company, 1990.

[14] M. Abramovici, M. A. Breuer, and A. D. Friedman, *Digital Systems Testing and Testable Design*. New York: IEEE Press, 1990.

[15] E. M. Sentovich *et al.*, "Sequential circuit design using synthesis and optimization," in *Proc. Intl. Conf. on Computer Design*, pp. 328–333, 1992.