(Proceedings of Asia South Pacific Design Automation Conference 2004)

Preserving Synchronizing Sequences of Sequential Circuits After Retiming

Maher N. Mneimneh Karem A. Sakallah John Moondanos
University of Michigan Intel Corporation
{maherm,karem}@umich.edu john.moondanos@intel.com

Abstract – We propose a novel approach to preserve the synchronizing sequences of a circuit after retiming. The significance of this problem stems from the necessity of maintaining correct initialization of circuits after retiming optimizations. It has been previously shown that forward retiming moves across fanout stems can destroy a synchronizing sequence. We build on this observation and introduce the notion of "invalid states" that might arise due to forward moves. We show that the set of synchronizing sequences of a given circuit can be preserved by modifying transitions from those invalid states. We present an algorithm that implicitly computes the set of invalid states. Then, we describe a post-retiming synthesis step that incrementally resynthesizes some next-state functions to alter the behavior of invalid states to ensure correct post-retiming initialization. We report promising experimental results on the ISCAS 89 benchmarks and on a set of retimed circuits from an Intel Pentium-III class microprocessor.

I. Introduction

Retiming [5] is a transformation applied to a sequential circuit to improve its performance. Retiming has been rigorously studied in the past and utilized to improve various design characteristics: delay, area, power, testability, etc. The implications of retiming transformations on circuit functionality have been investigated as well [1, 3, 4, 5, 8]. It has been shown that not every state of the original circuit necessarily has an equivalent state in the retimed circuit and vice-versa; only "sufficiently old" [5] states in both circuits have corresponding equivalent states. As a result, the retimed design has to be delayed by applying an arbitrary input sequence (of length equal to the maximum number of forward moves) after which its behavior will be identical to that of the original circuit. A direct consequence is that the original and retimed circuits do not always have the same set of synchronizing sequences [3, 8].

Synchronizing sequences play a vital role in initializing datapath components of a design. These sequential circuits usually have memory elements with no external reset input. After power-up, a synchronizing sequence is applied to bring the circuit to a known state. Consider as an example a memory array: it is not critical for the memory array to power-up in a known state. A memory controller can provide synchronization by applying, for example, an input sequence that sets all the memory locations to 0. Usually, it is not known apriori which synchronizing sequence is applied to bring the circuit to a known initial state. At other times, the selection of a synchronizing sequence is user-controlled. For example, in the case of the memory array, it might be the case that initialization is controlled by a user specified input sequence (through software). That sequence can set all locations to 0, all locations to 1, or some locations to 0 and others to 1. It is apparent from the above discussion that retiming circuits initialized by synchroninzing sequence can potentially cause initialization problems; a synchronizing sequence for a circuit might not synchronize a retimed version of that circuit. This restricts the applicability of retiming in a practical environment. Although a retimed circuit might be superior to the original one in terms of delay, area, or power, it might result in unexpected functionality.

In this paper, we propose a post-retiming synthesis step to maintain a circuit's set of synchronizing sequences. We show that the set of synchronizing sequences of a given circuit can be preserved by modifying transitions from states that are not "sufficiently old" in the retimed circuit. We call these states *invalid states*. Such states arise from forward moves of latches across fanout stems [3, 8]. These moves induce a don't care condition that we utilize to map the behavior of invalid states to valid ones. We incrementally synthesize particular next state functions to reflect the required changes. The outcome is a sequential circuit that can be initialized by any synchronizing sequence used on the original circuit.

The paper is organized as follows. In Section II, we review sequential circuit models, finite state machines, and retiming. Section III reviews the effect of retiming on synchronizing sequences and demonstrates how a post-retiming synthesis can maintain the set of synchronizing sequences. In Section IV, we present an algorithm for implicitly computing the set of invalid states. Section V presents our synthesis algorithm. Experimental results on the ISCAS 89 benchmarks and a set of circuits from an Intel Pentium-III class microprocessor are discussed in Section VI. The paper is concluded in Section VII with pointers to future work.

II. Preliminaries

A synchronous sequential circuit has a finite number m of inputs ($x_1, x_2, \ldots, x_m$), a finite number l of outputs ($z_1, z_2, \ldots, z_l$), and a finite number n of state or memory elements ($y_1, y_2, \ldots, y_n$). The combinational part of the circuit is made up of k internal signals ($w_1, w_2, \ldots, w_k$) representing the outputs of combinational gates. A clock signal clk synchronizes the operation of the memory elements. Each of these signals takes one of two possible values 0 or 1. We will refer to $x_1, x_2, \ldots, x_m$ as the *input variables*, $z_1, z_2, \ldots, z_l$ as the *output variables*, $y_1, y_2, \ldots, y_n$ as the *state variables*, and $w_1, w_2, \ldots, w_k$ as the *internal variables*. The *state* is defined by the assignment of values to the circuit's state variables. In what follows, we use a state s and its binary encoding $\boldsymbol{y} \in \mathbb{B}^n$ interchangeably ($\mathbb{B} = \{0, 1\}$.)

Synchronous sequential circuits can be modeled as a 3-tuple $M = (\langle V, E \rangle, G, L)$ where $\langle V, E \rangle$ is directed graph whose vertices V correspond to the inputs, outputs and internal nodes of the circuit, and whose edges represent the connections between the vertices. G is function mapping every vertex $v \in V$ to its type (primary input, primary output, flip-flop, fanout stem, and different combinational gates) and L is a labeling function mapping each vertex $v \in V$ to a Boolean variable.

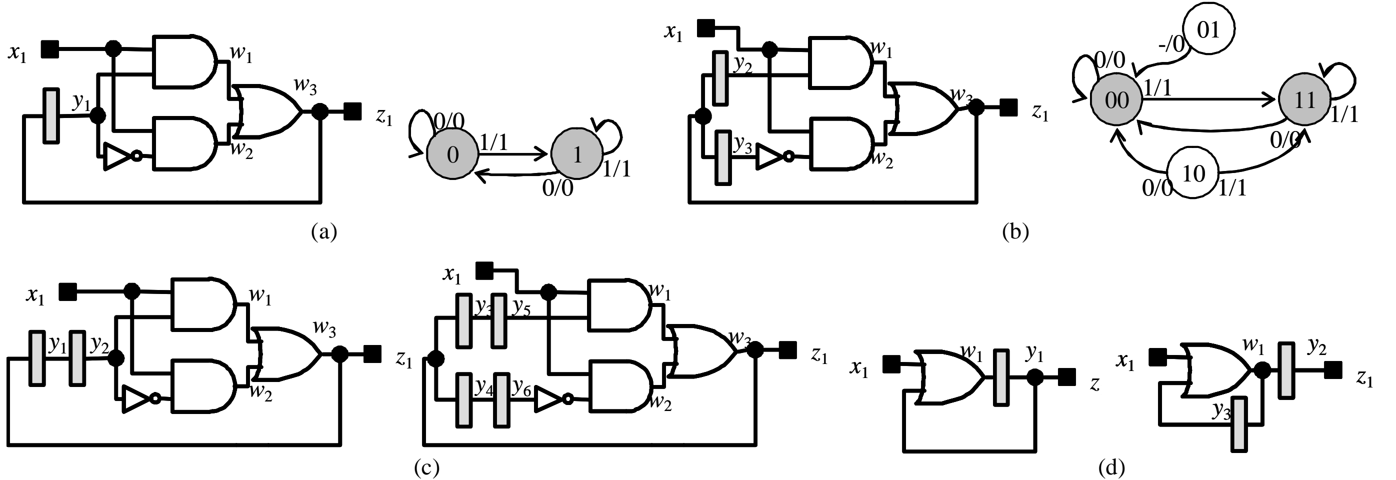

Figure 1: (a) and (b) Single latch movement across a fanout stems destroys a synchronizing sequence of length 1 (c) two latch movements across a fanout stem destroy a synchronizing sequence of length 2 and (d) latch movement across a fanout stem that does not alter the set of synchronizing sequences

A finite-state machine (FSM) M is defined as a 6-tuple $M = (Q, \Sigma, \Delta, \delta, \lambda, Q^0)$ where Q is a finite set of *states*, Σ is the *input alphabet*, Δ is the *output alphabet*, $\delta: Q \times \Sigma \rightarrow Q$ is the state-transition function, $\lambda: Q \times \Sigma \rightarrow \Delta$ is the output function, and Q^0 is the set of initial states. Synchronous Sequential circuits are modeled using FSMs. The state-transition function, δ, determines the next state of the machine based on its current state and inputs. The output function, λ, determines the machine's output based on its current state and inputs. We can write:

$$\boldsymbol{y}^+ = \delta(\boldsymbol{y}, \boldsymbol{x}) \qquad z = \lambda(\boldsymbol{y}, \boldsymbol{x})$$

where $\boldsymbol{x} \equiv (x_1, \ldots, x_m)$, $\boldsymbol{y} \equiv (y_1, \ldots, y_n)$, $\boldsymbol{y}^+ \equiv (y_1^+, \ldots, y_n^+)$, $\boldsymbol{z} \equiv (z_1, \ldots, z_l)$, $\delta \equiv (\delta_1, \ldots, \delta_n)$, and $\lambda \equiv (\lambda_1, \ldots, \lambda_l)$.

The state transition graph of an FSM M, $STG(M)$, is a labeled directed graph $\langle V, E \rangle$ where each vertex $v \in V$ corresponds to a state s_i of M (and is labeled with s_i), and each edge $e \in E$ between two vertices s_i and s_j corresponds to a transition from state s_i to state s_j in M. The edge is labeled i_k / o_l where i_k is the input that causes the transition from s_i to s_j and o_l is the output during that transition. A *synchronizing sequence* of a machine M is an input sequence that drives M to a specific state s_{reset}, regardless of the initial power-up state. If such a sequence exists, M is *resettable* and s_{reset} is a *reset state* of M. A synchronizing sequence is also called a *reset sequence*. Two states s_1 and s_2 of a machine M are *equivalent*, written as $s_1 \sim s_2$, if and only if, for every possible input sequence applied, the same output sequence results.

Retiming [5] is a sequential optimization technique that moves memory elements across combinational blocks. Retiming is formally defined by a function $r: V \rightarrow \mathbb{Z}$ where V is the set of vertices excluding the latches and $\mathbb{Z}$ is the set of integers. $r(v) < 0$ indicates that $|r(v)|$ latches were moved forward from the fanins of v to its fanout, while $r(v) > 0$ indicates that $r(v)$ latches were moved backward from v's fanout to its fanins. Retiming induces a functional relation between the latches of the original and retimed circuits; we call such a relation the *retiming invariant* [6].

III. Retiming and Synchronizing Sequences

It has been previously shown [3, 8] that retiming doesn't completely preserve synchronizing sequences. This scenario arises when latches are moved forward across fanout stems. As an example consider the circuit and its corresponding STG shown in Fig. 1(a). The input sequence 1 is a synchronizing sequence for this circuit. The circuit in Fig. 1(b) is obtained by a retiming across a fanout stem of the original circuit. The input sequence 1 is not a synchronizing sequence for the retimed circuit. As another example, consider the two circuit in Fig. 1(c). We have omitted their STGs for brevity. One can easily verify that the sequence $(0, 1)$ is a synchronizing sequence for the original but not the retimed circuit. It is instructive to note that it is possible for retiming moves across fanout stems to maintain synchronizing sequences. As an example, consider the circuits in Fig. 1(d) where one is derived from the other by a forward movement across a fanout stem. It can be easily shown that both circuits have the same set of synchronizing sequences.

The reason that causes forward latch movement across a fanout stem to destroy a synchronizing sequence is the introduction of additional states that do not correspond to any states in the original circuit. Let us go back to the example of Fig. 1(a). The STG of the retimed circuit has four states. States 00 and 11 (of the retimed circuit) correspond to states 0 and 1 (of the original circuit) respectively. However, 01 and 10 do not correspond to any states; those states violate the retiming condition that requires signals y_1, y_2, and y_3 to be equivalent i.e. $(y_1 = y_2 = y_3)$. We call this retiming condition the retiming invariant. A state violating this invariant is an invalid state. Note that an invalid state can be equivalent to a valid state. For example, 10 is equivalent to 00 in Fig. 1(b). Since transitions from invalid states are different from those of valid states, synchronizing sequences might not be preserved. In the retimed circuit of Fig. 1(b), the invalid state 01 transitions to 00 on an input of 1 while all valid states transition to 11 when the input 1 is applied. Consequently, the input sequence 1 is not a synchronizing sequence for the retimed circuit.

Definition: Let circuit M_2 be obtained from M_1 by retiming and let $I(M_1, M_2)$ be the corresponding retiming invariant. A state s in the STG of M_2 is invalid if it does not satisfy $I(M_1, M_2)$.

Backward retiming moves across a fanout stem do not destroy any existing synchronizing sequence, but might introduce new ones. As an example, if the original and retimed circuits of Fig. 1(a) and (b) reverse roles, then the backward movement of latches y_2 and y_3 introduces a new synchronizing sequence (the sequence 1) which does not synchronize the circuit to the right of the figure. However, introducing a new synchronizing sequence does not pose an initialization problem as does destroying one. Backward and forward movements across

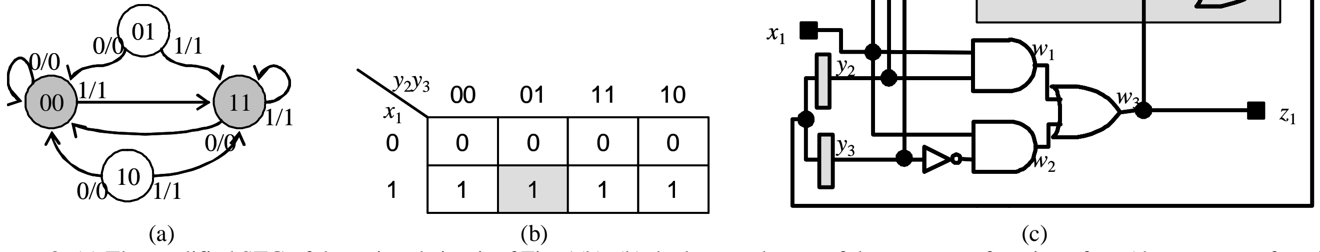

Figure 2: (a) The modified STG of the retimed circuit of Fig. 1(b), (b) the karnaugh map of the next-state function of y_2 (the next-state function of y_3 is identical), and (c) the resynthesized circuit that preserves the set of synchronizing sequences

combinational gates do not alter the set of synchronizing sequences after retiming [3, 8].

In what follows, we show how we can modify the combinational logic of a retimed circuit to preserve the set of synchronizing sequences. The underlying intuition is the observation that valid states never transition to invalid states:

Proposition: Let circuit M_2 be obtained from M_1 by retiming. Then, there exists no transitions from valid to invalid states in the STG of M_2.

Consequently, a retimed circuit can be operating in an invalid state if 1) it powers up in such state or 2) it transitions to it from another invalid state. By the above proposition, we can modify the transitions from invalid states without affecting the functionality of the circuit. By making an invalid state "behave" like a valid state, the set of synchronizing sequences can be preserved. Let us demonstrate the procedure on the retimed circuit of Fig. 1(a). We know that 01 and 10 are invalid states. One possibility is to make state 01 behave like state 00, and 10 behave like 11. In other words, the next state of 01 on an input is identical to the next state of 00 on that input. The same applies for states 10 and 11 . The resulting STG is shown in Fig. 2(a). The new next-state function for y_2 and y_3 is shown in Fig. 2(b). The minterm colored in grey corresponds to the difference between the old and new next-state function. Although we can resynthesize the new function from scratch, we propose to incrementally change the old next state function as illustrated in Fig. 2(c). We synthesize the new minterm $x_1y_2'y_3$ (the new AND gate) and OR it with the original next-state function. The logic introduced is shown in the grey box in Fig. 2(c). It is easy to verify that the new circuit has the same synchronizing sequences as that in Fig. 1(a).

It is instructive to note there exist other possibilities for modifying the behavior of invalid states to preserve synchronizing sequences. For example, one can make both invalid states behave like state 00 , or make both states behave like state 11 . Each different mapping results in a different next-state function. We will discuss how we perform our mapping in Section V. In the next section, we present an algorithm to compute the invalid states after retiming.

IV. Implicitly Computing the Set of Invalid States

We present a procedure to compute the invalid states of a circuit M_2, obtained from M_1 by retiming. The procedure consists of two steps. The first step discovers the latch movements that were applied to the original circuit by calculating the retiming function r at each node. In the second step, retiming is replayed on the original circuit using the retiming function r. During the second phase, whenever there is a latch movement that results in invalid states, the relation between the new latches is recorded. In what follows, we present these two steps in detail.

In the first step, the two circuits $M_1 = (\langle V_1, E_1 \rangle, G_1, L_1)$ and $M_2 = (\langle V_2, E_2 \rangle, G_2, L_2)$ are traversed to discover the latch movements that were carried out during retiming. The output of this step is a function $r: V \rightarrow Z$ which describes how M_2 was obtained from M_1. We assume that the primary input and output names did not change after retiming. We also assume that the order of fanins of each gate was not changed. These requirements simplify the algorithm for structural comparison. The algorithm is illustrated in Fig. 3. Function `DiscoverRetime` finds corresponding primary outputs of C_1 and C_2. It sets $r = 0$ for each of the primary outputs and calls `DiscoverRetimeRec`.

`DiscoverRetimeRec` is a recursive procedure. It first checks whether the node v1 is a primary input node. If this is the case, the procedure returns. If v1 was visited previously, the procedure returns as well. The core computation is performed in the for loop. For each incoming edge to v1, the procedure finds the number of latches, n1, between u1 and v1 where u1 is the first non-latch node on a path ending in v1. Then u2, the node that corresponds to u1 in the retimed circuit is obtained and n2 is computed similarly. Note that n1 and n2 represent the old and new edge weights in retiming terminology. The computation of r is based on the observation that $n2 = n1 + r(v1) - r(u1)$. Thus, $r(u1) = n1 - n2 + r(v1)$. The procedure is then recursively called on u1 and u2. Note that each node is visited once and consequently the complexity of this step is linear in the total number of nodes.

In the second step, the retiming moves computed previously are exercised on $C1$. The algorithm for this step is illustrated in Figure 4. The algorithm examines each non-latch node v with $r(v) \neq 0$ and decides how many latches, if any, can be moved across v. If $r(v) > 0$, we compute the maximum number of latches, m, that can be moved backward across v. If $m \geq r(v)$, only $r(v)$ latches are moved. Otherwise, m latches are moved and the node has to be revisited to complete the remaining moves.

When $r(v) < 0$, a similar procedure is used to move latches forward. In this case, we also compute the set of invalid states. This is achieved by the procedure UpdateCondList(). We maintain a list of Boolean functions called CondList; CondList records the retiming-induced relations that arise among latches that trace back to movements across fanout stems. Initially, CondList is empty. When a latch is moved forward across a fanout stem, the relation that exists between the new latches at the outputs of the fanout stem are added to CondList (the relation forces the values of the latches at the output of the stem to be identical.) In a later step, suppose that $y_1, y_2, \ldots, y_k$ are moved forward creating the retiming relation $\hat{y} = f(y_1, y_2, \ldots, y_k)$. CondList is updated as follows: each condition in CondList is checked to see if

```
function DiscoverRetime(C1,C2)
begin
   for(each v1 in C1)
        v1.isVisited = false
   for(each v1 in C1 with G(v1) == PO)
        find corresponding node v2 of C2
        r(v1) = 0
        DiscoverRetimeRec(v1,v2)
end
function DiscoverRetimeRec(v1,v2)
begin
   if(G(v1) == PI)
        return
   if(v1.isVisited == true)
        return
   v1.isVisited = true
   for(each incident edge on v1)
        Find first non-latch vertex u1
        Find corresponding node u2 of C2
        n1 = num. of latches between u1 and v1
        n2 = num. of latches between u2 and v2
        r(u1) = n1 - n2 + r(v1)
        DiscoverRetimeRec(u1,u2)
end
```

Figure 3: Algorithm for discovering the retimings moves

it has any of $y_1, y_2, \dots,$ or y_k in its support. If this is not the case, CondList is not modified. Otherwise, all the conditions that share variables with $y_1, y_2, \dots, y_k$ are conjoined together with $\hat{y} = f(y_1, y_2, \dots, y_k)$ and the variables $y_1, y_2, \dots, y_k$ are existentially quantified. The result is a new condition that replaces all old conditions.

Let us consider the application of the algorithm on the circuits of Fig. 5. It can be clearly seen that $r(f_1) = -1$, $r(w_1) = -1$ and $r(w_2) = -1$. The algorithm starts by moving y_1 across f_1. Two new latches n_1 and n_2 are created. In addition, the condition $(n_1 = n_2)$ is added to CondList. Next, n_1 and y_2 are moved across w_1 to create n_3. The relation $(n_3 = n_1 \cdot y_2)$ shares a variable with $(n_1 = n_2)$. Thus, both relations are ANDed to get $(n_1 = n_2)(n_3 = n_1 \cdot y_2)$. In addition, the n_1 and y_2 are existentially quantified since we only care for the relation between the new latches n_2 and n_3. The new condition $(n_2 + n_3')$ replaces $(n_1 = n_2)$. Finally, n_2 and y_3 are moved across w_2 creating n_4. As a result, $(n_4 = (n_2 + y_3))$ is ANDed with the previous relation to get $(n_2 + n_3') \cdot (n_4 = (n_2 + y_3))$. Finally, we existentially eliminate n_2 and y_3 to get $(n_3' + n_4)$.

From the retimed circuit, we observe that $y_4 = n_3$ and $y_5 = n_4$. Thus, the relation between the latches of the retimed circuit is $(y_4' + y_5)$. Any state satisfying $(y_4' + y_5)$ is a valid state while any other state is invalid. As an example, the state 10 is an invalid state. This indicates that 10 of the retimed circuit does not have any corresponding state in the original circuit. From the latch movements performed by retiming one can easily verify that for 10 to have a corresponding state, $y_1 = 1$ and $y_1 = 0$ must hold.

Finally, we explain one subtle point about the algorithm in Fig. 4. One can clearly observe that for each different movement across a fanout stem we create a different condition on CondList. Finally, we might end up with more than one condition. A state is valid if it satisfies all conditions. However, we do not conjoin all the conditions together. The intuition behind this is that we can resynthesize the next state functions that depend on each condition separately.

V. Resynthesizing Next-State Functions that depend on Invalid States

In this section, we describe how to modify the behavior of invalid states by resynthesizing particular next-state functions.

```
function ComputeValid(C,r)
begin
   CondList: list of Boolean functions
   CondList = empty
   finished = false
   while !finished
        finished = true
        for(each v in V)
             if(G(v) == latch or r(v) == 0)
                  continue;
             if(r(v) > 0)
                  m = max. backward moves across v
                  if(m >= r(v))
                       Move r(v) latches backward
                  else
                       Move m latches backward
                       finished = false
             else
                  m = max. forward moves across v
                  if(m >= |r(v)|)
                       Move |r(v)| latches forward
                  else
                       Move m latches forward
                       finished = false
                  UpdateCondList()
end
```

Figure 4: Algorithm for computing valid states

If a next-state function f shares variables with a given set of valid states c, the next-state function should be resynthesized. The set of valid states c acts as the care-set for the function f. Let the new synthesized next-state function be f_{new}. Consider a minterm x of f. f_{new} preserves the behavior of f whenever $c(x) = 1$. If $c(x) = 0$, then x corresponds to an invalid state. To map the behavior of this minterm to one that is part of a valid condition, we set $f_{new}(x) = f(y)$ where y is a minterm satisfying $c(y) = 1$. Formally,

$$f_{new}(x) = \begin{cases} f(x) & \text{if } c(x) = 1 \\ f(y) & \text{if } c(x) = 0 \end{cases}$$

where $c(y) = 1$.

A particular operator that satisfies the previous equation and that can be efficiently computed with BDDs is the *constrain operator* [2]. The definition of the constrain operator depends on the distance between two minterms which is defined as follows. If the Boolean variables $x_1, x_2, \dots, x_n$ have the BDD variable ordering $x_1 \le x_2 \le \dots \le x_n$ then the distance between two minterms $\boldsymbol{a} = (a_1, a_2, \dots, a_n)$ and $\boldsymbol{b} = (b_1, b_2, \dots, b_n)$ where $a_i, b_j \in \{0, 1\}$, $1 \le i \le n$, and $1 \le j \le n$ is defined as:

$$|\boldsymbol{a} - \boldsymbol{b}| = \sum_{i=1}^{n} |a_i - b_i| 2^{n-i}$$

The constrain operator $f|c$ is defined as follows:

$$f|c(\boldsymbol{x}) = \begin{cases} f(\boldsymbol{x}) & \text{if } c(\boldsymbol{x}) = 1 \\ f(\boldsymbol{y}) & \text{if } c(\boldsymbol{x}) = 0 \end{cases}$$

where $c(\boldsymbol{y}) = 1$ and $|\boldsymbol{x} - \boldsymbol{y}|$ is minimum. Note that there are other possible ways of constructing f_{new} from f and c. We choose the constrain operator since it can be efficiently computed using BDDs.
Using the constrain operator, we have $f_{new} = f|c$. We can use a synthesis tool to synthesize f_{new}. However, the original next-state function might have a special structure that we like to preserve. Instead, we modify f incrementally to reflect the differences between $f_{new'}$ and f.

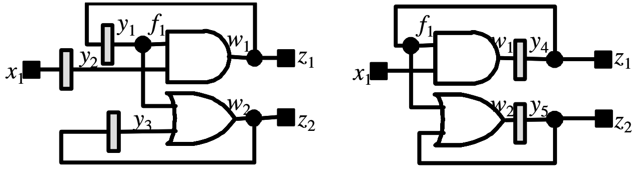

Figure 5: Two circuits one derived from the other using retiming

The synthesized f_{new} is illustrated Fig. 6. The grey logic indicates the incremental changes applied to obtain f_{new} from f. It is easy to verify that $(f+f_{new} \cdot f') \cdot (f_{new}' \cdot f)' = (f+f_{new}) \cdot (f'+f_{new}) = f_{new}$.

VI. Experimental Results

To experimentally evaluate our proposed framework, we implemented the proposed algorithm in C++. We use the CUDD BDD package [9] for Boolean function manipulations. We report our results on the ISCAS 89 benchmarks and circuits from an Intel Pentium-III class microprocessor. All experiments were conducted on a 2 GHz Pentium 4 machine having 1 GB of RAM and running the Linux operating system.

Our results are presented in Table 1. Circuit names are listed in column 1. Column 2 shows the retiming optimization performed using SIS [7] (version 1.2): "delay" indicates retiming for minimum clock period while "area" indicates retiming for minimum area. All circuits were first retimed for minimum clock period. Of these, the ones unaltered by minimum clock period retiming were retimed for minimum area. Only four circuits s344, s349, s820, and s832 were not altered under both constraints. For s35932 and s38417 retiming for minimum clock period did not alter the circuits and when retiming for minimum area SIS ran out of memory. Results for these circuits are not reported.

TABLE 1: Results on the ISCAS89 circuits

Circuit	Retimed For	base circuit	#resyn/#latches	Cover Size
s298	delay	retimed	0/14	0
s382	area	retimed	15/21	2209
s499	delay	retimed	0/22	0
s510	delay	-	-	-
s526	delay	retimed	0/21	0
s635	delay	retimed	0/32	0
s641	area	retimed	19/19	613
s713	area	retimed	19/19	613
s938	delay	retimed	MEM OUT	-
s953	delay	retimed	0/29	0
s967	area	retimed	MEM OUT	-
s991	delay	retimed	MEM OUT	-
s1196	area	-	-	-
s1269	delay	original	81/96	6,0929,794
		retimed	0/37	0
s1423	delay	retimed	0/24	0
s1488	delay	retimed	0/6	0
s1512	delay	-	-	-
s3271	delay	original	MEM OUT	-
		retimed	0/116	0
s3330	delay	original	MEM OUT	-
		retimed	0/81	0
s3384	delay	original	MEM OUT	-
		retimed	12/183	27,628,460

For most of the smaller circuits, retiming resulted in backward moves only. To be able to test our approach, we reversed the role of the original and retimed circuits to get forward moves. For circuits in which retiming resulted in forward and backward moves, we applied our algorithms on both the original and retimed circuit. This is shown in Column 3: "original" indicates that retiming was applied on the original circuit to get the retimed version while "retimed" indicates that retiming was applied on the retimed circuit to get the original circuit. For s510, s1196, and s1512 retiming did not result in any movement across fanout stems. Thus the set of synchronizing sequences is automatically preserved. This is indicated by a "-" in Column 3. Column 4 indicates the number of next state functions that need to be resynthesized out of the total number of these functions. The last column indicates the cover size after the next state function are synthesized. The number is obtained by finding the size of a BDD cover for the two functions $f_{new}f'$ and $f_{new}'f$ and accumulating that for all next-state functions that have to be resynthesized.

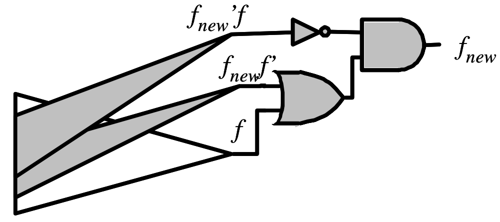

Figure 6: Resynthesized next-state function

The main observation from the results in the table is that in most cases, none of the next-state functions need to be resynthesized and consequently there is no area overhead. In other words, the set of synchronizing sequences are automatically preserved. For these circuits, even though retiming moved latches forward across fanout stems, no invalid states were generated by this process and thus there is no need to modify the next-state logic. We traced back the cases under which this occurs. We identified two cases that frequently occur for these circuits. The cases are illustrated in Fig. 7. In Fig. 7(a), although retiming moved the two registers y_1 and y_2 across a fanout stem, the final result of retiming is a single register y_3. One can easily see that $(y_3 = y_1'y_2')$ is the relation that holds between the three registers. When we existentially eliminate y_1 and y_2, we get the identity. Consequently, there are no restriction on the values that y_3 can assume. Fig. 7(b) shows two circuits where one is retimed from the other. We can compute the relation between the old and new latches as $(y_4 = y_1)(y_5' = (y_1'y_3' + y_2'y_3'))$. To obtain the relation between y_4 and y_5, we existentially quantify y_1, y_2, and y_3. On performing this, we obtain the identity relation. Thus, although registers might be moved across fanout stems, the combinational logic that they will move through might result in no invalid states. A particular condition of this second case that happens a lot in the ISCAS89 benchmarks is movement of a fanout register across identical combinational gates. For this case, no invalid states will result as well.

MEM OUT in Column 4 indicates that the BDD ran out of memory (1GB limit) when computing the set of invalid states.

TABLE 2: Results from an Intel Pentium-III class microprocessor

Circuit Name	Before Latches	After Latches	Time (sec)	Mem (MB)	Affected NSFs	Cover Size	Var Order
C1	219	222	158	301	35	3.09E+13	Dyn
C2	221	223	5.9	206	1	199848	Dflt
C3	226	229	6	198	1	84310	Dflt
C4	231	233	3.2	197	1	10520	Dflt
C5	243	246	4.6	204	1	244412	Dflt
C6	351	357	9.1	271	23	367916	Dflt
C7	377	379	28.1	427	1	63183252	Dflt
C8	428	437	13.6	304	25	280044	Dflt
C9	480	491	28.4	352	11	19207990	Dflt
C10	1855	1857	35.2	316	1	2.71345E+11	Dyn

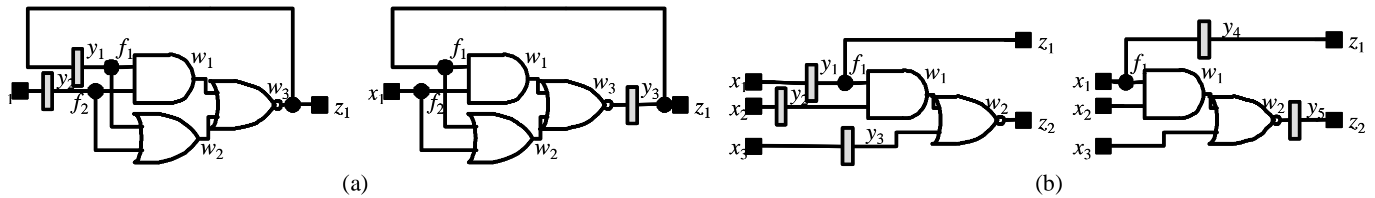

Figure 7: (a) and (b) Examples of retimed circuits with no invalid states

Similarly, in Table 2, we list the results of the application of our techniques on selected circuits from an Intel Pentium-III microprocessor. In Column 2, we list the number of latches in the circuit before it was retimed. In Column 3, we list the number of latches after retiming operations were performed. In this case, we considered only retiming operations that involved movements of latches forward through fanout stems. Columns 4 and 5 list the CPU time and memory requirements for the execution of the algorithms in Figure 3 and Figure 4 on dual Intel Xeon 2.4GHz servers with 1GB of RAM running the Linux operating system. These algorithms were implemented on top of the capabilities of Intel's Formal Equivalence Verification system. In Column 6, we list the number of next state functions that turned out to be affected and in need of resynthesis. In Column 7, we list the sizes of the covers of the BDDs that get computed in accordance to what we have listed in Column 5 of Table 1. Finally, in Column 8, we list whether dynamic reordering was necessary (indicated by Dyn) or the default ordering that the Formal Equivalence Verification system produces is sufficient (indicated by Dflt).

VII. Conclusion

Today's designs have complex performance requirements that necessitate the application of rigorous optimization algorithms. Retiming is one such algorithm that proved to be successful at improving various design parameters. However, the practicality of retiming is hindered by initialization problems that can alter a circuit's functionality. This is due to the fact that a synchronizing sequence that initializes a design might not initialize its retimed version.

We tackled the above problem in this paper. We showed that it is possible to maintain the synchronizing sequences of a circuit after retiming by introducing limited combinational modifications to the retimed circuit. We established the notion of "invalid states" that arise by forward movements of latches across fanout stems and showed that these states cause initialization problems after retiming. We presented an algorithm that implicitly computes the set of invalid states. Next, we showed how we can preserve synchronizing sequences by making invalid states "imitate" valid states in their behavior. We showed how to accomplish this by incremental resynthesis of particular next-state functions.

The experimental results were quiet interesting. For most of the retimed benchmarks, no invalid states were introduced even in the presence of forward movements across fanout stems. We identified general patterns under which such conditions arises. We also demonstracted the application of the algorithm on industrial circuits.

Currently, we are investigating the effect of our synthesis on the area and performance of the circuits in which the next-state functions had to be altered. In addition, we are trying to develop a more advanced engine for computing the invalid states that combines BDD and SAT technologies. This can help avoid the memory explosion we faced in some of the benchmarks. Finally, we like to study the modifications needed to preserve a subset of the synchronizing sequences. This problem has practical significance as the designer might have knowledge of what particular sequences are used in initialization.

VIII. Acknowledgments

This work is funded by the DARPA/MARCO Gigascale Silicon Research Center.

References

[1] S. Brookes, "Using Fixed-Point Semantics to Prove Retiming Lemmas," in *Formal Methods in System Design*, 1993

[2] Olivier Coudert, and Jean Christophe Madre, "Symbolic Computation of the Valid States of a Sequential Machine: Algorithms and Discussion," in *Proceedings of the International Worshop on Formal Methods in VLSI Design*, January 1991.

[3] A. El-Maleh, T. Marchok, J. Rajski, and W. Maly, "Behavior and Testability Preservation Under the Retiming Transformation," in *IEEE Transactions on Computer-Aided Design*, vol. 16, pp. 528-543, May 1997.

[4] G. Even, "The Retiming Lemma: A Simple Proof and Applications," *Integration, VLSI Journal* 1996.

[5] C. E. Leiserson and J. B. Saxe, "Retiming Synchronous Circuitry," *Algorithmica*, vol. 6, pp. 5-35, 1991.

[6] M. Mneimneh and K. Sakallah, "REVERSE: Efficient Sequential Verification for Retiming," International Workshop on Logic Synthesis, 2003.

[7] E. Sentovich, K. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. Stephen, R. Brayton, and A. SAngiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," *University of California, Berkeley, Tech. Report*, May 1994.

[8] V. Singhal, C. Pixley, R. L. Rudell, and R. K. Brayton, "The Validity of Retiming Sequential Circuits," in *Proceedings of the 32nd Design Automation Conference,* 1995.

[9] F. Somenzi, "CUDD: CU Decision Diagram Package Release 2.3.1," Technical Report, University of Colorado at Boulder, 2001.