Sequential Optimization in the Absence of Global Reset

VIGYAN SINGHAL
Tempus-Fugit
CARL PIXLEY
Synopsys
ADNAN AZIZ
University of Texas at Austin
SHAZ QADEER
Microsoft
and
ROBERT BRAYTON
University of California at Berkeley

We study the problem of optimizing synchronous sequential circuits. There have been previous efforts to optimize such circuits. However, all previous attempts make implicit or explicit assumptions about the design or the environment of the design. For example, it is widespread practice to assume the existence of a hardware reset line and consequently a fixed power-up state; in the absence of the same, a common premise is that the design's environment will apply an initializing sequence. We review the concept of *safe replaceability* which does away with these assumptions and the *delay-safe replaceability* notion, which is applicable when the design's output is not used for a certain number of cycles after power-up. We then develop procedures for optimizing the combinational next-state and output logic, as well as routines for reencoding the state space and removing state bits under these replaceability criteria. Experimental results demonstrate the effectiveness of our algorithms.

Categories and Subject Descriptors: B.6.3 [**Logic Design**]: Design aids—*automatic synthesis*

General Terms: Algorithms, Design

Additional Key Words and Phrases: Sequential logic synthesis, safe replaceability, no-reset latches

1. INTRODUCTION

Logic synthesis is the process of transforming an architectural- or behavioral-level description of a design into an optimized gate-level implementation. There

The support of the NSF under grant CCR-9702919 and The State of Texas Higher Education Coordinating Body under grant ARP 003658-0235-1997 is gratefully acknowledged.
Authors' addresses: V. Singhal, Tempus-Fugit, Albany, CA; C. Pixley, Synopsys, Hillsborough, OR; A. Aziz, University of Texas, ACE 6.120, Austin, TX 78712; email: adnan@ece.utexas.edu; S. Qadeer, Microsoft, Redmond, WA; R. Brayton, University of California, Berkeley, CA.

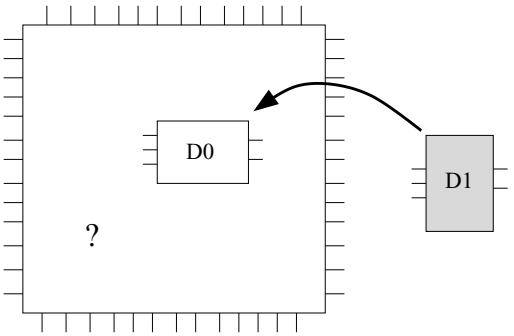

Fig. 1. Design replacement.

are many components of logic synthesis—transforming a register-transfer level (RTL) design into a gate-level design, restructuring a gate-level design to achieve an optimum combination of timing, area, power, and testability, mapping a gate-level design to a library of technology-dependent components, and so on.

Large designs are built in a *hierarchical* manner, that is, by designing individual pieces first, and then *composing* them. In this article, we are concerned with developing sequential synthesis procedures relative to an appropriate notion of *replaceability*, namely, a criterion for telling us when a design $D1$ can be used in place of another design $D0$, as illustrated in Figure 1. It should be clear that a criterion for design replacement is an integral part of developing synthesis algorithms. Two desirable features of any criterion for replacement are *soundness* and *completeness*.

Soundness of a replacement criterion entails that if a design $D1$ satisfies the criterion for replacing $D0$, then any design where $D0$ occurs as a subdesign should perform exactly as it did previously when $D0$ is replaced with $D1$. In this work we make no assumptions about the behavior of the logic with which $D0$ will be composed, which we refer to as $D0$'s environment. Specifically, we do not wish to take for granted that the environment may produce a specific initializing sequence. Even though soundness may seem to be an obvious characteristic of any replaceability criterion, we have shown that much previous work fails to meet the requirements stated above [Singhal et al. 2001, Section III].

Completeness of a replacement criterion entails that if a design $D1$ has the property that any design where $D0$ occurs as a subdesign performs exactly as it did previously when $D0$ is substituted by $D1$, then $D1$ should be a replacement for $D0$ under the criterion. Informally, this means that the criterion should be as "weak" as possible. Completeness is desirable, since when performing synthesis under the replacement criterion, it allows the maximum amount of flexibility for optimization.

The motivation for our work can be seen from the fact that even though there has been much research in sequential synthesis, there has been little transfer of the technology to industry. For example, when a gate-level netlist is passed on to a synthesis tool for optimization, such a netlist is almost always combinational, that is, without any memory elements; the design is often cut at

latch boundaries before passing to a synthesis tool. (Retiming is probably the only exception to this.)

One major problem with sequential synthesis methods in the literature is the assumption of a designated initial state for a design; often it is not possible to live with such an assumption for the optimization of an arbitrary sequential netlist in an industrial setting. For arbitrary sequential netlists without a designated initial state, one notion that can be used for replacement is the concept of FSM equivalence (Hartman is and Stearns [1996, p. 23], also Definition 2 in this article). However, this is often too strict, and we show that it is possible to use weaker notions and still preserve soundness. A number of other papers deal with sequential designs that lack a designated initial state. For example, Pixley [1990, 1992] defined the notion of sequential hardware equivalence (SHE) which does away with the DIS assumption; roughly, it requires equivalence relative to an initializing sequence. However, in Singhal et al. [2001, Section III] we showed that SHE is not sound. (Additional criticisms of basing a synthesis technique around initializing sequences is the complexity that they add, and the possibility that the environment may not be able to generate the desired sequence.) Other papers that do away with the DIS assumption and perform sequential redundancy identification and removal include those by Cheng et al. [Cheng 1993; Entrena and Cheng 1993] and by Pomeranz and Reddy [1993, 1996].

In Singhal et al. [2001, Section III] we provide a detailed account of these, and demonstrate that the replacements generated by the proposed techniques are not sound. By this we mean that there exists the possibility that an implementation synthesized by these techniques may not perform exactly as the original specification. We stress this is a possibility and not a certainty; existing procedures can still be used, as long as the implementation is verified against the specification.

The rest of this article is structured as follows. Basic issues and definitions related to synthesis are presented in Section 2. We review the notion of *safe replaceability*, and its generalization, *delay-safe replaceability*, in Section 3. We then present our specific contributions.

(1) In Section 4 we present an algorithm for optimization of next-state and output combinational logic that exploits the flexibility given by our replaceability criteria.
(2) In Section 5 we present an algorithm that performs reencoding of the state space, resulting in further optimization. Experimental results, specifically those in Table VI, demonstrate the effectiveness of our algorithms.

Finally, in the last section we discuss where our work leads in the future and the lessons we learned in this research.

We stress that the focus of this article is to develop algorithms for optimizing sequential designs under a sound and complete notion of design replacement. As such, the theoretical development of the replacement criteria is limited; we present a detailed exposition of the theory of safe and delay-safe replaceability in Singhal et al. [2001]. This article unites and extends results reported

by the authors at several conferences [Singhal and Pixley 1994; Qadeer et al. 1996; Singhal et al. 1995; Pixley et al. 1994] and in Singhal's PhD dissertation [1996].

2. PRELIMINARIES

2.1 Memory Elements

We are dealing with synchronous, sequential gate-level circuits. The sequential nature of such circuits is implemented by memory elements, either latches or flipflops. We assume that the memory elements are edge-triggered (most of our results generalize to level-sensitive latches; the details are tedious). We use the term "latch" to denote such an element.

Latches come in two flavors: latches with a hardware reset line (we call them "reset latches") and latches without a hardware reset line ("no-reset latches"). We assume that all latches in the designs are no-reset latches (in Singhal et al. [1996], we show how to model a reset latch with a no-reset latch and some combinational logic). When a design consisting of t no-reset latches is switched on, it nondeterministically starts up in one of the 2^t *power-up states*.

Many sequential synthesis and verification studies, for example, Coudert and Madre [1990], Cho et al. [1990], Lin et al. [1990], and Berry and Touati [1993], rely on the supposition that all latches are reset latches by assuming that every design has a designated initial state. Assuming a designated initial state greatly simplifies the analysis of many problems. However, although this assumption is applicable to some designs, we later argue that it does not apply to all designs. This article does not assume a designated initial state, and for this reason, it differs from many previous studies.

We now argue why it is so important to analyze designs without assuming a designated initial state. In our experience, many industrial designs do have no-reset latches. There are several reasons for this: no-reset latches occupy less area and contribute less delay to the circuit. The use of no-reset latches avoids the problem of routing a global reset line. Another reason for not assuming a designated initial state, even if all latches are reset latches, is that we cannot assume that whenever the circuit operates, the global reset line has been pulled. Indeed, we have observed real designs where it was not the case that the global reset was activated from the first clock cycle; in some modes of operation, the design was being used even before activating the reset line. In such situations it would have been very dangerous for a synthesis tool to make the assumption that the design was used only after pulling the reset line, and the behavior of the designs in the states unreachable from the designated initial state does not matter. Another problem with the designated initial state model is caused by circuits that have more than one class of latches; each class is wired to a different reset line. It is not clear what the designated initial state is in this situation. Also, we have observed some designs where some reset lines are outputs of combinational logic and it is not clear if there is a meaningful initial state at all.

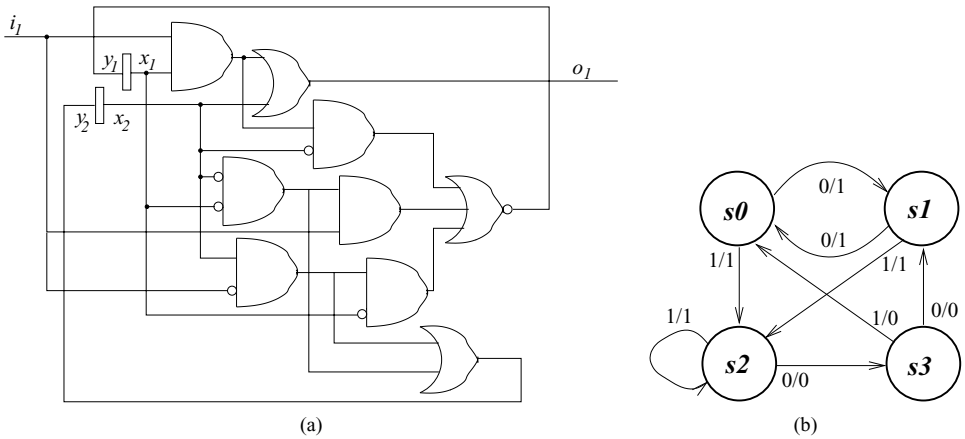

Fig. 2. A netlist and its corresponding STG: (a) example netlist C; (b) the STG for netlist C.

2.2 Terminology

The theory of sequential replacement that we present in this article is applicable at many levels of abstraction. However, we restrict ourselves to two different levels of abstraction for representing digital designs: behavioral-level (finite-state machines) and gate-level (represented by netlists).

Formally, a **Finite State Machine** (FSM) is a quintuple, $(Q, I, O, \lambda, \delta)$, where Q is the set of states, I is the set of input values, O is the set of output values, λ is the output function, and δ is the next state function. The output function λ is a completely specified function with domain $Q \times I$ and range O. The next state function is a completely specified function with domain $Q \times I$ and range Q.

We find it convenient to represent an FSM by a **state transition graph** (STG). This is a directed graph where the vertices represent the states and the edges are transitions between states. An edge is labeled with the input value which causes that transition and the resulting output value.

A netlist D consists of a set of interconnected latches and gates. A design with a input wires, b output wires, and c latches is naturally associated with an FSM D. The input space $I_D = \{0,1\}^a$, the output space $O_D = \{0,1\}^b$, and the state space $Q_D = \{0,1\}^c$. The next state function is $\delta_D : Q_D \times I_D \mapsto D$ and output function $\lambda_D : Q_D \times I_D \mapsto O_D$ is defined by the corresponding logic. An example of this association is shown in Figure 2.

We often abuse notation and use D to also denote the set of states of the associated FSM; it is clear from the context if D refers to the design or to the set of states. We are not assuming explicit set or reset pins to any latch; thus when the design powers up it can nondeterministically power up in any one of the 2^c states.

We also use λ_D and δ_D to denote the output and next state functions on sequences of inputs: for any state $s \in Q_D$ we have $\delta_D(s, \epsilon) = s$ and $\lambda_D(s, \epsilon) = \epsilon$, where ϵ represents the length-0 sequence; otherwise, if $\pi = a_1 \cdot a_2 \cdot a_3 \cdots a_q \in$

I_D^q is a sequence of q inputs, then $\lambda_D(s,\pi) = \lambda_D(s,a_1) \cdot \lambda_D(\delta_D(s,a_1),\pi')$ and $\delta_D(s,\pi) = \delta_D(\delta_D(s,a_1),\pi')$, where $\pi' = a_2 \cdot a_3 \cdots a_p$.

All notions of design replacement described in this article are meaningful only if the two designs have the same number of input and output wires.

Definition 1. Given two states $s_0 \in Q_{D_0}$ and $s_1 \in Q_{D_1}$, state s_0 is *equivalent* to state s_1 (denoted by $s_0 \sim s_1$) if for any sequence of inputs $\pi \in I^*$, it is the case that $\lambda_{D_0}(s_0,\pi) = \lambda_{D_1}(s_1,\pi)$.

The notion of state equivalence generalizes to *FSM equivalence* [Hartmanis and Stearns 1966, p. 23]:

Definition 2. Two FSMs M_1 and M_2 are *equivalent* ($M_1 \equiv M_2$) if for each state s in M_1 there is a state t in M_2 such that $s \sim t$, and for each state t in M_2 there is a state s in M_1 such that $s \sim t$.

Definition 3. A set of states $S \subseteq Q_D$ is a *strongly connected component* (SCC) if it is a maximal set of states such that for any two states $s_0, s_1 \in S$, there exist input sequences π_0 and π_1 so that $\delta_D(s_0,\pi_0) = s_1$ and $\delta_D(s_1,\pi_1) = s_0$.

Definition 4. A set of states $S \subseteq Q_D$ is a *terminal strongly connected component* (tSCC) if S is an SCC and if for any state $s \in S$ and any input a: $\delta_D(s,a) \in S$.

2.3 Design Composition

Because we are interested in the problem of replacing designs without affecting the interaction with the environment, the issue of composing designs is crucial to this article. Therefore, it is important to define precisely what we mean by "design composition" and the subtleties involved in the composition of designs.

Composition of two netlists simply comprises placing the two netlists next to each other and connecting the pairs of input–output signals which are required by the composition. Composition of two designs entails "hiding" of some signals: each signal that is hidden is an output of one design and an input of the other. The inputs of the composed design are the inputs of the component designs that have not been connected to an output. Some subset of the components' outputs is designated as being the outputs of the composed design. We denote a composition of netlists $D1$ and $D2$ by $D1 \otimes D2$.

As an example, consider the design in Figure 3. Design $D1$ has input $x1$ and output $L2$; design $D2$ has inputs $u1$ and $u2$ and output $v1$. The composition involves connecting $L2$ with $u2$ and $v1$ with $x1$; thus $u1$ is the only input of the composed design. The signals $v1$ and $L2$ are designated to be outputs.

However, such a composition may sometimes create combinational cycles. If a netlist has a combinational cycle, it may be impossible to associate an STG with it because the next state and output functions may not be determined from such a circuit. The hardware realization in silicon of such a circuit may oscillate for an indeterminate time. Issues regarding behavior of circuits with combinational loops have been dealt with elsewhere [Shiple et al. 1996]. In this article we talk about design composition only when it does not result in combinational loops.

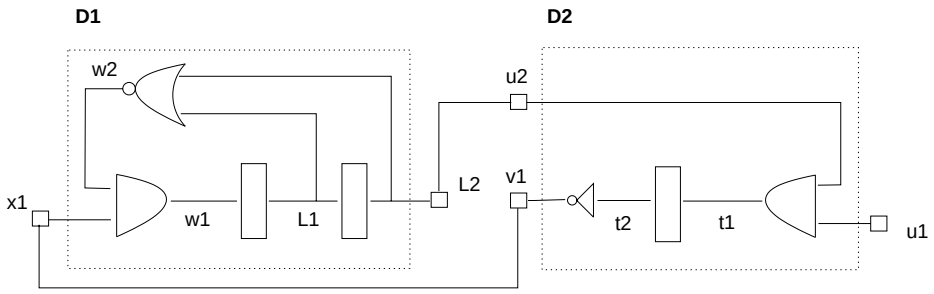

Fig. 3. Netlist composition.

3. SAFE AND DELAY-SAFE REPLACEABILITY

In this section we review the notions of safe replaceability and delay-safe replaceability for sequential circuits. Detailed proofs and illustrations of the properties of safe and delay-safe replaceability are given in Singhal et al. [2001]; here we simply state the properties and give the intuition behind them.

Because modern digital designs are so large and complex, we take an *arbitrary* block of sequential logic and replace it with an optimized component so that it is not possible to detect the replacement, regardless of the surrounding logic. For the reasons given in Section 2.1, we make no assumptions about the behavior of the surrounding logic. Specifically, we do not assume the existence of an initializing sequence that the environment may apply after power-up.

For combinational circuits, there is a widely used notion of replacement that is used in logic synthesis tools: a circuit C is a valid replacement for circuit D, if for any input the output vector produced by D is identical to that produced by C. This is an acceptable criterion because it implies that for any environment E, the composition of E with C behaves identically to the composition of E with D.

3.1 Notion of Safe Replaceability

Definition 5. Design D_1 is a *safe replacement* for design D_0 (written as $D_1 \preceq D_0$) if given any state $s_1 \in Q_{D_1}$ and any finite input sequence $\pi \in I^*$, there exists *some* state $s_0 \in Q_{D_0}$ such that the output behavior $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$.

We show in Singhal et al. [2001] that the condition for safe replacement is complete and sound; that is, it provides maximum flexibility while guaranteeing that the replacement cannot be detected by the environment. Intuitively, this is for the following reason. First, if we make the above condition any weaker, then there exists an input sequence π and a state s in the new design D_1 so that if D_1 powers up in s and π is applied to D_1, the resulting output sequence could not have been seen from any state in D_0. Hence some environment could detect the replacement. Conversely, if $D_1 \preceq D_0$, then for every input sequence any power-up state of D_1 behaves like some power-up state of D_0, implying that any behavior from any state of D_1 is acceptable.

3.2 Properties

Now we present some interesting properties of the safe replacement condition. Illustrations and proofs of these properties may be found in [Singhal et al. 2001].

The following proposition guarantees that safe replacements are preserved under arbitrary composition.

PROPOSITION 3.1. *If $D \preceq C$, then for any composed design R, we have $R \otimes D \preceq R \otimes C$.*

Even though there is some flexibility for the implementation of the replacement design, it cannot have arbitrarily few states: each tSCC (cf. Definition 4) in the replacement design must be equivalent (in the sense of Definition 2) to some tSCC in the original design.

THEOREM 3.2. *If $D_1 \preceq D_0$, and M_1 is a tSCC in design D_1, then there must be a tSCC M_0 in design D_0 such that $M_0 \equiv M_1$.*

3.3 Notion of Delay-Safe Replacement

It is often the case that in practice after a design is powered up, some clock cycles will be allowed to run through the design before the design is used by its environment. We can use this flexibility to justify a notion of replacement that is weaker than "safe replacement" because the environment of the design agrees to let the replacement design stabilize for a few extra clock cycles after power-up. In this section, we investigate this notion of "delay replacement."

It is already the case that several effective optimization techniques, such as retiming, do not always result in safe replacements, but cause delay replacements. To understand why, consider the state transition graph again. In every design, there is a subset of states into which the design must eventually fall no matter what sequence of inputs is given to the design. For example, suppose there is a state s_1 to which no state (including itself) transitions under any input. This state represents an *ephemeral* state of the machine that cannot be visited beyond one clock cycle. Any such one-cycle ephemeral state is irrelevant to the steady-state operation of the design and so, by letting the design just "coast" for one cycle, it can be eliminated. To get to the "core" behavior of a design, delete all such one-cycle ephemeral states. However, notice that a new ephemeral state may appear, that is, a state, say s_2, to which only a one-cycle ephemeral states can transition.

Definition 6. Given a design D, the *n-cycle delayed design* (denoted by D^n) is the restriction of D to the set of states $\{s | \exists \pi \in I^n, s' \in D : \delta_D(s', \pi) = s\}$; that is, a state s belongs to D^n if and only if there exist a power-up state s' in D and an input sequence of length n which drives s' to s.

Definition 7. Given a design D, denote by D^∞ the set of states $\{s | \forall n : \exists \pi \in I^n, s' \in D : \delta_D(s', \pi) = s\}$, that is, a state s belongs to D^∞ if and only if for each natural number n there exist a power-up state s' in D and an input sequence of length n which drives s' to s.

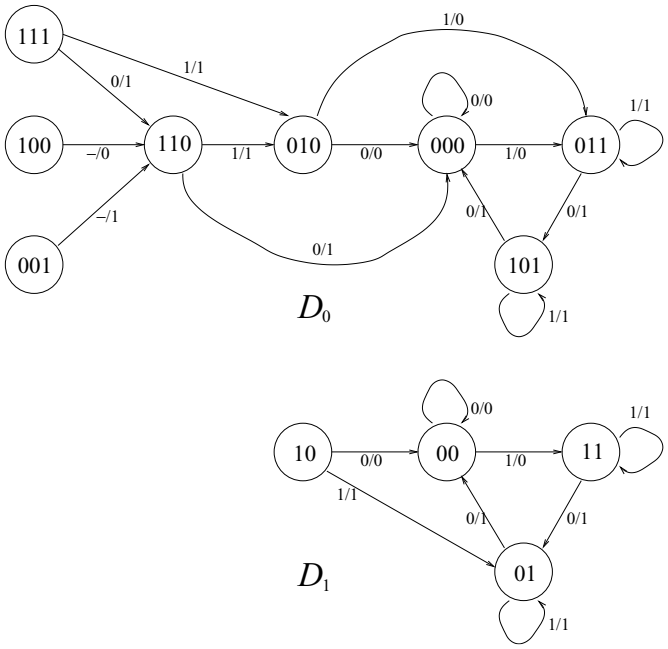

Fig. 4. Example of a safe replacement ($D_1 \preceq D_0$).

The set D^n is the set of states into which any state must fall when clocked n times with any sequence of inputs. It is easy to see that if $m > n$, the set of states in D^m is a subset of the states in D^n. The design D^∞ can be obtained by a fixed-point operation starting from D, because $D^\infty = D^n$, where n is the smallest number such that $D^{n-1} = D^n$.

We refer to states in D^∞ as the **stable** states of D, and the states in $D \setminus D^\infty$ as the **transient** states of D. After powering up a design, if a sufficiently long sequence of arbitrary inputs is applied to the design, it will enter the stable set.

For example, consider the design shown in Figure 4. For this design R, the various n-delayed designs are: $R = \{111, 100, 001, 110, 010, 000, 101, 011\}$, $R^1 = \{111, 110, 010, 000, 101, 011\}$, $R^2 = \{010, 000, 101, 011\}$, and $R^3 = R^4 = \cdots = R^\infty = \{000, 101, 011\}$.

We now present our condition for delay replacement.

Definition 8. Given a design D, a new design C is an *n-delay replacement* for D, if $C^n \preceq D$.

As an example of delay replacement, consider designs D_0 and R in Figures 4 and 5. It can be seen that $R^1 \preceq D_0$; however, $R \not\preceq D_0$ (the state $100 \in R$ produces the output sequence $0 \cdot 0 \cdot 0$ on input sequence $0 \cdot 1 \cdot 0$; this input–output behavior cannot be seen from any state in D_0).

The notion of n-delay replaceability is analogous to the ordering on the natural numbers. For example, a two-delay replacement followed by a three-delay replacement on the same design results in a n-delay replacement for any $n \geq 5$. More generally, we prove the following series of results in Singhal et al. [2001].

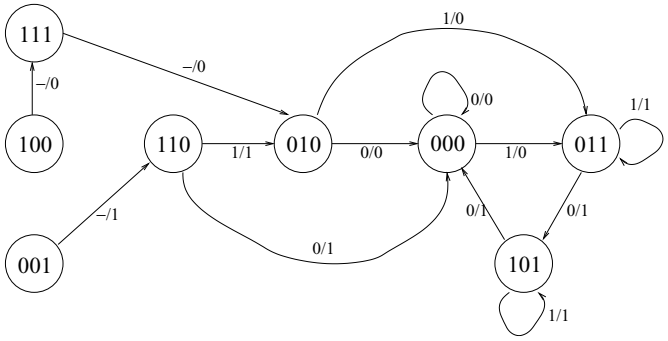

Fig. 5. Example design R.

PROPOSITION 3.3. *If $C^n \preceq D$, and $m > n$, then $C^m \preceq D$. If $C^n \preceq D$ and $B^m \preceq C$, then $B^{m+n} \preceq D$.*

Properties of Delay Replacements

For the optimization of any design, we can select arbitrary subpieces of the design and perform delay replacements on these. We now state results on the effect of making a delay replacement on the larger design.

PROPOSITION 3.4. *If $Q^m \preceq R$ and $C^n \preceq D$, then $(Q \otimes C)^p \preceq (R \otimes D)$, where $p = \max(m, n)$. If $Q^m \preceq R$ and $C^n \preceq D$, then $(Q \otimes C \otimes P)^p \preceq (R \otimes D \otimes P)$, where $p = \max(m, n)$.*

This means that delay replacements can be made in different parts of the design, and the resulting overall design is as safe as the weakest individual replacement (the replacement with the greatest slack).

However, if consecutive delay replacements are made on overlapping subpieces on a large design, the delays add up. This can be seen as a consequence of Propositions 3.3 and 3.4.

PROPOSITION 3.5. *Let design $D = P \otimes Q$. Let $R^m \preceq P$, and let design $C = R \otimes Q$ also be equal to $S \otimes T$ (another decomposition of the same design C), and let $U^n \preceq S$. Then, if $B = U \otimes T$, it is true that $B^{m+n} \preceq D$.*

Note that, as a special case of this, if the two subpieces are identical, we already know from Proposition 3.3 that the entire design in $2n$-delay safe. In summary, making n-delay replacements on nonoverlapping subpieces of a design will produce an entire design which is n-delay safe; on the other hand, if two consecutive n-delay optimizations are made on subpieces that are overlapping, we get an entire design that is $2n$-delay safe.

3.4 Implications for Synthesis

The notion of delay-safe replaceability relies on the assumption that any realistic design will be used only after some cycles have elapsed after power-up. This initialization slack is known in advance: it is part of the design specification. Sequential optimization can use this initialization slack, say n cycles, in

the following ways. First, the design can be partitioned into nonoverlapping, tractably sized components. Then each component can be optimized using the n initialization slack cycles. Proposition 3.4 formalizes the fact that this strategy will produce an entire design that is n-cycle delay-safe replacement. Second, designs can be optimized using components that do overlap. Proposition 3.5 shows that in the worst case, the delay needed accumulates for overlapping pieces. So the designer must take care not to exceed the total allotment of n cycles. Of course hybrids of the two approaches can be used. It is important to note that for any of the approaches, only the transient behavior of the design is modified.

In Sections 4 and 5 we present our synthesis strategy for making delay replacement, and show that significant sequential logic optimizations can be obtained.

4. COMBINATIONAL RESYNTHESIS FOR DELAY-SAFE REPLACEMENT

In this section we use the flexibility due to the delay replacements to perform optimization on the combinational part of the circuit. In the next section we show how to perform latch minimization using this flexibility.

Let the original design D have $(k + 1)$ distinct delayed designs, D^0, $D^1, \ldots, D^k$; for any $j \geq k : D^j = D^k = D^\infty$. Clearly, any state reachable from itself under some input sequence belongs to each D^i and to D^∞. We do not alter the behavior of any such state (a "stable" state). All the flexibility for resynthesis comes from the set of "transient" states; that is, $D \setminus D^\infty$.

Recall that an n-delay replacement criterion allows the new design to have some flexibility over the first n clock cycles after the power-up; after these clock cycles its input–output behavior is indistinguishable from the original design. We use this slack n for resynthesis: the higher n is, the greater the flexibility allowed for resynthesis. (Of course, the environment of the design cannot rely on the input/output behavior of the design for the first n cycles.)

For an n-delay replacement, we express the flexibility to obtain a new design C such that C^n is exactly the same as D^n. Note that this is a conservative strategy since only $C^n \preceq D$ is needed.

The primary inputs and primary outputs to the design are denoted by $\vec{i}$ and $\vec{o}$, respectively, and the next-state and present-state variables are denoted by $\vec{y}$ and $\vec{x}$, respectively. To perform resynthesis for delay replaceability, we obtain a Boolean relation $\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ which describes the flexibility for replacement. We then use this relation to do multilevel resynthesis on our design.

First, we informally describe the flexibility that is specified later using a Boolean relation $\mathcal{V}$. We note that techniques for using a Boolean relation to do multilevel synthesis [Savoj and Brayton 1991] require the relation to be such that the starting design satisfies the relation. We construct $\mathcal{V}$ so that the behavior of the states in D^n is preserved. For $0 \leq i < n$, on any input, the relation allows a state in $(D^{i-1} \setminus D^i)$ to transition to any state in D^i. Clearly, the original design satisfies this flexibility relation. It is also the case that for any design C that satisfies the relation we have $C^n = D^n$; that is, after the first n clock cycles every state that C could possibly be in is equivalent to a state

in D^n. Also note that since we do not care about the outputs during the first n clock cycles, the outputs of the states in $D \setminus D^n$ can be chosen arbitrarily.

Formally, the Boolean relation $\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ that characterizes this flexibility is:

$$\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y}) = \sum_{j=0}^{n-1} [(\vec{x} \in D^j \setminus D^{j+1}) \wedge (\vec{y} \in D^{j+1})] + [(\vec{x} \in D^n) \wedge (\vec{y} = \delta_D(\vec{x}, \vec{i})) \wedge (\vec{o} = \lambda_D(\vec{x}, \vec{i}))]. \quad (1)$$

The intuition for the above relation $\mathcal{V}$ is that, given n, we choose to preserve the behavior of all states in the set D^n; that is, states in this set are forced to have the same output and next-state functions as in the original design D. For states outside D^n, if the state lies in $D^j \setminus D^{j+1}$, we allow the next state of such a state to be any state in D^{j+1}; we do not care about the output from this state. This ensures that n cycles after power-up, the new design is in a state in D^n, and thus is an n-delay replacement for the original design.

Note that for any integer $m \geq k$, the delayed design D^m is the same as D^∞, that is, the set of stable states. Thus the flexibility described by the relation $\mathcal{V}$ for m-delay replacement is the same as that for k-delay replacement. If we compute the number k for a design in advance, we know that we will not get any additional flexibility by allowing a slack greater than k.

4.1 Combinational Resynthesis Algorithms

The area of the hardware implementation of a design is strongly correlated to the total number of literals in the **factored form** [Brayton et al. 1990] representation of the functions at the internal nodes of the corresponding logic network. Thus minimizing the logic network with respect to the total literal count constitutes a powerful synthesis technique.

At any intermediate node of a logic network there is a local function $f_i : B^r \mapsto B$, where r is the number of fanins of the node. **Node simplification** is the process of optimizing a Boolean network by using don't cares in conjunction with a two-level minimizer [Brayton et al. 1984] to optimize the functions at the nodes. These don't cares arise in several ways.

(1) Because of the structure of the network, only a certain subset of B^r may be generated by assignments to the inputs. This gives rise to **satisfiability don't care** (SDC) points for f_i [Brayton et al. 1990].
(2) For certain input assignments, the values taken by the primary outputs of $\mathcal{N}$ may be independent of the function computed by a node; these are **observability don't care** points (ODC) for that node [Savoj et al. 1991].
(3) For certain input assignments the functionality of the node can be changed without destroying safe replaceability; this flexibility, as we have described in the preceding sections, leads to the **replaceability don't cares** points (RDC).

Let $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ be a Boolean relation expressing all the flexibility in the choice of combinational logic for a sequential circuit. Cerny and Marin [1977]

demonstrate a close relationship between optimizing a Boolean network with respect to a given Boolean relation, and computing observability don't care sets. The starting network $\mathcal{N}$ must satisfy the relation $\mathcal{R}$. The relation can be viewed as a single node with inputs $\vec{i}, \vec{x}, \vec{o}, \vec{y}$; this node is referred to as the **observability node**. Cerny and Marin [1977] and Savoj and Brayton [1991] show that composing this node with the network yields an **observability network** $\mathcal{N}'$. They prove that all the don't cares that can be used to optimize the nodes in $\mathcal{N}$ using the relation $\mathcal{R}$ are equal to the ODC of the same node in the network $\mathcal{N}'$.

In our context, the Boolean relation $\mathcal{V}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$, defined by Equation (1), plays the role of $\mathcal{R}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ in the previous paragraph. We use BDDs to represent the next-state functions, and relations on the state space. There is a variable associated with each primary input and each primary output; for each latch there is a present state variable and a next state variable. Let u be a node in the design for which the ODC is to be computed. We add a new BDD variable α_u corresponding to the output of u, and generate BDDs for the next state and output functions in terms of the primary inputs, latch outputs, and α_u. These are composed with the flexibility relation to obtain the BDD for the function computed by the observability network. Let $f(\vec{i}, \vec{x}, \alpha_u)$ be the output of the observability network; then it can be shown that the ODC for node u is given by $f_{\alpha_u}(\vec{i}, \vec{x}, \alpha_u) f_{\bar{\alpha_u}}(\vec{i}, \vec{x}, \alpha_u) + \bar{f}_{\alpha_u}(\vec{i}, \vec{x}, \alpha_u) \bar{f}_{\bar{\alpha_u}}(\vec{i}, \vec{x}, \alpha_u)$ [Savoj et al. 1991]. This is in terms of primary inputs; we then use the techniques of Savoj et al. [1991] to project this set into the space comprised of the fanins of the node. These are used in conjunction with a subset of the satisfiability don't care set to optimize the function at u.

In our experiments, we found that the BDD for $\mathcal{V}$ grew large. So, instead, we build a relation that is easier to build, but correctly expresses the flexibility for the states outside D^n:

$$\mathcal{U}(\vec{x}, \vec{y}) = \sum_{j=0}^{n-1} [(\vec{x} \in D^j \setminus D^{j+1}) \wedge (\vec{y} \in D^{j+1})] + [(\vec{x} \in D^n)].$$

Note that the relation $\mathcal{U}(\vec{x}, \vec{y})$ specifies the flexibility of the states inside D^n incorrectly (actually, it says that the states inside D^n can choose their next-state and outputs totally arbitrarily). We resolve this by restricting the don't cares to states outside D^n when we compute the don't cares for internal nodes in the network.

The final procedure for optimization is given in Figure 6. The image projection step in the algorithm in the figure can be done using a variety of algorithms; we use the algorithm presented in Touati et al. [1990].

Experiments

We implemented the procedure in Figure 6 in the SIS synthesis system [Sentovich et al. 1992], and performed experiments on the ISCAS89 benchmarks. We used BDDs to manipulate relations and sets. We focus on the area reduction for n-delay replacements, and report results for various values of n.

procedure delay-replacement (**input:** network in terms of $(\vec{i}, \vec{x}, \vec{o}, \vec{y})$, parameter n) {
  for (each node p in the network)
    Compute a BDD $\mathcal{S}_p(\vec{i}, \vec{x})$ representing node p in terms of $(\vec{i}, \vec{x})$
  $\mathcal{C}(\vec{x}) \leftarrow 1$
  $\mathcal{U}(\vec{x}, \vec{y}) \leftarrow 0$
  loop n **times** {
    $\mathcal{C}(\vec{y}) \leftarrow \exists \vec{x} \exists \vec{i} : [\mathcal{C}(\vec{x}) \cdot \prod_{j=1}^{t} (y_j \equiv \mathcal{S}_{y_j}(\vec{i}, \vec{x}))]$
    $\mathcal{C}(\vec{x}) \leftarrow \mathcal{C}(\vec{y})_{(\vec{y} \leftarrow \vec{x})}$
    $\mathcal{U}(\vec{x}, \vec{y}) \leftarrow \mathcal{U}(\vec{x}, \vec{y}) + \overline{\mathcal{C}(\vec{x})} \cdot \mathcal{C}(\vec{y})$
  }
  $\mathcal{U}(\vec{x}, \vec{y}) \leftarrow \mathcal{U}(\vec{x}, \vec{y}) + \mathcal{C}(\vec{x})$
  for (each node p in the network) {
    Add BDD variable p to the BDD variable list
    for (each node r in the network) {
      if (r is not in the transitive fanout of p)
        $\mathcal{S}_r(\vec{i}, \vec{x}, p) \leftarrow \mathcal{S}_r(\vec{i}, \vec{x})$
      else
        recompute $\mathcal{S}_r(\vec{i}, \vec{x}, p)$ representing node r in terms of $(\vec{i}, \vec{x}, p)$
    }
    $\mathcal{F}(\vec{i}, \vec{x}, p) = \exists \vec{o}, \vec{y} : [\mathcal{U}(\vec{x}, \vec{y}) \cdot \prod_{j=1}^{t} (y_j \equiv \mathcal{S}_{y_j}(\vec{i}, \vec{x}, p)) \cdot \prod_{k=1}^{s} (o_k \equiv \mathcal{S}_{o_k}(\vec{i}, \vec{x}, p))]$
    $\mathcal{G}(\vec{i}, \vec{x}) = \mathcal{F}(\vec{i}, \vec{x}, p)_{p=0} \cdot \mathcal{F}(\vec{i}, \vec{x}, p)_{p=1} + \overline{\mathcal{F}(\vec{i}, \vec{x}, p)_{p=0}} \cdot \overline{\mathcal{F}(\vec{i}, \vec{x}, p)_{p=1}}$
    $\mathcal{D}(\vec{i}, \vec{x}) = \mathcal{G}(\vec{i}, \vec{x}) \cdot \overline{\mathcal{C}(\vec{x})}$
    Image project $\overline{\mathcal{D}(\vec{i}, \vec{x})}$ to the space of local inputs of p
    Use the flexibility to simplify the local functionality of p
    Remove BDD variable p from the BDD variable list
    if (functionality of node p changed)
      for (each fanout node r of node p)
        recompute $\mathcal{S}_r(\vec{i}, \vec{x})$ representing node r in terms of $(\vec{i}, \vec{x})$.
  }
}

Fig. 6. Procedure to optimize a multilevel network for n-delay replacement. Here t is the number of latches, and s is the number of outputs.

In order to compare the optimizations due to safe replaceability with the known techniques for combinational synthesis, we report the optimizations (in reduction in number of literals) due to satisfiability don't cares (SDC), observability don't cares (ODC), and delay-safe replaceability don't cares separately. (We could have reported the amount of savings only due to the replaceability don't cares, but this would not have been fair since SDC and ODC are existing techniques and they also take less CPU time.)

The experimental results are shown in Table I. The starting circuits are ISCAS89 benchmark circuits that have been optimized with the SIS commands `(sweep; eliminate -1)` [Sentovich et al. 1992]. These eliminate single-input

Table I. Experimental Results for Delay Replacements[a]

Ckt.	Initial Size	Combinational Resynthesis SDC	Combinational Resynthesis ODC	n-Delay Replacements $n=1$ Final	$n=1$ Time	$n=2$ Final	$n=2$ Time	$n=5$ Final	$n=5$ Time	$n=\infty$ Final	$n=\infty$ Time
s27	12	12	0.05	12	0.04						
s298	150	130	12	130	0.95	123	1.01	113	1.00	107	1.56
s344	156	152	130	153	1.90	147	3.21	147	5.83	144	10.31
s349	160	155	152	154	1.97	148	3.38	148	5.95	145	10.63
s382	176	164	154	162	5.87	162	19.85	162	28.41	156	80.62
s386	204	197	187	127	1.19						
s400	184	166	162	160	6.13	160	21.17	160	29.75	154	83.53
s444	184	167	163	161	6.42	161	23.05	161	36.21	156	101.80
s510	280	279	279	279	2.94	279	2.95	279	2.96		
s526	283	242	240	240	3.68	238	5.78	237	5.98	188	35.89
s641	199	timeout		194	10.64						
s713	204	timeout		195	10.99						
s820	504	379	361	359	6.16						
s832	521	389	364	353	7.43						
s953	489	485	484	484	24.03						
s1196	618	608	600	600	95.95	600	103.82				
s1238	690	668	625	625	115.57	625	120.16				
s1488	813	759	755	697	12.33						
s1494	819	760	742	697	12.35						

[a]The initial size of the circuits and the final size are in the number of literals. Since we know that for any integers $m \geq k$, we would get the same results for both m- and k-delay replacements, the redundant experiments (denoted by blanks in the above table) were not performed.

and constant nodes and collapse nodes that do not fan out to more than one node.

We show the optimizations obtained by the SDC and the ODC. Then we show the results of the method presented in this section for n-delay replacements for $n = 1, 2, 5, \infty$. For the $n = \infty$ column, the value of n was less than 668 for all the examples. The table shows that, for many examples, significant additional optimizations are obtained by allowing power-up delay. Even for $n = 1$, we see good results for some examples, such as `s386`, `s1488`, `s1494`. Also, in most cases the CPU times for n-delay replacements are within an order of magnitude of the CPU time for combinational resynthesis.

We performed an experiment on one of the benchmark circuits (`s526`) to explore the trade-offs between flexibility and the power-up delay allowed. The results are in Table II. By allowing more delay n we do get additional flexibility. Also, the CPU times increase with higher values of n, partly because the time taken to compute the Boolean relation $\mathcal{U}$ goes up with higher n.

In the experiments corresponding to Table I, the initial nodes of the circuits were relatively small. Since we are minimizing the network one node at a time, very small nodes are unlikely to yield much optimization. We hypothesized that we might get better use of the delay-safe replaceability don't cares by using larger node sizes in the circuits. We executed the SIS command `sweep` followed by `eliminate 10` to partially collapse the network. We ran our algorithm for delay replacement on the resulting networks. Results are reported in Table III. Overall, we see our hypothesis validated.

Table II. Power-Up Delay/Flexibility Trade-Off for s526[a]

n	1	100	200	300	400	500	600	667
reduction	43	54	58	70	69	69	72	95
CPU time	3.68	11.52	14.55	17.33	20.38	22.51	24.37	36.27

[a]The initial size is 283 literals; reduction is in number of literals.

Table III. Experimental Results with Collapsed Nodes in the Starting Netlist

		n-Delay Replacements							
	Initial	$n=1$		$n=2$		$n=5$		$n=\infty$	
Ckt.	Size	Final	Time	Final	Time	Final	Time	Final	Time
s27	12	12	0.03						
s298	156	137	0.93	127	0.96	116	0.96	109	1.54
s344	168	155	1.83	150	3.10	150	5.61	148	9.92
s349	173	156	1.97	151	3.14	151	5.83	149	10.02
s382	204	164	5.59	164	17.55	164	26.53	160	80.32
s386	205	99	0.97						
s400	229	166	5.96	166	19.04	166	27.19	157	83.48
s444	236	169	5.99	169	19.99	169	31.96	166	91.49
s510	307	253	1.52	254	1.50	251	1.54		
s526	323	233	4.73	233	5.56	232	6.20	208	27.13
s641	234	207	14.11						
s713	285	202	14.97						
s820	468	331	3.84						
s832	470	334	3.94						
s953	700	590	21.50						
s1196	788	626	109.14	626	112.92				
s1238	882	636	170.87	636	178.53				
s1488	886	541	19.20						
s1494	896	524	20.08						

5. LATCH REMOVAL UNDER DELAY-SAFE REPLACEMENT

In Section 4 we described methods that optimize the output and next-state logic of sequential circuits under the safe replaceability and delay replaceability criteria. We now present an approach that removes latches while ensuring that the new circuit is a delay replacement of the original circuit.

The method for optimization presented in this section is inspired by sequential optimization techniques for circuits with a designated initial state [Berthet et al. 1990; Lin 1991; Berry and Touati 1993]. Since the circuit is always assumed to start in this designated initial state (we refer to this as the *DIS* assumption), we can arbitrarily change the behavior of the set of states that cannot be reached from this initial state without affecting the functionality of the circuit. In particular, it may be possible to express the value of a latch as a combinational function of other latch values. Since latches are relatively large and have the overhead of a clock signal, in such situations, heuristically, it is advantageous to replace the latch and its next-state logic by combinational logic.

Of course, as discussed in Section 2, we do not make the *DIS* assumption in this article. Nevertheless, we show how the synthesis technique with the *DIS* assumption inspires our solution. So we briefly review this technique in Section 5.1.

The replacement criteria (Definitions 5 and 8) do not require the original design and the replaced design to have an identical number of states. In fact, if we select tSCC of the original design, and reimplement it with a minimum length encoding, we can minimize the number of latches while preserving safe replaceability.

We decided not to use such an approach for the following reason. There is an analogous strategy for removing latches for circuits under the *DIS* assumption. There we compute the set of reachable states from the initial state, then reencode just this set of states with the minimum number of latches and produce a replacement design (discarding the structure of the original netlist in this process). Unfortunately, this strategy has been empirically observed [Villa] to produce much larger circuits than the original circuit reinforcing the belief that the starting multilevel netlists are a very good starting point, and if our synthesis procedure throws them away it becomes very hard to reconstruct as good a netlist just from the input–output functionality of a design. Thus for our synthesis methods we decided to start with the original multilevel netlist and make iterative modifications to it rather than extracting the functionality of the netlist and throwing away the original multilevel structure. This leads us back to the strategy of replacing a subset of the original latches with some combinational logic.

5.1 Removing Redundant Latches Under the *DIS* Assumption

We briefly review the techniques in Berthet et al. [1990], Lin [1991], and Berry and Touati [1993] for reducing the latches for circuits assuming DIS.

Given the initial state s, we extract the set of states reachable from s (using, e.g., the method of Lin et al. [1990]). Let $\mathcal{C}(\vec{x})$ be the characteristic function of the set of reachable states, where $\vec{x} = \{x_1, x_2, \ldots, x_t\}$ represent the t latches in the design. Since the states outside $\mathcal{C}(\vec{x})$ are not reachable we do not care about their behavior. This allows some latches to be replaced with combinational logic. For example, suppose among all the states in $\mathcal{C}(\vec{x})$, the value of the state bit x_j is a function of $(x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_t)$, Then we can replace latch x_j by that combinational function. This replacement is guaranteed not to affect the behavior of the states in $\mathcal{C}(\vec{x})$. Replacing a latch by combinational logic removes a latch as well as some combinational logic that was only used to drive this latch (see Figure 7).

The following conditions must be satisfied to allow latches $\{x_{t'+1}, \ldots, x_t\}$ to be replaced by Boolean functions $f_{t'+1}, \ldots, f_t$, defined over the latch variables $\{x_1, \ldots, x_{t'}\}$.

(1) *Latch redundancy condition*:

$$\forall i \in \{t'+1, \ldots, t\} : \forall x_i \exists x_{i+1} \exists x_{i+2} \cdots \exists x_t : [\mathcal{C}(\vec{x}) = 0]. \tag{2}$$

This ensures that for any $i \in \{t'+1, \ldots, t\}$, for every state in $\mathcal{C}(\vec{x})$, the value of latch x_i is uniquely determined by the value of latches $\{x_1, \ldots, x_{t'}\}$.

(2) *Function-set selection condition:*

$$\text{if } (a_1, \ldots, a_t) \in \mathcal{C}(\vec{x}) \text{ then } \forall i \in \{t'+1, \ldots, t\} : f_i(a_1, \ldots, a_{t'}) = a_i. \tag{3}$$

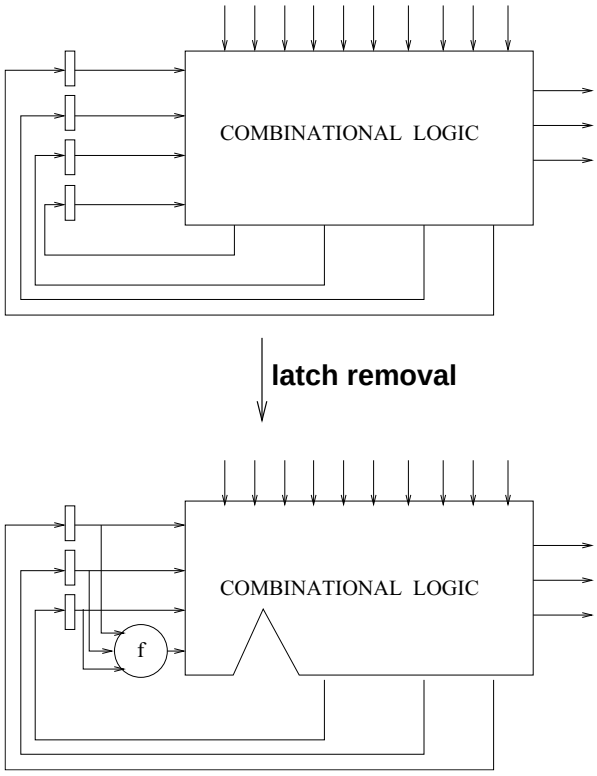

Fig. 7. Replacing a latch with function f.

This ensures that the function-set F is chosen so that for each state in $\mathcal{C}(\vec{x})$, the values of the replaced latches are correctly represented.

As an example, consider a design D whose STG is identical to that of D_0 as the one shown in Figure 4. Design D is implemented by three latches $\{x_1, x_2, x_3\}$ whose encodings are shown in the figure; D also has a designated state 000 (we emphasize that D is a different design than D_0 because it has an extra input line, the reset line, that, when asserted, sends the design to state 000). The set of reachable states from 000 is $\mathcal{C}(\vec{x}) = \{000, 011, 101\}$. Using Conditions 2 and 3 above, we can replace latch x_3 with the function $f_3 = x_1 + x_2$, and we get design C whose STG is identical to that of design A shown in Figure 8; the new designated initial state of design C is 00. States 00, 01, and 10 are, respectively, equivalent to states 000, 011, and 101 in D. State 11 is not equivalent to any state in D_0 but its behavior is immaterial because the new design always starts in state 00, and can never reach state 11.

5.2 Removing Redundant Latches Without the *DIS* Assumption

We now look at the naive extension of the algorithm in the previous section, and see why it does not work if the design can power up in any state.

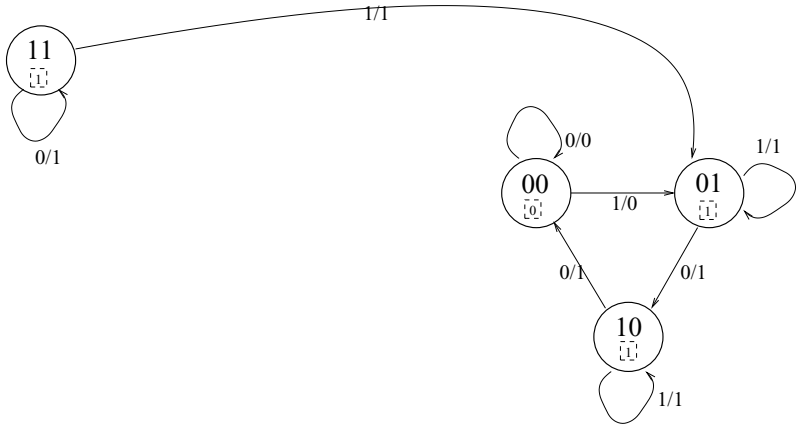

Fig. 8. Design A. The value of $f_3 = x_1 + x_2$ is shown in the dotted box.

We find it convenient to refer to a set of states that is closed under application of all inputs as a *core* of the design.

Without the *DIS* assumption, the set of states in any core represents the desirable steady-state behavior of a design. So a naive strategy for removing redundant latches will be to make any core set of states of a design play the role of $\mathcal{C}(\vec{x})$ in the previous section and then use exactly the same strategy, with Conditions 2 and 3, for redundant latch identification and removal. Thus we may obtain design A (shown in Figure 8) as a safe replacement (a 0-delay replacement) for design D_0 in Figure 4. However, A is not an n-delay replacement of D_0 for any n. This is because $A^n = A$ and $A \not\preceq D_0$. Also observe that D_0 can be reset from any state to state 000 under input sequence $0 \cdot 0$, but the same input sequence does not reset A even if we wait for an arbitrary number of clock cycles after power-up (because A can remain in state 11 arbitrary many cycles). Thus the behavior of the states outside the core must be controlled to make sure that the new design is an n-delay replacement.

Let $\mathcal{C}(\vec{x})$ be a core that we are going to use for delay replacement on the original design D. There are certain factors that determine the behavior of the new design A. First, the selection of latches to be replaced, $\hat{x} = (x_{t'+1}, \ldots, x_t)$ partitions the 2^t states in design D into $2^{t'}$ equivalence classes, each with $2^{t-t'}$ states. Two states $\vec{a} = (a_1, \ldots, a_t)$ and $\vec{b} = (b_1, \ldots, b_t)$ lie in the same equivalence class if and only if for all $i \in \{1, \ldots, t'\}$: $a_i = b_i$. We denote the equivalence class of $\vec{a}$ by $\Omega(\vec{a})$. Each state of A represents an equivalence class in the original design; that is, $\tilde{a} = (a_1, \ldots, a_{t'})$ represents $\Omega(\vec{a})$. Second, the selection of the function-set $F = (f_{t'+1}, \ldots, f_t)$ determines the representative state (of D) in each equivalence class, and the representative state determines the behavior of this class in the new design. Thus if $f_{t'+1}(\tilde{a}) = c_1$, $f_{t'+2}(\tilde{a}) = c_2$, $f_t(\tilde{a}) = c_{t-t'}$, then $(a_1, \ldots, a_{t'}, c_1, \ldots c_{t-t'})$ is the representative state of the class $\tilde{a}$ (or $\Omega(\vec{a})$). Let $\phi(\tilde{a})$ denote the representative state of the class $\tilde{a}$. Now, for input $\vec{i}$, the next state of $\tilde{a}$ in A is $\delta_A(\tilde{a}, \vec{i}) = \Omega(\delta_D(\phi(\tilde{a}), \vec{i}))$ and the output is $\lambda_A(\tilde{a}, \vec{i}) = \lambda_D(\phi(\tilde{a}), \vec{i})$.

Note that the set of replaced latches $(x_{t'+1}, \ldots, x_t)$ must satisfy the latch redundancy condition in Equation (2), where $\mathcal{C}(\vec{x})$ is the core. Also the selection

of the function-set F must satisfy the function-set selection condition in Equation (3). Based on the derivation of the behavior of the new design, discussed above, it can be seen that if we replace latch x_3 in design D_0 of Figure 4 with the function-set $F = (f_1)$ such that $f_1 = x_1 + x_2$ we get the new design shown in Figure 8 (e.g., state 11 goes to state 11 on input 0 because $\delta_A(11, 0) = \Omega(\delta_{D_0}(\phi(11), 0)) = \Omega(\delta_{D_0}(111, 0)) = \Omega(110) = 11$). For this example, the representative elements can be read off by using the number in the dotted box in Figure 8; for example, $\phi(\Omega(100)) = \phi(\Omega(101)) = \phi(10) = 101$.

However, as previously discussed, the function replacement $F = (x_1 + x_2)$ is not an n-delay replacement for D_0. Thus we also need to regulate the behavior of equivalence classes that do not contain any state in the core of the original design. Note that each state in the core is the representative element of its class, and the behavior of this class in the new design is equivalent to that of the core state in the original design.

As in Section 4 we satisfy the requirement for n-delay replacement by ensuring, if we wait for n clock cycles with arbitrary input vectors, we reach a state inside the core. The procedure we describe next will guarantee this and thus we obtain an n-delayed replacement.

Choices for the Function-Set F.

Definition 9. Given a set of states S in $\{0, 1\}^t$ (i.e., a set of states of the original design), a function-set $F = (f_{t'+1}, \ldots, f_t)$ is *compatible* with S if for any vector $\tilde{a} = (a_1, \ldots, a_{t'})$, $(a_1, \ldots, a_{t'}, f_{t'+1}(\tilde{a}), \ldots, f_t(\tilde{a})) \in S$.

First assume that we have chosen the core set of states and a set of latches $(x_{t'+1}, \ldots, x_n)$ to be replaced (so that the condition in Equation (2) is satisfied). Thus we already have a partition of equivalence classes for states in $\{0, 1\}^t$. Now we select a set of states S in $\{0, 1\}^t$, and derive a function-set F compatible with S. We use the procedure in Figure 9 to obtain S (which is denoted by its characteristic function $\mathcal{S}(\vec{x})$). States in set S are candidates for representative elements of their respective equivalence classes. Each equivalence class has at least one state in S; each equivalence class whose state is in the core has exactly one state in S.

THEOREM 5.1. *If the procedure "compatible-set" returns a set $\mathcal{S}(\vec{x})$, then there exists at least one function-set $F = (f_{t'+1}, \ldots, f_t) : \{0, 1\}^{t'} \mapsto \{0, 1\}^{t-t'}$ compatible with $\mathcal{S}$. Furthermore, if we replace latches $(x_{t'+1}, \ldots, x_t)$ by F, then the new design is an n-delay replacement of the original design.*

PROOF. First notice that the set $\mathcal{P}(\vec{x})$ is the set of all equivalence classes that have at least one member in $\mathcal{S}(\vec{x})$. Since the procedure returns a set $\mathcal{S}(\vec{x})$ only when $\mathcal{P}(\vec{x}) = 1$ we know that there exists at least one function-set that is compatible with $\mathcal{S}$.

Now observe that if a state from an equivalence class is added to $\mathcal{S}(\vec{x})$ in iteration $j = p$, then for any iteration $i > p$, no state from this equivalence class is added to $\mathcal{S}(\vec{x})$. If one or more states from some equivalence classes are added in iteration $j = p$, we say that this equivalence class is covered in the pth iteration. Suppose we obtain a new design by choosing a function-set that

```
procedure compatible-set (input: C(x), {x_{t'+1}, ..., x_t}, T(x, i, y), n) {
    j ← 0
    S(x) ← C(x)
    P(x) ← ∃x_{t'+1} ∃x_{t'+2} ··· ∃x_t : [S(x)]
    while ((j < n) and (P(x) ≠ 1)) {
        j ← j + 1
        P(y) ← (P(x))_{(y ← x)}
        R(x) ← ∀i ∃y : [T(x, i, y) ∧ P(y)]
        if (R(x) · ¬P(x) = 0) goto end;
        S(x) ← S(x) + R(x) · ¬P(x)
        P(x) ← ∃x_{t'+1} ∃x_{t'+2} ··· ∃x_t : [S(x)]
    }
end: if (P(x) = 1) return S(x)
    else return FAIL
}
```

Fig. 9. Procedure to optimize $\mathcal{S}(\vec{x})$ (input $\mathcal{C}(\vec{x})$ is the core set of states, $\mathcal{T}(\vec{x}, \vec{i}, \vec{y})$ is the transition relation of the design, and n is the delay parameter for n-delay replacement).

is compatible with $\mathcal{S}(\vec{x})$. Now consider any state in the new design. This state $\tilde{a}$ represents an equivalence class in the old design and derives its behavior from the representative state $\vec{a}$ of that equivalence class. Let this equivalence class be covered in the qth iteration; thus $\vec{a}$ was added to $\mathcal{S}(\vec{x})$ in iteration $j = q$. Thus $\vec{a} \in \mathcal{R}(\vec{x})$ in iteration $j = q$, and for any input, the next state of $\vec{a}$ lies in an equivalence class that was covered in the rth iteration, for some $r < q$. Thus in the new design, for any input vector, state $\tilde{a}$ goes to some state that represents an equivalence class that was covered in an earlier iteration. Since we have at most $j = n$ iterations, it is clear that for any state $\tilde{a}$ in the new design, if we apply any arbitrary input vectors for n steps, we will reach a state $\tilde{b}$ representing an equivalence class that was covered in iteration $j = 0$. Since iteration $j = 0$ only covers the the equivalence classes for the core states, $\tilde{b}$ is equivalent to a state in the core of the old design. Thus the new design is an n-delay replacement of the old one. □

As an example, start with design D_0 in Figure 4 and choose the set {000, 011, 101} as the core and latch x_3 to be replaced. The procedure *compatible-set* returns the set $\mathcal{S}(\vec{x}) = \{000, 011, 101, 110\}$. $F = (f_3)$ where $f_3 = x_1 \oplus x_2$ is the only function-set compatible with this. The resulting design B (shown in Figure 10) is a one-delay replacement of design D_0.

Selecting an Optimal Function-Set F. If the procedure *compatible-set* returns a set S, any function-set F compatible with S is allowed. However, our overall goal is to optimize the area of the original circuit, so we would like to choose an F that has the smallest area. In fact, if we cannot find F within a certain bound, the savings gained by removing the redundant latches may actually be lost (refer once again to Figure 7 for the trade-off in area saving).

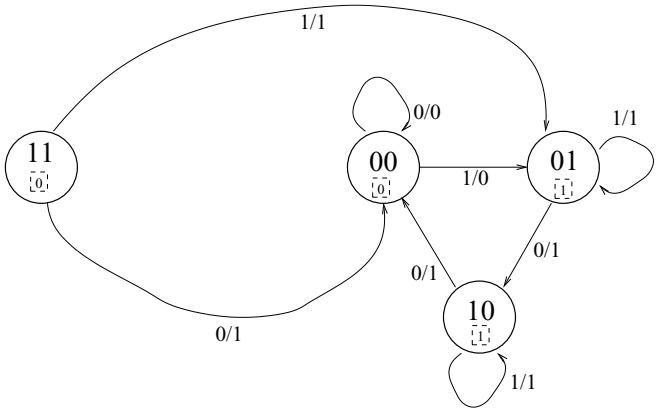

Fig. 10. Design B. The value of $f_3 = x_1 \oplus x_2$ is shown in the dotted box.

Selecting the function-set F is equivalent to determining a multilevel network with t' inputs and $(t - t')$ outputs so that the network is compatible with $S(\tilde{x}, \hat{x})$ where $\tilde{x} = \{x_1, \ldots, x_{t'}\}$ and $\hat{x} = \{x_{t'+1}, \ldots, x_t\}$. Note that this is the problem of obtaining a multilevel network compatible with a Boolean relation in terms of its inputs and outputs. This can be solved in two steps. We obtain any arbitrary multilevel network representing some F compatible with S. Then one approach would be to use the techniques in [Savoj and Brayton 1991] (as we used in Section 4) to optimize this network, while maintaining compatibility with S.

However, our experiments (presented later in this section) show that the size of F is relatively small compared to the given circuit sizes and that the approach described by procedure *greedy-function-set* in Figure 11 worked well. We order the latches arbitrarily. For each latch to be replaced, we find the on-set and don't care set for the replacement function. Then we use a heuristic to find a function compatible with this on-set and don't care set. Since all the Boolean quantities are represented as BDDs, we use a BDD minimization procedure, which heuristically finds an implementation that has the smallest support (we use a simpler version of the algorithm in Lin [1993]); alternately, we could have used *bdd-generalized-cofactor* [Coudert and Madre 1990] or any of many other BDD minimization algorithms with respect to a don't care set [Shiple 1994].

Since we are dealing with very small functions, in our application, the choice of the heuristic does not matter. The minimized BDD is converted to a similar looking network whose nodes are multiplexers controlled by the variable of the corresponding BDD node. Once we find a function replacement for a latch we restrict the compatible set S by this function.

Choosing the Core. Before selecting the function-set F, we need to choose a core to identify a set of latches that satisfies the condition in Equation (2) so that we can remove these latches. Obviously, if a set of latches satisfies the condition in Equation (2) for a choice of core, it also satisfies the condition for any subset of the core. As argued earlier, most real designs have a single tSCC,

```
procedure greedy-function-set (input: S(x̃, x̂), {x_1, ..., x_t'}, {x_t'+1, ..., x_t}) {
    for j = t' to t do {
        O(x̃) ← ∃x_t'+1 ∃x_t'+2 ··· ∃x_t : [S(x̃, x̂) · (x_i = 1)]
        /* O(x̃) is the on set for f_j */
        D(x̃) ← ∀x_j ∃x_t'+1 ∃x_t'+2 ··· ∃x_j−1 ∃x_j+1 ∃x_j+2 ... x_t : [S(x̃, x̂)]
        /* D(x̃) is the dont-care set for f_j */
        f_j(x̃) ← bdd-minimize(O(x̃), D(x̃))
        S(x̃, x̂) ← S(x̃, x̂) · (x_j = f_j(x̃))
    }
    F ← (f_t'+1, ..., f_t)
    return F
}
```

Fig. 11. Procedure to obtain a function-set.

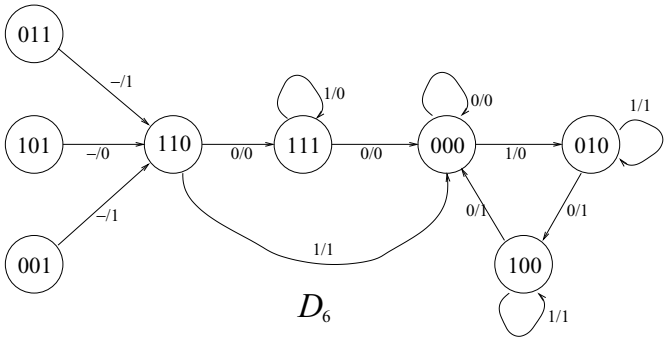

Fig. 12. Design D_6.

and for such designs each choice of the core must be a superset of the tSCC. Thus the tSCC is the obvious choice since it allows us the most flexibility in choosing the set of redundant latches. However, this choice may not be the right one for the following reason.

Consider design D_6 in Figure 12. Suppose we choose the tSCC {000, 010, 100} as the core and the third latch x_3 as redundant. The procedure *compatible-set* identifies candidates to be representatives for their equivalence classes (000, 010, and 100 are already representatives for their respective equivalence classes 00, 01, and 10; we need to find candidates to represent the equivalence class 11). We observe that for all inputs, neither of the states 110 or 111 goes to equivalence class 00, 10, or 01 (the value of $\mathcal{P}(\vec{x})$ at iteration $j = 0$). This means that $(\mathcal{R}(\vec{x}) \cdot \overline{\mathcal{P}(\vec{x})} = 0)$ and the procedure returns FAIL. Thus we are unable to replace latch x_3. On the other hand, if we choose D_6^{∞} {000, 010, 100, 111}, latch x_3 could be replaced by the function $f_3 = x_1 \cdot x_2$ since the procedure *compatible-set* returns $\mathcal{S}(\vec{x}) = \{000, 010, 100, 111\}$. In fact,

we can prove that if we choose the set D^∞ of a design D as the core, the procedure *compatible-set* never returns FAIL for any $n \geq m$, where m is such that $D^\infty = D^m$.

THEOREM 5.2. *If we set $\mathcal{C}(\vec{x})$ to be the set of states in D^k, the k-cycle delayed design, and set n to be equal to k, then the procedure "compatible-set" does not return FAIL.*

PROOF. We prove the above by showing that after the **while** loop terminates, $\mathcal{P}(\vec{x}) = 1$. We show that by proving the following claim. In the following, we say that state $\vec{a}$ lies in the lth *onion ring* if and only if $\vec{a} \in Q_{D^{k-l}} \setminus Q_{D^{k-l+1}}$ for $l \in \{1, 2, \ldots, k\}$; if $\vec{a} \in Q_{D^k}$, we say that $\vec{a}$ lies in the 0th onion ring. It should be obvious that for any l such that $0 < l \leq k$, for any arbitrary input vector, any state in the lth onion ring goes to a state that lies in the pth onion ring for some $p < l$.

Claim: If state $\vec{a}(a_1, \ldots, a_t)$ belongs to the lth onion ring, then after iteration $j = l$, $\mathcal{P}(\vec{x})$ contains $\tilde{a} = (a_1, \ldots, a_{t'})$.
We prove this claim by induction.

Base case. ($l = 0$): Since $\vec{a}$ belongs to D^k, it belongs to $\mathcal{C}(\vec{x})$ and hence $\tilde{a}$ belongs to $\mathcal{P}(\vec{x})$ when it is initialized at $j = 0$.

Induction step. Assume that the claim is true for any state that lies in the pth onion-ring for some $p < l$. We prove the induction step for a state $\vec{a}$ that belongs to the lth onion ring. Consider iteration $j = l$. Either $\mathcal{P}(\vec{x})$ already contains $\tilde{a}$ before this iteration, in which case we are done, or else, $\mathcal{P}(\vec{x})$ does not contain $\tilde{a}$ before this iteration. However, we know that for any arbitrary input vector, $\vec{a}$ transitions to a state $\vec{b}$ that lies in the pth onion ring for some $p < l$. By our induction hypothesis, $\tilde{b}$ belongs to $\mathcal{P}(\vec{x})$ from the previous iteration. Thus $\vec{a}$ must belong to the set $\mathcal{R}(\vec{x})$ computed at $j = l$. Hence $\tilde{a}$ belongs to $\mathcal{P}(\vec{x})$ after iteration $j = l$. □

This theorem shows that if the core is chosen as the set D^∞ and there is a nonempty set of candidate redundant latches, then those latches can be removed. On the other hand, it is relatively straightforward to construct examples in which choosing the tSCC as the core lets us replace a latch, whereas no latch can be removed if the set D^∞ is chosen.

In our experiments, we first chose the tSCC as the core (we use a procedure described in Singhal et al. [1996] to obtain the tSCC). If we were not able to replace any latches, we then chose the set D^∞ as the core. In fact, as we show in the experimental results, for all except two examples, we were able to successfully use the tSCC as the core. For `s344` and `s349` (see Table IV), *compatible-set* returned FAIL in the case where we used the tSCC as the core; however, when we used D^∞ as the core, we were still unable to remove any latches; that is, no set of latches satisfied the condition in Equation (2).

Selecting the Set of Redundant Latches. Given a set $\mathcal{C}(\vec{x})$, Lin [1991] presented a heuristic for selecting a latch ordering to test if a latch is

Table IV. Experimental Results for Latch Replacement[a]

Ckt.	Orig. Size	Final Circuit: Latches Removed	Delay n	Size of F	Total Size	Time
s298	150	2	7	4	130	1.26
s344	156	0	—	—	156	5.20
s349	160	0	—	—	160	5.33
s382	176	3	101	6	160	35.08
s386	204	0	—	—	204	0.21
s400	184	3	101	6	166	33.32
s444	184	3	101	6	166	33.09
s526	283	2	667	4	263	44.63
s641	199	5	2	9	172	3.81
s713	204	5	2	9	175	4.33
s953	489	9	1	96	522	12.77
s1196	618	0	—	—	618	11.80
s1238	690	0	—	—	690	9.14

[a]For s344 and s349, *compatible-set* returned FAIL when passed the tSCC.

redundant to obtain the set of redundant latches to select the set of latches $\hat{x} = \{x_{t'+1}, \ldots, x_t\}$ that satisfy the latch redundancy condition (Equation (2)). The heuristic maximizes the number of latches chosen as redundant by ordering latches by decreasing unateness. The unateness of a variable x_i is the absolute value of the difference between the number of minterms in $\mathcal{C}(\vec{x})_{x_i=0}$ and $\mathcal{C}(\vec{x})_{x_i=1}$. The intuition is that the more unate a variable is, the less it contributes to distinguishing the states in $\mathcal{C}(\vec{x})$. We use this heuristic and then select the set of redundant latches by going through this order and adding a variable to the set if the current set still satisfies Equation (2).

5.3 Experiments

Our entire algorithm for replacing latches with combinational logic is shown in Figure 13.

We used ∞ as the value of n to pass to *compatible-set* because in all examples, the **while** loop terminated in less than 1000 iterations. Thus we obtain n-delay replacements, where $n < 1000$. It is usually safe to assume that more than 1000 cycles are allowed before circuit operation starts (e.g., for a 1 GHz design, 1000 clock cycles amount to 0.001 ms).

We implemented this algorithm in SIS and experimented with the same ISCAS89 circuits that we used in Section 4. We preprocessed the circuits with the same commands as before: `sweep; eliminate -1`. We chose the subset of circuits from our previous experiments where the number of states in the tSCC were at most half the total number of states. The results appear in Table IV. We see from Table IV that for most of the circuits we tried, we were able to remove some latches. Also, the size of the function-set F was usually very small (the exception being s953). As a result, the total final size of the combinational part of the circuit was smaller than the original size because the addition of F was more than offset by the removal of the fanin logic of the latches (Figure 7).

```
procedure latch-replacement (input: network in terms of (i⃗, x⃗, o⃗, y⃗)) {
    /* let y_j denote the BDD for the j-th next state variable */
    /* let o_k denote the BDD for the k-th output variable */
    for (each node p in the network)
        Compute a BDD S_p(i⃗, x⃗) representing node p in terms of (i⃗, x⃗)
    T(i⃗, x⃗, y⃗) ← (∏_{j=1}^{t} (y_j ≡ S_{y_j}(i⃗, x⃗))) · (∏_{k=1}^{n} (o_k ≡ S_{o_k}(i⃗, x⃗)))
    C(x⃗) ← tSCC(T(i⃗, x⃗, y⃗))
    x̂ ← ∅; B(x⃗) ← C(x⃗)
    foreach (latch x) do
        if (∀x : [B(x⃗)] = 0) {
            x̂ ← x̂ ∪ {x}
            B(x⃗) ← ∃x : [B(x⃗)]
        }
    x̃ ← x⃗ \ x̂
    A(x⃗) ← compatible-set(C(x⃗), x̂, T(i⃗, x⃗, y⃗), ∞)
    Let n ← j, if the loop in compatible-set() terminates in the j-th iteration
    F ← greedy-function-set(A(x⃗), x̃, x̂)
    Replace latches x̂ by an implementation of F
    Recursively sweep away dead logic which fans out to nowhere
    return (optimized network, n)
}
```

Fig. 13. Procedure to replace latches with logic.

So far we have removed latches and replaced them by a combinational function of other latches. It seems very conceivable that such an operation would increase the delay of the paths between the latches. This intuition is based on the observation that a sequential path between two latches in the original design may become a purely combinational path due to the replacement of the latch on the path; thus there might be combinational paths in the new designs that are longer than all combinational paths in the original design. To test this hypothesis that our latch replacement algorithm achieves area optimization only at the expense of performance, we performed the following experiment. We ran the SIS script `script.rugged`[1] to do technology-independent optimization, followed by a technology mapping step using the MCNC91 library. The results appear in Table V.

We see that, for this *subset* of circuits, we get an average of 10.4% (and a maximum of 18.0%) improvement in mapped area and even an average of 3.7% (and a maximum of 23.6%) improvement in the length of the topologically longest path, which goes against the intuition that latch replacement will increase the delay of the circuit. Note that for `s953` the reduction in latch count more than makes up for the fact that the replacement combinational logic was larger than that of the original circuit (cf. Table IV).

[1]In the last step of `script.rugged`, we used the sequential optimization procedure described in Section 4 instead of `full_simplify` because we wanted to exploit the sequential flexibility in addition to the combinational flexibility.

Table V. Latch Replacement Results After Technology Mapping

	With Latch Replacement			No Latch Replacement			Ratios	
Ckt.	Mapped Area	Delay	CPU Time	Mapped Area	Delay	CPU Time	Area	Delay
s298	278	17.6	8.49	312	19.1	8.83	0.891	0.921
s382	428	21.6	55.62	472	21.6	168.30	0.907	1.000
s400	419	27.0	53.76	460	26.9	162.71	0.911	1.004
s444	419	27.0	57.10	458	27.0	178.67	0.915	1.000
s526	456	22.4	108.22	460	18.0	73.24	0.991	1.245
s641	378	28.1	30.29	460	36.5	34.91	0.822	0.770
s713	377	27.9	31.10	460	36.5	42.04	0.820	0.764
s953	940	22.8	70.68	1012	20.8	75.85	0.929	1.096

The only circuit where latch replacement significantly increases delay is `s526`. Note that the CPU time with latch replacement includes the time taken to remove redundant latches. It is worth noting that the total CPU time with latch removal is actually lower than the total CPU time without this option! This is probably because logic optimization becomes simpler once we have fewer latches (and thus fewer state variables). In particular, reachability computations become much faster.

Overall, for this set of circuits, we have demonstrated that replacing redundant latches reduces the number of latches, the size of mapped circuits, and even the delay for these circuits.

5.4 Final Results

Here we analyze the combined effect of the algorithms in the previous two sections. We wish to compare technology-mapped circuits after using our optimizations versus mapped circuits obtained using existing combinational optimization techniques. For the experiment we first remove redundant latches by using the algorithm in Section 5. Then we run the SIS script `script.rugged` on this circuit followed by sequential optimization as described in Section 4. At this point we map the circuit to the MCNC91 library of gates and latches. We compare this mapped circuit against the circuit we would get by just running `script.rugged` followed by the technology mapping. The results appear in Table VI.

These experiments show that we get an average of 7.0% (and a maximum of 19.2%) area optimization over all circuits. More surprisingly, we also see an average improvement of 4.5% (and a maximum of 23.6%) in the delay of the circuits (as reported by the `print_delay` command in SIS), even though this was not our targeted objective. The CPU times are reasonable compared to those for the existing combinational optimization routines in SIS, and we can expect that the times can be reduced further with a careful tuning of the code.

6. CONCLUSIONS

We discussed design replacement for an arbitrary design in the absence of any knowledge of its environment, and reviewed our notion of safe and delay-safe

Table VI. Comparing Sequential Optimization Under Delay Replaceability with Purely Combinational Optimization[a]

	With Delay Replacement			Pure Combinational Optimization			Ratios	
Ckt.	Mapped Area	Delay	CPU Time	Mapped Area	Delay	CPU Time	Area	Delay
s27	48	10.9	1.3	48	10.9	1.1	1.000	1.000
s298	278	17.6	8.5	344	8.2	18.8	0.808	0.936
s344	355	28.8	31.2	373	33.4	9.2	0.952	0.862
s349	357	28.8	31.6	377	33.4	9.4	0.947	0.862
s382	428	21.6	55.9	472	20.8	10.9	0.907	1.038
s386	262	27.2	9.5	283	27.5	8.1	0.926	0.989
s400	419	27.0	53.8	461	26.9	10.4	0.909	1.004
s444	419	27.0	57.1	462	28.5	10.5	0.907	0.947
s510	513	28.5	42.1	493	30.7	23.2	1.041	0.928
s526	456	22.4	108.2	514	18.6	15.0	0.887	1.204
s641	378	28.1	30.3	460	36.5	12.9	0.822	0.770
s713	377	27.9	31.1	460	36.5	13.3	0.820	0.764
s820	581	19.9	79.7	573	20.8	38.5	1.014	0.957
s832	535	21.7	80.0	544	21.4	32.7	0.983	1.014
s953	940	22.8	70.7	1030	22.0	43.7	0.913	1.036
s1196	1152	29.3	403.7	1172	29.4	195.0	0.982	0.997
s1238	1131	29.4	342.6	1140	32.6	102.3	0.992	0.902
s1488	1173	18.9	299.1	1208	19.0	94.2	0.971	0.995
s1494	1139	20.1	209.4	1216	19.5	112.8	0.937	1.031

[a]Results after technology mapping.

replaceability. We showed how to perform optimization on the next-state and output logic, as well as on the latches, while guaranteeing delay-safe replaceability. Our experimental results (Table VI) indicate that we can exploit the flexibility afforded to us by the replaceability criteria to achieve significant optimization.

It should be stressed that our entire approach is based on the notions of safe and delay-safe replacement. Our reasons for doing so were spelled out in Section 1. However, it may be possible to perform more optimizations if additional information about the environment were known, for example, an initializing sequence.

Our optimization algorithms operate on the state transition graphs (even though they do so implicitly, using BDDs). This limits the size of circuits we can handle; in our experiments, we limited ourselves to designs with less than 30 latches. Certainly, we would not recommend even an optimized implementation of our algorithm on designs with more than several hundred latches. However, the power of our approach is in the condition of safe replacement. Since we make replacement without assuming anything about the environment, we envision a more productive use of our optimization algorithm in scenarios where reasonably sized designs are cut out of larger designs and safe replacements are made on them. This would work with manual as well as automatic partitioning. Our algorithms combined with powerful automatic partitioning methods and applied to a set of larger designs is a problem worth investigating. It would also be very useful to come up with other optimization algorithms for safe replacement, especially those that are more “structural”

in nature and do not need state transition graphs. Starting points include ATPG [Kunz and Pradhan 1994; Mehrotra et al. 1997] and SAT [Silva and Sakallah 1996].

One more area that needs investigation is state encoding and implementation of designs which are specified at the behavioral level (i.e., as STGs). The traditional method of obtaining such an implementation relies on the fact that the design has a designated initial state and thus the behavior of encoded states that are unreachable from the designated initial state is not important. However, without the designated initial state assumption, all 2^t encodings of an implementation (which has t latches) must satisfy the safe replacement condition with respect to the given behavioral-level specification. This will add additional constraints to the traditional problem of state encoding and implementation (see Saldanha et al. [1994] for the traditional formulation).

REFERENCES

BERRY, G. AND TOUATI, H. J. 1993. Optimized controller synthesis using Esterel. In *Workshop Notes of International Workshop on Logic Synthesis* (Tahoe City, CA, May).

BERTHET, C., COUDERT, O., AND MADRE, J. C. 1990. New Ideas on symbolic manipulation of finite state machines. In *Proceedings of the International Conference on Computer Design* (Cambridge, MA, October) 224–227.

BRAYTON, R. K., HACHTEL, G. D., MCMULLEN, C. T., AND SANGIOVANNI-VINCENTELLI, A. L. 1984. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic, Hingham, MA.

BRAYTON, R. K., HACHTEL, G. D., AND SANGIOVANNI-VINCENTELLI, A. L. 1990. Multilevel logic synthesis. *Proc. IEEE 78*, 2 (Feb.), 264–300.

CERNY, E. AND MARIN, M. A. 1977. An approach to unified methodology of combinational switching circuits. *IEEE Trans. Comput. 27*, 8.

CHENG, K.-T. 1993. Redundancy removal for sequential circuits without reset states. *IEEE Trans. Comput.-Aided Des. Integ. Circ. 12*, 1 (Jan.), 13–24.

CHO, H., HACHTEL, G. D., JEONG, S.-W., PLESSIER, B., SCHWARZ, E., AND SOMENZI, F. 1990. ATPG aspects of FSM verification. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November), 134–137.

COUDERT, O. AND MADRE, J. C. 1990. A Unified framework for the formal verification of sequential circuits. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November), 126–129.

ENTRENA, L. AND CHENG, K.-T. 1993. Sequential logic optimization by redundancy addition and removal. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November), 310–315.

HARTMANIS, J. AND STEARNS, R. E. 1966. *Algebraic Structure Theory of Sequential Machines*. Intl. Series in Applied Mathematics. Prentice-Hall, Englewood Cliffs, NJ.

KUNZ, W. AND PRADHAN, D. 1994. Recursive learning: A precise implication procedure and its application to test verification and and optimization. *IEEE Trans. Comput. Aided Des. Integ. Circ. Syst.* (Sept.).

LIN, B. 1991. Synthesis of VLSI design with symbolic techniques. PhD Thesis, Electronics Research Laboratory, University of California, Berkeley, November. Memorandum No. UCB/ERL M91/105.

LIN, B. 1993. Efficient symbolic support manipulation. In *Proceedings of the International Conference on Computer Design* (Cambridge, MA, October), 513–516.

LIN, B., TOUATI, H. J., AND NEWTON, A. R. 1990. Don't care minimization of multi-level sequential logic networks. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November), 414–417.

MEHROTRA, A., QADEER, S., SINGHAL, V., AZIZ, A., BRAYTON, R., AND SANGIOVANNI-VINCENTELLI, A. 1997. Sequential optimisation without state space exploration. In *Proceedings of the International Conference on Computer-Aided Design* (November), 208–215.

PIXLEY, C. 1990. A computational theory and implementation of sequential hardware equivalence. In *Proceedings of the Workshop on Computer-Aided Verification*, vol. 3 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, E. M. Clarke and R. P. Kurshan, Eds. American Mathematical Society, June, 293–320.

PIXLEY, C. 1992. A theory and implementation of sequential hardware equivalence. *IEEE Trans. Comput.-Aided Des. Integ. Circ. 11*, 12 (Dec.), 1469–1494.

PIXLEY, C., SINGHAL, V., AZIZ, A., AND BRAYTON, R. 1994. Multi-level synthesis for safe replaceability. In *Proceedings of the International Conference on Computer-Aided Design* (November), 442–449.

POMERANZ, I. AND REDDY, S. M. 1993. Classification of faults in synchronous sequential circuits. *IEEE Trans. Comput. 42*, 9 (Sept.), 1066–1077.

POMERANZ, I. AND REDDY, S. M. 1996. On removing redundancies from synchronous sequential circuits with synchronizing sequences. *IEEE Trans. Comput. 45*, 1, 20–32.

QADEER, S., SINGHAL, V., PIXLEY, C., AND BRAYTON, R. K. 1996. Latch redundancy removal without global reset. In *International Conference on Computer Design* (October).

SALDANHA, A., VILLA, T., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1994. Satisfaction of input and output encoding constraints. *IEEE Trans. Comput. Aided Des. Integ. Circ. 13*, 5 (May), 589–602.

SAVOJ, H. AND BRAYTON, R. K. 1991. Observability relations and observability don't cares. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November), 518–521.

SAVOJ, H., BRAYTON, R. K., AND TOUATI, H. 1991. Extracting local don't cares for network optimization. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November), 514–517.

SENTOVICH, E. M., SINGH, K. J., MOON, C., SAVOJ, H., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1992. Sequential circuit design using synthesis and optimization. In *Proceedings of the International Conference on Computer Design* (Cambridge, MA, October), 328–333.

SHIPLE, T. R., HOJATI, R., SANGIOVANNI-VINCENTELLI, A. L., AND BRAYTON, R. K. 1994. Heuristic minimization of BDDs using don't cares. In *Proceedings of the International Conference on Computer-Aided Design* (San Diego, June), 225–231.

SHIPLE, T. R., SINGHAL, V., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1996. Analysis of combinational cycles in sequential circuits. In *Proceedings of the International Symposium on Circuits and Systems* (Atlanta, May).

SILVA, J. AND SAKALLAH, K. 1996. GRASP–A new search algorithm for satisfiability. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November).

SINGHAL, V. 1996. Design replacements for sequential circuits. PhD Thesis, The University of California at Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley.

SINGHAL, V. AND PIXLEY, C. 1994. The verification problem for replaceability. In *Comput. Aided Verif.* (July).

SINGHAL, V., MALIK, S., AND BRAYTON, R. K. 1996. The case for retiming with explicit reset circuitry. In *Proceedings of the International Conference on Computer-Aided Design* (San Jose, CA, November).

SINGHAL, V., PIXLEY, C., AZIZ, A., AND BRAYTON, R. 1995. Exploiting power-up delay for sequential optimization. In *Proceedings of the European Design Automation Conference* (September), 54–59.

SINGHAL, V., PIXLEY, C., AZIZ, A., AND BRAYTON, R. 2001. A theory of safe replacements for sequential circuits. *IEEE Trans. Comput. Aided Des. Integ. Circ. Syst. 20*, 2 (Feb.).

TOUATI, H., SAVOJ, H., LIN, B., BRAYTON, R. K., AND SANGIOVANNI-VINCENTELLI, A. L. 1990. Implicit state enumeration of finite state machines using BDD's. In *Proceedings of the International Conference on Computer-Aided Design* (Santa Clara, CA, November), 130–133.

VILLA, T. University of California, Berkeley. Personal communication.

Received August 2001; accepted November 2002