Exploiting Power-up Delay for Sequential Optimization

Vigyan Singhal[*] **Carl Pixley**[†] **Adnan Aziz**[‡] **Robert K. Brayton**[‡]

Abstract

Recent work has identified the notion of safe replacement for sequential synchronous designs that may not have reset hardware or even explicitly known initial states. Safe replacement requires that a replacement design be indistinguishable from the original from the very first clock cycle after power-up. However, in almost any realistic application, the design is allowed to stabilize for many clock cycles before it is used. In this paper, we investigate the safety of a replacement if the replacement design is allowed to be clocked some cycles (that is, delayed) with arbitrary inputs before the design is reset. Having argued the safety of "delay" replacements, we investigate a new method of sequential optimization based upon the notion. We present experimental results to demonstrate that significant area optimizations can be gained by using this new notion of delay replaceability, and that there is a trade-off between the allowed number of clock cycles after power-up and the amount of optimization that can be obtained.

1 Introduction

The problem of design replacement for gate-level synchronous, sequential circuits is to replace a given design with another (optimized in some respect) making only minimal and reasonable assumptions about the interactions of the designs with its environment. The sequential nature of the design comes from the use of memory elements (such as latches and flip-flops). In synchronous designs all memory elements are triggered by a common clock signal. In some designs, all memory elements can be initialized by asserting an input that is hardwired to a specific port of each memory element.

In theory, designs with universal reset lines are easy to deal with because one design can safely replace another functionally if both designs have the same reset state and any sequence of identical inputs to the two designs in their respective reset states elicits identical sequences of outputs. However, the price for the ubiquitous reset line is the cost of routing wires to each memory element and the circuitry within each memory element to accomplish the initialization. Therefore, many real, industrial circuits contain at least some memory elements with no reset lines. This, however, raises a harder question about what it means for one sequential design without hardware reset to safely replace another. Furthermore, initializing such a design is a consequence of the interaction of a design with its environment.

Reflection [1] and experience show that, in the absence of knowledge about the intended use of a design and information about its environment, a necessary condition for safe replacement is that the State Transition Graphs (STGs) of a design and its replacement have equivalent Terminal Strongly Connected Components (TSCCs). However, since it is generally conceded that one cannot control the state in which a design is powered up, some condition concerning the way a design and its replacement are initialized is required. In [1], it was suggested that a necessary (but not sufficient) condition is that, in addition to having identical TSCCs, a design and its replacement have at least one synchronizing sequence in common. More recently a "safe replacement" condition was presented [2] which implies that every synchronization sequence for the original design also synchronizes the replacement design (this is equivalent to the redundant condition for fault detection in [3]). It was argued that that this notion of safe replacement was the weakest possible without knowledge or assumptions about the design's environment.

Based upon the notion of "safe replacement," a sequential synthesis approach was developed [4] that guaranteed that the new, optimized design would be a safe replacement. However, it was noticed that several effective optimization techniques, such as retiming, did not always result in safe 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 which cannot be visited beyond one clock cycle. Given that the machine can power up in any state, s_1 might be the initial state of the machine. Any such 1-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 1-cycle ephemeral states. However, notice that a new ephemeral may appear, that is a state, say s_2, to which only a 1-cycle ephemeral states can transition. States such as s_2 cannot re-appear after two clock cycles, no matter what inputs are given during the two cycles. The n-cycle ephemeral state sets form an "onion ring" structure in the STG [5]. The 1-cycle ephemeral states constitute onion ring 1, 2-cycle ephemeral states constitute onion ring 2, and so on. Since designs are finite state machines, for any design, there is a bound on the number of onion rings.

Elimination of all such ephemeral states for all clock cycles leaves a set of states, called the outer-envelope (OE) [1] or D^∞ (see Section 4 of this paper). Many well-crafted designs have ephemeral states due to the fact that binary encoding often leaves some states of the implementation without a corresponding state in the specified design. These states should be ephemeral. Also one-hot encodings of machines with n states results in $2^n - n$ states which may also be ephemeral. In addition, a forward retiming move across a fanout junction will also create ephemeral states [6]. However, all retiming moves preserve the outer-envelope.

The sequential method employed in [4] uses an "aliasing" technique. A core set of states is identified (all states except the 1-cycle ephemeral states). Then any 1-cycle ephemeral state, say s, is allowed to act like an alias. That is, for any input a, the state s is allowed to alias to any state s_0 which transitions to a state *inside the core* on input a, i.e. on input a state s is allowed to have the same next state and output as its alias s_0. One degree of flexibility comes from the fact that the s can alias one state s_0 for one input a_0 and to another state s_1 for another input a_1. However, this technique

[*] Cadence Berkeley Labs, 1919 Addison St., Berkeley, CA 94704, USA
[†] Motorola Inc., Bridgepoint Plaza I, 5918 W. Courtyard Dr., Suite 200, Austin, TX 78730, USA
[‡] Department of EECS, University of California, Berkeley, CA 94720, USA

does not seem to yield dramatic improvement in the design.

The method presented here is more flexible. We use the onion ring structure more exactly so that any ephemeral state, say s in onion ring i, can alias to a state which, for input a, transitions to a state inside an onion ring j where $j > i$ or to the OE. But this replacement is only delay-safe (i.e. we have assume that the environment of the design does not care about the behavior of the design till the i-th clock cycle after power-up).

The notion presented in the current work relies on the assumption that any realistic design will be used only after some cycles have elapsed after power-up; these cycles are necessary for the voltages and current to settle immediately after power-up. This initialization slack, often at least a few thousand clock cycles (usually milliseconds), is known in advance and is part of the design specification. Sequential optimization can use this initialization slack, say N cycles, in the following two ways. First, the design can be partitioned into non-overlapping, manageable sized pieces. Then each piece can be optimized using the N initialization slack cycles. Proposition 4.3 will show that this strategy will produce an entire design that is N-cycle delay-safe replacement. Second, designs can be optimized using pieces that do overlap. Proposition 4.4 will show that in the worst case, the delay needed accumulates for overlapping pieces. So the designer must take care not to exceed a 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 designs is modified. Specifically, the outer-envelope is unchanged.

Previous efforts have been made to tackle the sequential design replacement problem[1]. A notion of design equivalence was defined in [1]; however, they did not present any approach to use that notion for logic optimization. Another notion of equivalence was suggested in [7] which was used for sequential resynthesis by redundancy removal in [8]. Their notion of redundancy results in replacements which are not 0-cycle delay-safe replacements [9]; it has not been shown that if their replacements are n-cycle delay-safe replacements, neither what this n might be. A sequential resynthesis technique relevant to our model is retiming and resynthesis [10]. Retiming and resynthesis results in delay-safe replacements as shown in [11, 6]. However, this approach has not given much optimization for arbitrary control logic, and also the benchmark circuits; it seems to work best for pipelined data paths [10]. Another method for sequential optimization is the use of synchronous recurrence equations as described in [12]. While we suspect that synchronous recurrence equations [12] might lead to valid delay-safe replacements, our procedure returns greater area optimizations and consume less CPU time (see Section 6). Moreover, it is not clear how the initialization slack n can be used by the method described in [12] so that it is guaranteed that the replacement is an n-cycle safe-replacement.

To summarize, our optimization techniques for delay replacements allow us to achieve significant logic optimizations while allowing the designer to specify the flexibility in the power-up delay, N.

2 Terminology and Background

We now make precise the notion of a finite state machine and our model for sequential hardware.

Definition 1 *A* deterministic Finite State Machine (DFSM) M *is a quintuple,* $(Q, I, O, \lambda, \delta)$*, where* Q *is the set of states,* I *is the set*

[1]In this paper, we will not discuss other sequential resynthesis techniques which use the information of a designated start state of the circuit to extract flexibility and use this for optimization. These methods cannot be applied to the case where memory elements do not have reset lines.

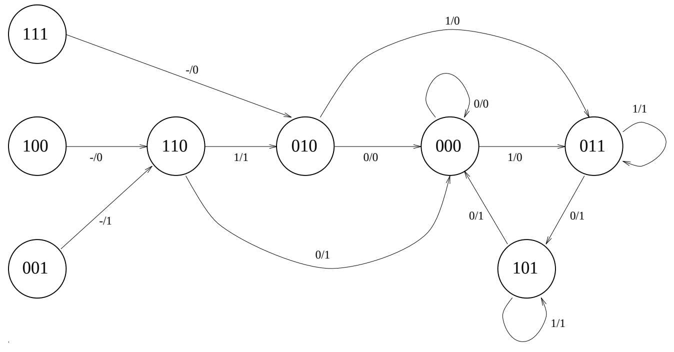

Figure 1: Original design P.

of input values, O *is the set of output values. The output function* $\lambda : (Q \times I) \rightarrow O$ *and the next state function* $\delta : (Q \times I) \rightarrow Q$ *are completely-specified functions.*

A hardware *design* D consists of a set of interconnected latches and gates. A design with n input wires, m output wires and t latches is naturally associated with a DFSM D. We will use D to also denote the set of states in the state space $\{0,1\}^t$ (it will be clear from the context if D refers to the design or the set of states). The input space $I_D = \{0,1\}^p$, and output space $O_D = \{0,1\}^m$, the next state function is $\delta_D : D \times I_D \rightarrow D$ and output function $\lambda_D : D \times I_D \rightarrow O_D$ are defined by the corresponding logic. We are not assuming explicit set or reset pin to any latch; thus when the design powers up, it can non-deterministically power up in any one of the 2^t states. If some of the latches in the hardware design being optimized have set or reset pins, they can easily be modeled by using latches without such pins and adding some additional logic.

We will also use λ_D and δ_D to denote the output and next state functions on sequences of inputs. So, if $\pi = a_1 \cdot a_2 \cdot a_3 \cdots a_q \in I_D^q$ is a sequence of q inputs, these functions are recursively defined as $\lambda_D(s, \pi) = \lambda_D(s, a_1) \cdot \lambda_D(\delta_D(s, a_1), \pi')$ and $\delta_D(s, \pi) = \delta_D(\delta_D(s, a_1), \pi')$, where $\pi' = a_2 \cdot a_3 \cdots a_p$.

Since distinct netlists of gates can compute the same combinational function, distinct hardware designs can correspond to the same DFSM. Given a DFSM, we use the term "design" to denote any gate level implementation of the DFSM.

3 Safe Replacements

The safe replacement condition requires that if the original design is replaced with the new design, there is no way the environment can detect the replacement by looking at the input-output behavior of the design. The precise condition for safe replacement was presented in [2]:

Definition 2 *Design* D_1 *is a* safe replacement *for design* D_0 *(denoted by* $D_1 \preceq D_0$*) if given any state* $s_1 \in D_1$ *and any finite input sequence* $\pi \in I^*$*, there exists* <u>*some*</u> *state* $s_0 \in D_0$ *such that the output behavior* $\lambda_{D_1}(s_1, \pi) = \lambda_{D_0}(s_0, \pi)$*.*

As an example of safe replacement, consider designs P and Q in Figures 1 and 2, respectively. It can easily be seen that $Q \preceq P$ (see [2] for details).

It was shown in [2] that Definition 2 is the weakest condition which guarantees that no environment can detect the replacement. The replacement is indistinguishable from the original design even for environments which can observe and control the design from the very first clock cycle after power-up.

We frequently use the following two observations from [2]:

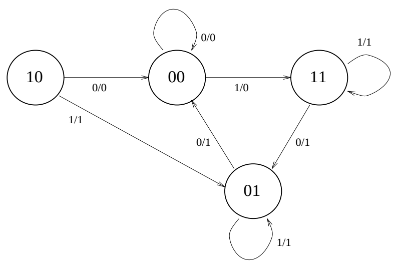

Figure 2: Design Q (a safe replacement for design P)

- If each state of design C is equivalent to a state in design D, then $C \preceq D$, but $C \preceq D$ does not imply that every state of design C is equivalent to a state in design D.
- The relation $\preceq$ is transitive.

Unfortunately, the requirement for one design to be a safe replacement of another was shown to be very strong. Not much flexibility exists in choosing replacements that are safe; in particular this means that little scope exists for significant design optimization. This motivated our current work, where based on the number of clock cycles between the power-up and when a design is actually is used, substantial flexibility can be extracted from a design.

4 Delay Replacements

Definition 3 *Given a design D, the n-*cycle delayed design *(denoted by D^n) is the restriction of D to the set of states $\{s | \exists \pi \in I^n, s' \in D : \delta_D(s', \pi) = s\}$, i.e. 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. Define D^∞ to be the set of states $\{s | \forall n : \exists \pi \in I^n, s' \in D : \delta_D(s', \pi) = s\}$, i.e. 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.*

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 the set of states in D^n is closed under all inputs; also that the set of states in D^m is a subset of the states in D^n if $m > n$. 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 $D^{n-1} = D^n$. Using the terminology in [5], this number n is the number of *onion rings* of the design D. D^∞ is also the same as the outer-envelope (OE) of [1].

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

We now present our condition for delay replacement. As we noted in Section 1, as part of the system specification, an initialization slack of n clock cycles is available. We can use the flexibility afforded by this slack by requiring a design to be an acceptable replacement if it is allowed to clock n extra cycles with arbitrary inputs before it is used:

Definition 4 *Given a design D, a new design C is an n-*delay *replacement for D, if $C^n \preceq D$.*

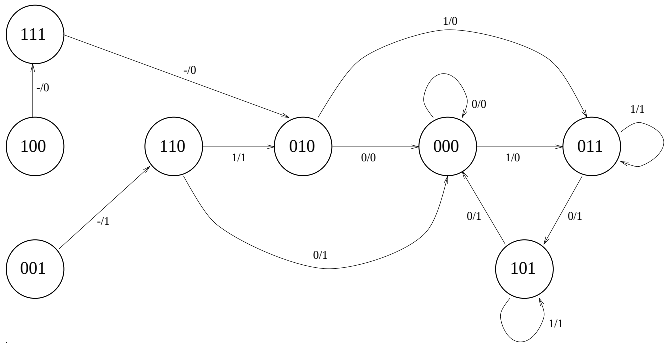

Figure 3: Example design R

We should note here that Leiserson and Saxe [11] had a similar notion for valid replacements which they used to justify the validity of retimed designs (see also [6]). They considered a replacement design C a valid replacement of design D if each state in C^n is equivalent to a state in D. It should be clear that their notion is strictly stronger than ours, and thus allows less flexibility for replacement. The following two properties of delay replacements follow (we refer the reader to [13] for the proofs of all the results that are presented in this paper):

Proposition 4.1 *If $C^n \preceq D$, and $m > n$, then $C^m \preceq D$.*

Proposition 4.2 *If $C^n \preceq D$ and $B^m \preceq C$, then $B^{m+n} \preceq D$.*

Thus, for example, a 2-delay replacement followed by a 3-delay replacement on the same design results in a n-delay replacement for any $n \geq 5$.

As an example of delay replacement, consider designs P and R in Figures 1 and 3. It can be seen that $R^1 \preceq P$; however, $R \not\preceq P$ (the state $100 \in R$ produces out 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 P).

Compositionality of Delay Replacements

For the optimization of any design, we can select arbitrary subpieces of the design and perform delay replacements on these. In this subsection, we examine the effect of making a delay replacement on the larger design. We will show that making n-delay replacements on non-overlapping sub-pieces 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 sub-pieces which are overlapping, we get an entire design which is $2n$-delay safe (notice that, as a special case of this, if the two sub-pieces are identical, we already know from Proposition 4.2 that the entire design in $2n$-delay safe).

Informally, two given hardware designs can be "wired" together by driving some of the inputs of one by outputs of the other and vice versa. The remaining inputs and outputs will be primary inputs and primary outputs of the composed design. Given designs A and B, we will use $A \otimes B$ to denote their composition. A formal definition of design composition is given in [14]. We obtain the following result which shows that composing two delay replacements adds up the delays.

Proposition 4.3 *If $Q^m \preceq R$ and $C^n \preceq D$, then $(Q \otimes C)^p \preceq (R \otimes D)$, 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 sub-pieces on a large design, the delays add up. This can be seen as a consequence of Propositions 4.2 and 4.3.

Proposition 4.4 *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 partition of the same design C), and let $U^n \preceq S$. Then, if $B = U \otimes T$, it is true that $B^{m+n} \preceq D$.*

Delay-Preserving Replacements

This subsection discusses another interesting replacement notion relating power-up delays to replacements. However, the reader can choose to ignore this section without sacrificing any understanding of this paper.

We had originally formulated the following conditions for allowing a replacement in the presence of the power-up slack:

Definition 5 *Given a design D, a new design C is an n-*delay-preserving *replacement for D, if $C^n \preceq D^n$.*

As examples of delay-preserving replacements consider designs P and R in Figures 1 and 3. We already know that R is a 1-delay replacement of P (i.e. $R^1 \preceq P$). However, R is not a 1-delay-preserving replacement of P; instead R is a 2-delay-preserving replacement of P.

It is easily seen that every n-delay-preserving replacement is also an n-delay replacement (of Definition 4). Delay-preserving replacement is a useful notion because successive n-delay-preserving replacements on the same design still result in an n-delay-preserving replacement (unlike n-delay replacements, which add up, as shown in Proposition 4.2). However, we found an example which illustrated that delay-preserving replacements do not have nice compositionality properties which allow us to make substitutions on sub-pieces of designs. Thus, the iterative optimization strategy that we proposed in the previous sub-section will not work.

The reader is referred to [13] for the above-mentioned example and other properties of delay-preserving replacements. That paper also relates delay-preserving replacements to delay replacements (called delay-accumulating replacements in [13]) and includes an interesting result which shows that in the limit, as n approaches ∞, these two notions converge.

5 Resynthesis for Delay Replaceability

Typically, the synthesis process has two stages: first, the set of all possible implementations is characterized using the flexibility given by the replacement condition. Following this, one implementation is chosen according to some optimality criteria. In this section we describe how to extract the flexibility for delay-preserving replaceability, and then describe how it is used to minimize the area of the combinational logic associated with a hardware design.

Let the original design D have $(k+1)$ different delayed designs, $D^0, D^1, \ldots, D^k$; for any $j \geq k : D^j = D^k = D^\infty$. These $(k+1)$ delayed designs form an "onion ring" structure [5]. Clearly any state reachable from itself under some input sequence belongs to each D^i and to D^∞. We will not alter the behavior of any such state (a "stable" state). All the flexibility for resynthesis comes from the set of "transient" states, i.e. $D \setminus D^\infty$.

Recall that an n-delay replacement allows the new design to have some flexibility over the first n clock cycles after the power-up, and after these clock cycles it is indistinguishable from the

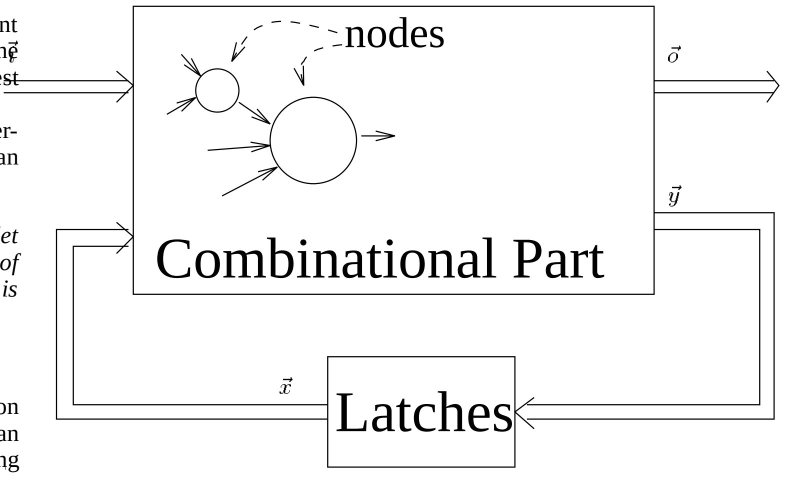

Figure 4: Multi-level Sequential Network = Combinational Part + Latches

original design. As we said before, often it is known in advance that the design will be allowed to settle for at least a fixed number of clock cycles before it is used. We use this slack n for resynthesis— the higher n is, the greater the flexibility allowed for resynthesis; however, 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 will 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.

A sequential gate-level design can be viewed as a connection between a purely combinational part and memory elements. The combinational part can be represented by a directed acyclic graph, individual vertices of which compute combinational functions. The primary inputs and outputs to the design are denoted by $\vec{x}$ and $\vec{y}$, respectively (Figure 4). The outputs of the combinational part which feed the memory elements are denotes by $\vec{y}$, and the lines which feed the output of the memory elements to the combinational part are denoted by $\vec{x}$. To do resynthesis for delay replaceability, we obtain a Boolean relation $\mathcal{Q}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ which describes the flexibility for replacement. We will then use this relation to do multi-level resynthesis on our design.

First, we informally describe the flexibility which will be specified later using a Boolean relation $\mathcal{Q}$. We note here that techniques for using a Boolean relation to do multi-level synthesis [15] require the relation to be such that the starting design satisfies the relation. The Boolean relation is such that the behavior of the states in D^n (the "stable" states) 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 can also be seen that for any design C that satisfies the relation it is true that $C^n = D^n$, i.e. after the first n clock cycles we every reachable state in C is equivalent to one 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 arbitrarily chosen.

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

$$\begin{aligned}\mathcal{Q}(\vec{i}, \vec{x}, \vec{o}, \vec{y}) \quad &= \quad \Sigma_{i=0}^{n-1}[(\vec{x} \in D^i \setminus D^{i+1}) \wedge (\vec{y} \in D^{i+1})] + \\ &\quad [(\vec{x} \in D^n) \wedge (\vec{y} = \delta_D(\vec{x}, \vec{i})) \wedge (\vec{o} = \lambda_D(\vec{x}, \vec{i}))]\end{aligned}$$

Ckt.	I/O/L	k	Initial	ODC only		Safe		n-delay replacements							
			size			replacement		$n = 1$		$n = 2$		$n = 5$		$n = \infty$	
				r	time	r	time	r	time	r	time	r	time	r	time
s27	4/1/3	1	12	0	0.05	0	0.10	0	0.08						
s208	10/1/8	0	95	7	0.66										
s298	3/6/14	7	150	20	1.13	20	4.12	20	1.70	20	1.76	37	1.98	43	3.02
s349	9/11/15	7	160	6	2.24	6	316.54	6	3.58	12	5.76	12	11.44	15	21.17
s382	3/6/21	101	176	12	1.61	12	17.59	14	10.58	14	33.77	14	54.81	20	161.51
s386	7/7/6	1	204	17	1.46	31	2.34	77	2.04						
s400	3/6/21	101	184	22	1.57	22	18.57	24	11.04	24	36.48	24	57.52	30	163.88
s444	3/6/21	101	184	21	1.64	21	20.33	23	11.33	23	41.85	23	70.13	28	188.03
s510	19/7/6	4	280	1	4.26	1	9.94	1	5.09	1	5.19	1	5.51		
s526	3/6/21	667	283	43	3.07	43	15.33	43	6.54	45	9.85	46	11.00	95	66.30
s641	35/23/19	1	199	5	4.13	timeout		5	19.06						
s713	35/23/19	1	204	9	4.25	timeout		9	20.37						
s820	18/19/5	1	504	142	9.87	143	32.92	156	11.07						
s832	18/19/5	1	521	152	11.15	152	32.49	163	13.53						
s953	16/23/29	1	489	5	12.94	5	70.27	5	45.24						
s1196	14/14/18	2	618	18	78.71	18	1556.71	18	187.62	18	207.11				
s1238	14/14/18	2	690	65	107.12	65	1976.36	65	224.32	65	240.97				
s1488	8/19/6	1	813	57	23.54	57	102.59	116	22.80						
s1494	8/19/6	1	819	65	23.82	65	90.50	122	22.81						

Table 1: Experimental results. I/O/L denotes the number of inputs, outputs and latches. The number of onion rings for each design $(k + 1)$. The initial size of the circuits and the savings r are in the number of literals. ODC only denotes optimization obtained only by combinational resynthesis. Since we know that for any integers $m \geq k$, we would get the same results for both m-delay replacement and k-delay replacement, the redundant experiments (denoted by blanks in the above table) were not performed.

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

Notice that for any integer $m \geq k$, the delayed design D^m is the same as D^∞, i.e. the set of stable states. Thus, the flexibility described by the relation $\mathcal{Q}$ 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.

Once we have the BDD for the Boolean relation $\mathcal{Q}(\vec{i}, \vec{x}, \vec{o}, \vec{y})$ we can use standard BDD-based multi-level synthesis techniques to propagate this flexibility to individual nodes in the network and then minimize the nodes [4, 15].

6 Experiments

In this section we report experimental results using the algorithm described in Section 5 on ISCAS85 sequential circuits. We used BDDs to manipulate the Boolean relations and sets. We focused on the area reduction for n-delay replacements. We report results for various values of n.

The experimental results, obtained using a DECstation5900 are shown in Table 1. The starting circuits were obtained from ISCAS89 benchmark circuits with the SIS commands (`sweep; eliminate -1`) applied [16]. These eliminate single-input and constant nodes and collapse nodes which do not fan out to more than one node. First we show the optimizations obtained by just doing the standard multi-level combinational optimization [17] using observability don't cares (ODC) propagated to the individual nodes in the network. We also show the optimizations obtained by the safe replacement resynthesis method described in [4]. Then, we show the results of the method presented in this paper for n-delay replacements for $n = 1, 2, 5, \infty$. 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, e.g. `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. The much larger CPU times for pure safe replaceability can be attributed to the rigid conditions for safe replacement, which require placing constraints on the outputs from the "transient" states as well as correlating the next states of "transient" states with the inputs. Both these constraints are avoided by the resynthesis method presented in Section 5. This leads to a much smaller BDD to express the relation $\mathcal{Q}$, and hence, faster multi-level resynthesis.

We suspect that synchronous recurrence equations [12] also lead to delay replacements. However, the experimental results presented there indicate that using synchronous recurrence equations is not very effective; our optimization method (using the SIS commands `sweep; eliminate -1` followed by the optimization procedure described in this paper) produces smaller circuits using less CPU times than those reported in [12]. (Note that the CPU times indicated in Table 1 do not include the time used by the SIS pre-processing commands; however, for all the examples, that time is much less than the times reported for the sequential optimizations. However, the CPU times reported in [12] are greater than double the times reported in Table 1.)

We performed an experiment on one of the benchmark circuits with large number of onion rings (`s526`) to explore the tradeoffs between flexibility and the power-up delay allowed. The results are in Table 2. The table shows that 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

n-delay replacements initial size = 283 literals		
n	reduction	time
1	43	5.93
100	54	22.47
200	58	29.24
300	70	34.52
400	69	38.60
500	69	44.05
600	72	48.78
667	95	68.14

Table 2: Power-up delay/flexibility tradeoff for s526; reduction is in number of literals

Boolean relation $\mathcal{Q}$ goes up with higher n.

In the experiments above, the initial nodes of the circuits have been collapsed minimally. We see that safe replaceability gives very marginal improvements over pure combinational reductions, and at a much larger CPU time cost. For this reason, it was argued in [4] that we might get better use of the safe replaceability notion by using larger node sizes in the circuits. They increased the node sizes in these benchmark circuits by using SIS commands `eliminate 10` or `collapse`. For the sake of completeness, we ran our algorithm on these starting points also. For the results, the reader is referred to [13]. This experiment showed that once again, using n-delay replacements instead of safe replacements allows us greater flexibility for resynthesis and gives much better optimizations. Also, the CPU times are once again much better than those for safe replacements.

7 Conclusions

We presented the notions of delayed replacement which allows additional degree of flexibility over safe replacement by letting the design settle down for a certain number of clock cycles after power-up. The number can be controlled by the designer and input to our synthesis tool, and then we can use all the available flexibility while guaranteeing the degree of safeness the designer specified. We have suggested how this notion can be used in an iterative synthesis methodology where successive design replacements are made. We have shown experimental results to illustrate that we obtain significant optimizations at affordable CPU costs by using our notion of delay replacement.

8 Acknowledgements

We would like to thank Noel Strader of Motorola Inc., Austin for many very useful discussions. This research was supported by NSF/DARPA Grant MIP-8719546 and SRC Grant 95-DC-324.

References

[1] C. Pixley, “A Theory and Implementation of Sequential Hardware Equivalence,” *IEEE Trans. Computer-Aided Design*, vol. 11, pp. 1469–1494, Dec. 1992.

[2] V. Singhal and C. Pixley, “The Verification Problem for Safe Replaceability,” in *Proc. of the Conf. on Computer-Aided Verification* (D. L. Dill, ed.), vol. 818 of *Lecture Notes in Computer Science*, pp. 311–323, Springer-Verlag, June 1994.

[3] I. Pomeranz and S. M. Reddy, “Classification of Faults in Synchronous Sequential Circuits,” *IEEE Trans. Computers*, vol. 42, pp. 1066–1077, Sept. 1993.

[4] C. Pixley, V. Singhal, A. Aziz, and R. K. Brayton, “Multi-level Synthesis for Safe Replaceability,” in *Proc. Intl. Conf. on Computer-Aided Design*, (San Jose, CA), pp. 442–449, Nov. 1994.

[5] S.-W. Jeong, *Binary Decision Diagrams and their Applications to Implicit Enumeration Techniques in Logic Synthesis*. PhD thesis, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO 80309, 1992.

[6] V. Singhal, C. Pixley, R. L. Rudell, and R. K. Brayton, “The Validity of Retiming Sequential Circuits,” in *Proc. of the Design Automation Conf.*, (San Francisco, CA), pp. 316–321, June 1995.

[7] K.-T. Cheng, “Redundancy Removal for Sequential Circuits Without Reset States,” *IEEE Trans. Computer-Aided Design*, vol. 12, pp. 13–24, Jan. 1993.

[8] L. Entrena and K.-T. Cheng, “Sequential Logic Optimization by Redundancy Addition and Removal,” in *Proc. Intl. Conf. on Computer-Aided Design*, (Santa Clara, CA), pp. 310–315, Nov. 1993.

[9] M. A. Iyer, D. E. Long, and M. Abramovici, “Identifying Sequential Redundancies Without Search.” Unpublished manuscript, 1995.

[10] S. Malik, E. M. Sentovich, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Retiming and Resynthesis: Optimization of Sequential Networks with Combinational Techniques,” *IEEE Trans. Computer-Aided Design*, vol. 10, pp. 74–84, Jan. 1991.

[11] C. E. Leiserson and J. B. Saxe, “Optimizing Synchronous Systems,” *Journal of VLSI and Computer Systems*, vol. 1, pp. 41–67, Spring 1983.

[12] M. Damiani and G. De Micheli, “Recurrence Equations and the Optimization of Synchronous Logic Circuits,” in *Proc. of the Design Automation Conf.*, (Anaheim, CA), pp. 556–561, June 1992.

[13] V. Singhal, C. Pixley, A. Aziz, and R. K. Brayton, “Delaying Safeness for More Flexibility,” Tech. Rep. UCB/ERL M95/5, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, Jan. 1995.

[14] A. Aziz, T. R. Shiple, V. Singhal, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Formula-Dependent Equivalence for Compositional CTL Model Checking,” in *Proc. of the Conf. on Computer-Aided Verification* (D. L. Dill, ed.), vol. 818 of *Lecture Notes in Computer Science*, pp. 324–336, Springer-Verlag, June 1994.

[15] H. Savoj and R. K. Brayton, “Observability Relations and Observability Don’t Cares,” in *Proc. Intl. Conf. on Computer-Aided Design*, (Santa Clara, CA), pp. 518–521, Nov. 1991.

[16] E. M. Sentovich, K. J. Singh, C. Moon, H. Savoj, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, “Sequential Circuit Design Using Synthesis and Optimization,” in *Proc. Intl. Conf. on Computer Design*, (Cambridge, MA), pp. 328–333, Oct. 1992.

[17] H. Savoj, R. K. Brayton, and H. Touati, “Extracting Local Don’t Cares for Network Optimization,” in *Proc. Intl. Conf. on Computer-Aided Design*, (Santa Clara, CA), pp. 514–517, Nov. 1991.