Sequential Optimisation without State Space Exploration

A Mehrotra[*] S Qadeer[*] V Singhal[†] R K Brayton[*] A Aziz[‡] A L Sangiovanni-Vincentelli[*]

Abstract

We propose an algorithm for area optimisation of sequential circuits through redundancy removal. The algorithm finds *compatible* redundancies by implying values over nets in the circuit. The potentially exponential cost of state space traversal is avoided and the redundancies found can all be removed at once. The optimised circuit is a safe delayed replacement of the original circuit. The algorithm computes a set of compatible sequential redundancies and simplifies the circuit by propagating them through the circuit. We demonstrate the efficacy of the algorithm even for large circuits through experimental results on benchmark circuits.

1 Introduction

Sequential optimisation seeks to replace a given sequential circuit with another one optimised with respect to some criterion – area, performance or power, in a way such that the environment of the circuit cannot detect the replacement. In this work, we deal with the problem of optimising sequential circuits for area. We present an algorithm which computes sequential redundancies in the circuit by propagating implications over its nets. The redundancies we compute are compatible in the sense that they form a set that can be removed simultaneously. Our algorithm works for large circuits and scales better than those algorithms that depend on state space exploration.

The starting point of our work is [1], in which a method was described to identify sequential redundancies without exploring the state space. The basic algorithm is that for any net, two cases are considered: the net value is 0 and the net value is 1. For each case, constants as well as unobservability conditions are learnt on other nets. If some other net is either set to the same constant for both cases, or to a constant in one case and is unobservable in the other, it is identified as redundant. For example, consider the trivial circuit shown in Figure 1. For the value $n1 = 0$ the net $n2$ is unobservable and for the value $n1 = 1$, the net $n2$ is 1. Thus net $n2$ is stuck-at-1 redundant. However, the redundancies found by the method in [1] are not compatible in the sense that they remain redundant even in the

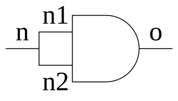

Figure 1: Example of incompatible redundancies

presence of each other. For instance, the redundancy identification algorithm will declare both the inputs n_1 and n_2 as stuck-at-1 redundant. However, for logic optimisation, it is incorrect to replace *both* the nets by a constant 1.

The straightforward application of Iyer's method to redundancy removal is to identify one redundancy by their implication procedure, remove the redundancy and iterate until convergence. Our goal is to learn all *compatible* implications in the circuit in one step and use the compatibility of these implications to remove all the redundancies simultaneously (in this sense our method for finding compatible unobservabilities is related to the work in [2, 3] for computing compatible ODC's (observability don't cares)). This is our first contribution. Secondly, we generalise the implication procedure by combining it with recursive learning [4] to enhance the capability of the redundancy identification procedure. Recursive learning lets us perform case split on unjustified gates so that it is possible to learn more implications at the expense of computation time. Consider the circuit in Figure 2. Setting net a to 0 implies that net f is 0. If we set a to 1, $a1$ becomes 1, but the AND-gate connected to $a1$ remains unjustified. If we perform recursive learning for the two justifications: $d = 0$ and $d = 1$, then for the former case, net f becomes 0, and for the latter case, f becomes unobservable because e is 1. Thus, for all the possible cases, either f is 0 or it is unobservable. Hence f is declared stuck-at-0 redundant. Recursive learning helps identify these kinds of new redundancies. We present data which shows that we are able to gain significant optimisations on large benchmark circuits using these two new improvements. In fact, for some circuits, we find that recursive learning not only gives us more optimisation, it is even faster since a previous recursive learning step makes the circuit simpler for a later stage.

We do not assume designated initial states for circuits. For sequential optimisation, we use the notion of c-delay replacement [1, 5]. This notion guarantees that every possible input-output behaviour that can be observed in the new circuit after it has been clocked for c cycles after power-up, must have been present in the old circuit. In contrast to the work in [5, 6], the synthesis method presented here does not require state space

[*]University of California at Berkeley, Berkeley, CA 94720
[†]Cadence Berkeley Labs, Berkeley, CA 94704
[‡]University of Texas at Austin, Austin, TX 78712

0-89791-993-9/97 $10.00 © 1997 IEEE

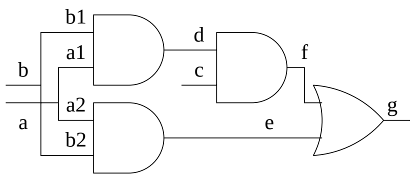

Figure 2: Example of recursive learning

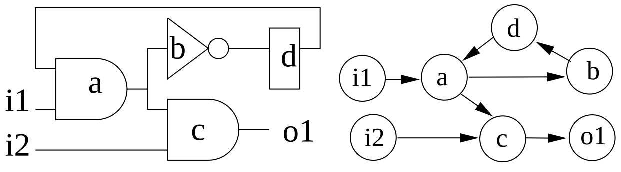

Figure 3: A circuit and its graph

traversal, and can therefore be applied to large sequential circuits. Recursive learning has been used earlier for optimisation, as described in [7], but their method is applied only to combinational circuits and they do not use unobservability conditions. Another procedure to do redundancy removal is described in [8], but as [9] shows, their notion of replacement is not compositional and may also identify redundancies which destroy the initialisability of the circuit. We have therefore chosen to use the notion of safe delayed replacement which preserves responses to all initialising sequences. We are interested in compositionality because we would like a notion of replacement that is valid without making any assumptions about the environment of the circuit. This is why our replacement notion is safer than that used in [10] which identifies sequential redundancies by preserving weak synchronising sequences. Their work implicitly assumes that the environment of the circuit has total control so that it can supply the arbitrary sequence that the redundancy identification tool has in mind. Our approach does not pose any such restrictions.

The rest of the paper is organised as follows. In Section 2, we present our algorithm to compute compatible redundancies on combinational and sequential circuits. In Section 3, we present experimental results on some large circuits from the ISCAS benchmark set. In Section 4, we conclude with some directions for future work.

2 Redundancy Removal

We present an algorithm for sequential circuits that have been mapped using edge-triggered latches, inverters and 2-input gates; note that any combinational implementation can be mapped to a circuit containing only inverters and 2-input gates. We use the notion of circuit graph for explaining our algorithm. A *circuit graph* is a labelled directed graph whose vertices correspond to primary inputs, primary outputs, logic gates and latches, and edges correspond to wires between the elements of the circuit. The label of a vertex identifies the type of element it represents (e.g. two-input gates, inverters or latches). We refer to an edge in the circuit graph as a *net*. Figure 3 shows an example of a circuit graph.

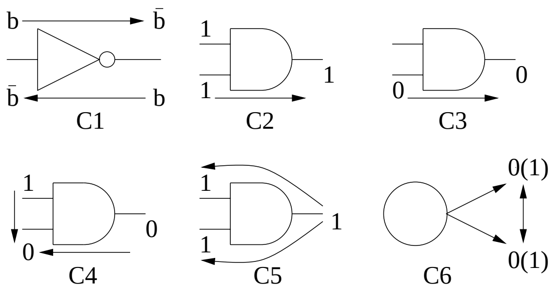

Figure 4: Rules for implying constants

2.1 Combinational redundancies

We explain our algorithm and prove its correctness for combinational circuits and later extend it to sequential circuits. Consider a circuit graph $G = (V,E)$ of a circuit, where V is the set of vertices and E is the set of nets. An *assumption* A on the subset $P \subseteq E$ is a labelling of the nets in P by values from the set $\{0,1\}$. Let $n \in P$ be a net. We write $A : n \mapsto v$ if A labels the net n with the value v. An assumption is denoted by an ordered tuple. The set of all possible assumptions on the set P of nets is denoted by A_P. Consider the set $P = \{m,n\}$. The assumption labelling m with 0 and n with 1 is denoted by $\langle m \mapsto 0, n \mapsto 1\rangle$ and $A_P = \{\langle m \mapsto 0, n \mapsto 0\rangle, \langle m \mapsto 0, n \mapsto 1\rangle, \langle m \mapsto 1, n \mapsto 0\rangle, \langle m \mapsto 1, n \mapsto 1\rangle\}$. An assumption $A \in A_P$ is *inconsistent* if it is not satisfiable for any assignments to the primary inputs of the circuits. For instance, an assumption of 0 at the input and 1 at the output of an AND gate is inconsistent.

In the algorithm, values are implied at nets in $E \setminus P$ from an assumption on P. We imply either constants or *unobservability* indicators at nets. We indicate unobservability at a net by implying a symbolic value $\otimes$ at it. Let $R = \{0,1,\otimes\}$ be the set of all possible value that can be implied at any net. An *implication* is a label $(n_i = r)$ where n_i is a net and $r \in R$. Figure 4 illustrates the rules for implying constants. Rules C1, C2, C3 and C5 are self-explanatory. Rule C4 states that for an AND gate, 0 at the output and 1 at an input implies 0 at the other input. Rule C6 states that a constant at some fanout net of a gate implies the same constant at all other fanout nets. Figure 5 illustrates the rules for implying $\otimes$'s. Rule O1 states that 0 at an input of an AND gate implies a $\otimes$ at the other input. Rule O2 states that a $\otimes$ at every fanout net of a gate implies a $\otimes$ at every fanin net of that gate. Note that constants can be implied in both directions across a gate while $\otimes$ propagates only backwards. We have shown rules only for inverters and AND gates but similar rules can be easily formulated for other gates as well. We use these rules to label the edges of the circuit graph. A constant (0 or 1) label on a net indicates that

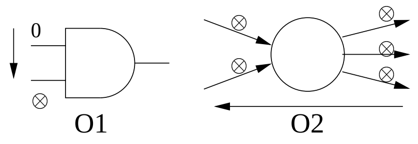

Figure 5: Rules for implying unobservability

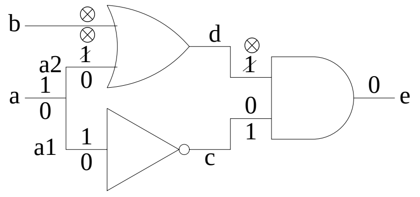

Figure 6: Overwriting constants with unobservability indicators

the net assumes the respective constant value under the current assumption. A $\otimes$ label indicates that the net is not observable at any primary output. Hence, it can be freely assigned to either 0 or 1 under the current assumption. Suppose for every assumption in A_P, some net n is labelled either with constant v or with $\otimes$, then we can safely replace n with constant v. This is because we have shown that under every possible assumption, either the net takes the value v or its value does not affect the output. We can therefore conclude that net n is *stuck-at-v* redundant.

We are concerned about the compatibility of all labellings because otherwise we run the danger of marking nets with labels so that all labels are not consistent. For example, consider the circuit in Figure 1. For the purpose of identifying redundancies, [1] would infer the implications $(n1 = 1)$ and $(n2 = 1)$ from the assumption $\langle n \mapsto 1\rangle$. Additionally, the assumption $\langle n \mapsto 0\rangle$ implies that $(n1 = 0)$ and $(n2 = *)$; similarly $\langle n \mapsto 0\rangle$ implies that $(n2 = 0)$ and $(n1 = *)$ (notice that [1] use the symbol $*$ to denote unobservability while we use $\otimes$ to denote *compatible* unobservability). So, [1] would rightly claim that both $n1$ and $n2$ are *stuck-at-1* redundant in isolation; however, for redundancy removal it is easy to see that we cannot set both $n1$ and $n2$ to 1 simultaneously. This is why we want to make all labellings compatible.

A sufficient condition for the redundancies to be compatible is to ensure that the procedure for computing implications from an assumption returns *compatible* implications, i.e., every implication is valid in the presence of all other implications. It is easy to see that if the labelling of edges in the circuit graph is done by invoking the rules described above and no label is ever overwritten, then the set of learnt implications will be compatible. For instance, in the circuit of Figure 1, once $n1$ is labelled with $\otimes$, a $\otimes$ cannot be inferred at $n2$ because $(n1 = \otimes)$ cannot be overwritten with $(n1 = 0)$. But this approach is conservative and will miss some redundancies. In Figure 6, we show an example where overwriting a constant with a $\otimes$ yields a redundancy which could not have been found otherwise. We propagate implications from assumptions on the net a. The implications from $\langle a \mapsto 0\rangle$ are written below and those from $\langle a \mapsto 1\rangle$ are written above the wires. Note that while propagating implications from $\langle a \mapsto 1\rangle$, $a2$ and d are initially labelled with 1 but after labelling c with 0, the labels at d and $a2$ are successively overwritten with $\otimes$'s. Hence, $a2$ is found to be *stuck-at-0* redundant. As a result, the OR gate can be removed. We prove later in this section that this overwriting does not make previously learnt implications invalid, i.e., compatibility of implications is maintained, if the only overwriting that is allowed is that of constants with unobservability indicators.

```
**redundancy_remove** ($G = (V,E)$)
/* find and remove redundancies from the circuit graph */
while (there is an unvisited net $n$ in the circuit graph) {
    $S$ := learn_implications ($G$, $\langle n \mapsto 1\rangle$)
    $T := \{(l = v) \mid (l = v) \in S \vee (l = \otimes) \in S\}$
    $\overline{S}$ := learn_implications ($G$, $\langle n \mapsto 0\rangle$)
    $\overline{T} := \{(l = v) \mid (l = v) \in \overline{S} \vee (l = \otimes) \in \overline{S}\}$
    $R := T \wedge \overline{T}$
    for every implication $(n = v) \in R$ set net $n$ to constant $v$
    propagate constants and simplify
}

**learn_implications** ($G = (V,E), A$)
/* propagate implications on the circuit graph given an assignment */
{
    forall $n$ such that $A : n \mapsto v$ {
        label $n \leftarrow v$
    }
    while (some rule can be invoked) {
        let $(n = b)$ be the new implication
        if $(b = \otimes)$
            label $n \leftarrow b$
        if ($(n = b)$ conflicts with a current label)
            return $\{l = * \mid l \in E\}$
        else
            label $n \leftarrow b$
    }
    return set of all current labels
}
```

Figure 7: Combinational redundancy removal algorithm

Our algorithm for removing combinational redundancies is given in Figure 7. The function *learn_implications* takes as input an assumption A on an arbitrary subset of nets and labels nets with values from $\{0, 1, \otimes\}$ learnt through implications. Initially all nets n such that $A : n \mapsto v$ is an assumption, are labelled. Then we derive new labels by invoking the rules C1-C6 and O1-O2 and similar rules for other kinds of two input gates. Note that at all times each net has a unique label and constants can be overwritten with $\otimes$'s but not vice-versa. It returns the set of all final labels. The function *redundancy_remove* takes as input a circuit graph G and calls *learn_implications* successively with assumptions $\langle n_i \mapsto 0\rangle$ and $\langle n_i \mapsto 1\rangle$ on the singleton subset $\{n_i\}$. The two sets of labels are used to compute all pairs n and v such that n is *stuck-at-v* redundant. We later show that our labelling procedure for learning implica-

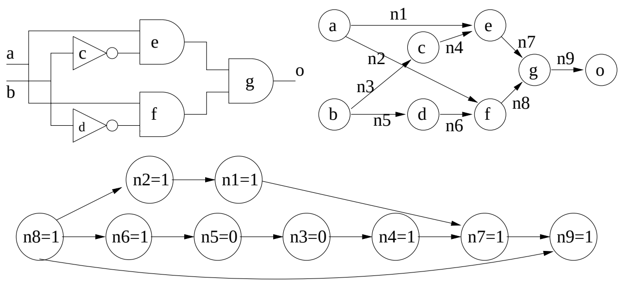

Figure 8: An implication graph

tions guarantees that all such redundancies can be removed simultaneously. These redundancies are used to simplify the network. The process is repeated until all nets have been considered. Note that the function *redundancy_remove* considers assumptions on only a single net but in general any number of nets could be used to generate assumptions. We later show results for the case when we considered assumptions on two nets, the second one corresponding to the unjustified node closest to n_i. This is an instance of recursive learning.

We now formalise the notion of a valid label as one for which an *implication graph* exists. We will use the notion of implication graph for proving the compatibility of the set of labels generated by the algorithm. Let A be an assumption on a set P of nets. An *implication graph* for the label $(n = r)$ from assumption A is a directed acyclic graph $G_I = (V_I, E_I, L_I)$, where L_I is a set of labels of the form $(m = a)$ for some net m and some $a \in \{0, 1, \otimes\}$ labelling every vertex $v \in V_I$, such that

- Every root [1] vertex is labelled with $(m = a)$ where $A : m \mapsto a$
- There is exactly one leaf [2] vertex $v \in V_I$ which is labelled $(n = r)$
- For any vertex $v \in V_I$, if v is not a root node the implication labelling it can be obtained from the implications labelling its parents by invoking an inference rule.

An example of an implication graph for the label $(n9 = 1)$ from the assumption $\langle n8 \mapsto 1\rangle$ is shown in Figure 8. A set of labels $\mathcal{C}$ derived from an assumption A is *compatible* if for every label $C \in \mathcal{C}$ there exists an implication graph $G_C = (V_C, E_C, L_C)$ of C from A such that $L_C \subseteq \mathcal{C}$.

We now prove the compatibility of implications returned by our labelling procedure. At each step, the labelling procedure either labels a node for the first time or overwrites a constant with a $\otimes$. We prove the invariant that at any time, the current set of implications $\mathcal{C}$ is compatible. We must prove that if a label is overwritten with a new label, every other label must have an implication graph which does not depend on the overwritten label. This claim is proved in the following lemma and is needed for all current labels to be simultaneously valid. Note that overwriting a 0 with a 1 (or vice-versa) implies an inconsistent assumption and the procedure exits.

Lemma 2.1 *Let A be a consistent assumption. If a label $(m = a)$ is overwritten by the label $(m = \otimes)$ in the current set of labels, then for all labels $(n_j = b_j)$, there is an implication graph such that $(m = a)$ is not a label of any vertex in the graph.*

Proof: We call net m a parent of net n if there is a node v of the circuit graph such that m is an incoming arc and n an outgoing arc of v. We also say that n is a child of m. We say m is a sibling of n if there is a node v such that both m and n are outgoing edges of v.

We prove the claim by contradiction. Suppose it is false. Let the replacement of $(m = a)$ by $(m = \otimes)$ be the first instance that makes it false. Therefore, there was an implication graph for each current implication before this happened. Let $(n_j = b_j)$ be an implication that does not have a valid implication graph now. Consider any path in the old implication graph for a net n_j, $(n_1 = b_1) \to \cdots \to (n_j = b_j)$, such that $(m = a)$ is the ith implication on the path. We consider the case where b_j is a constant. Hence, all b_k's in the path are constants since a $\otimes$ at a net can only imply a $\otimes$ at another. The case in which $b_k = \otimes$ is considered later. We show that if the assumption A is consistent then it is possible to replace $n_i = b_i$ in the implication graph for n_j. There are three cases on the relation between n_{i-1} and n_i.

Case 1: The circuit edge n_{i-1} is a child of n_i. $\otimes$ can be inferred at n_i only if either $n_{i-1} = \otimes$ is a current implication or $n_{i'} = 0$ is a current implication and $n_{i'}$ and n_i are inputs to an AND gate. In the first case, the fact that an implication graph existed in which n_{i-1} was labelled with a constant is contradicted. In the second case, n_{i-1} is the output of an AND gate, whose two inputs are n_i and $n_{i'}$. Since $(n_{i-1} = b_{i-1}) \to (n_i = b_i)$ is a valid inference, either $n_{i-1} = 1$ (to imply $n_i = 1$ and $n_{i'} = 1$) or $n_{i-1} = 0$ and $n_{i'} = 1$ (to imply $n_i = 0$). In either case $n_{i'} = 1$ which contradicts $n_{i'} = 0$.

Case 2: n_{i-1} and n_i are siblings and $(n_{i-1} = b_{i-1}) \to (n_i = b_i)$ is an application of Rule C6. If n_{i+1} is either the parent or a sibling of n_i then $n_i = b_i$ can be removed from the implication graph for $n_j = b_j$, i.e., $(n_{i-1} = b_{i-1}) \to (n_{i+1} = b_{i+1})$ is a valid implication. If n_{i+1} is a child of n_i, then $\otimes$ can be inferred at n_i only if either $n_{i+1} = \otimes$ is a current implication or $n_{i'} = 0$ is a current implication and $n_{i'}$ and n_i are inputs to an AND gate. In the first case, the fact that an implication graph existed in which n_{i+1} was labelled with a constant is contradicted. In the second case, clearly n_{i+1} is labelled with 0, i.e., $b_{i+1} = 0$ otherwise the assumption A is inconsistent, and the path $(n_i = b_i) \to (n_{i+1} = b_{i+1})$ can be replaced by the path $(n_{i'} = 0) \to (n_{i+1} = 0)$. Note that to get a new implication graph for $n_j = b_j$, we need the implication graph for $n_{i'} = 0$ but that exists and is not affected by the overwriting of the previous label of

[1]A vertex with no incoming edges
[2]A vertex with no outgoing edges

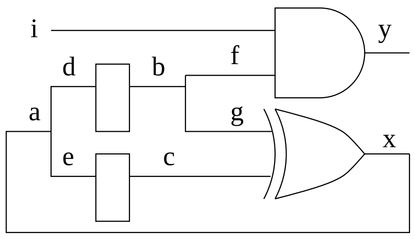

Figure 9: Sequential circuit C

n_i with $\otimes$.

Case 3: n_{i-1} is a parent of n_i. The reasoning is same as in Case 2.

Thus we have shown that if the assumption was consistent, each vertex labelled with $(n_i = b_i)$ in the implication graph of a current implication $(n_j = b_j)$ can be replaced with some other current implication. This shows that the replacement of $n_i = b_i$ by $n_i = \otimes$ does not falsify the claim which is a contradiction.

Now we consider the case in which $b_j = \otimes$. Then, there is a greatest k such that b_k is a constant, b_l is constant for all $1 \leq l \leq k$, and $b_l = \otimes$ for all $k < l \leq j$. From the proof before, we know there exists an implication graph for $n_k = b_k$ in which $n_i = b_i$ is not used. This yields an implication graph for $n_j = b_j$ in which $n_i = b_i$ is not used. ■

Lemma 2.2 *Let A be a consistent assumption. Then the set of labels returned by the algorithm is compatible.*

Proof: At each step in the algorithm, either a value is implied at a net for the first time or a constant is overwritten by a $\otimes$. The proof of this lemma follows by induction on the number of steps of the algorithm and by using Lemma 2.1 to prove the induction step. ■

Theorem 2.1 *Let n_i stuck-at-v_i redundant, for all $1 \leq i \leq n$, be the set of redundant faults reported by the algorithm. Then the circuit obtained by setting $n_i = v_i$ for all $1 \leq i \leq n$ is combinationally equivalent to the original.*

2.2 Sequential redundancies

Now we extend the algorithm for combinational circuits described in the previous section to find sequential redundancies by propagating implications across latches. The implications may not be valid on the first clock cycle since the latches power-up nondeterministically and have a random boolean value initially. Nevertheless, we can use the notion of k-delayed replacement which requires that the modified circuit produce the same behaviour as the original only after k clock cycles have elapsed. Thus, for example, if implying constant v at a latch output from constant v at its input yields a redundancy, a 1-delay replacement [3] is guaranteed on the removal of

[3] If we have latches where a reset value is guaranteed on the first cycle of operation, it is sufficient to ensure that the constant v is equal to the reset value; in this case the replacement is a 0-delay replacement.

$$\begin{tikzcd}
 & e^t = 0 \arrow[r] & c^{t+1} = 0 \arrow[drr] & & & & \\
a^t = 0 \arrow[r] \arrow[ur] & d^t = 0 \arrow[r] & b^{t+1} = 0 \arrow[r] & g^{t+1} = 0 \arrow[r] & a^{t+1} = 0 \arrow[r] & e^{t+1} = 0 \arrow[r] & c^{t+2} = 0
\end{tikzcd}$$

Figure 10: A sequential implication graph from assumption $a^t = 0$ for the circuit C

$$\begin{tikzcd}
 & e^t = 1 \arrow[r] & c^{t+1} = 1 \arrow[drr] & & & & \\
a^t = 1 \arrow[r] \arrow[ur] & d^t = 1 \arrow[r] & b^{t+1} = 1 \arrow[r] & g^{t+1} = 1 \arrow[r] & a^{t+1} = 0 \arrow[r] & e^{t+1} = 0 \arrow[r] & c^{t+2} = 0
\end{tikzcd}$$

Figure 11: An incorrect sequential implication graph from assumption $a^t = 1$ for the circuit C

that redundancy.

The notion of a label in the implication graph is modified so that it also contains an integer time offset with respect to a global symbolic time step t. The rules for learning implications are exactly the same as before with the addition of a new rule which allows us to propagate implications across latches: when we go across a latch we modify the time offset accordingly, e.g. if the output of a latch is labelled with 1 and offset -2, the input of the latch can be labelled with 1 and offset -3. An example of an implication graph for the circuit C in Figure 9 is shown in Figure 10.

This example also shows a potential problem with learning sequential implications. Consider the circuit C in Figure 9. For the two assumptions $\langle a^t \mapsto 0\rangle$ (a is 0 at t and t denotes the global symbolic time) and $\langle a^t \mapsto 1\rangle$ we get two implication graphs (in Figures 10 and 11) which both imply $(c^{t+2} = 0)$. This might lead us to believe that the $c = 0$ is a (2-cycle) redundancy. However, the new circuit obtained by replacing c with 0, if it powers up in state 11 (each latch at 1), remains forever in 11 with the circuit output $x = 1$. However, the original circuit produces $x = 0$ for all time $t \geq 1$, no matter which state it powers up in. Thus we do not have a k-delay replacement for any k. The reason for this incorrect redundancy identification is that in order to infer $(c^{t+2} = 0)$ from the assumption $\langle a^t \mapsto 1\rangle$, we needed $(c^{t+1} = 1)$. However, if we replace net c with 0 (i.e., for all times), c could not have been 1 at $t + 1$.

One way of solving the above problem is to ensure that no net is labelled with different labels for different times. We will label a net with at most one label, and if a net is labelled we will associate a list of integers with this label which denotes the time offset when this label is valid. Thus, for the above example, during the implication propagation phase for the assumption $\langle a^t \mapsto 1\rangle$ we will never infer $(a^{t+1} = 0)$ and we will not get the second implication graph in Figure 10. Labeling one net with at most one label also obviates the need for the validation step described in [1].

The algorithm replaces a net n with the constant v if for some time offset t', it is either labelled with v or is unobservable for all assumptions. With each such replacement, we associate a time k as follows [1]. To validate a redundancy n *stuck-at-v* at time t', we have a set of implication graphs, one for each assumption, that imply either $n^{t'} = v$ or $n^{t'} = \otimes$. Let t'' be the least time offset on any label in these implication graphs such that for some net m, $m^{t''}$ is labelled with a constant. Then $k = 0$ if $t'' > t'$ otherwise $k = t' - t''$. We say that n is k-cycle *stuck-at-v* redundant. We use the following theorem to claim that the circuit obtained by replacing net n with constant v is a k-delayed safe replacement.

Lemma 2.3 ([1]) *Let a net n be k-cycle stuck-at-v redundant. Then the circuit obtained by setting net $n = v$ results in a k-delayed safe replacement of the original circuit.*

As in the combinational case, we allow overwriting of constants with unobservability indicators. We make sure that the label at net n at time $t + a$ is overwritten only if the new label is $\otimes$ and net n is not labelled at any other time offset (this is to prevent the problem shown in Figure 11). This may make our algorithm dependent on the order of application of rules, but we have not explored the various options. The proof of the following two lemmas follows by easy extensions of Lemmas 2.1 and 2.2.

Lemma 2.4 *Let A be a consistent assumption. If a label $m^t = a$ is replaced with $m^t = \otimes$ in the current set of labels, then for all labels $m_j^{t'} = b_j$, there is an implication graph such that $m^t = a$ is not a label in the graph.*

Lemma 2.5 *Let A be a consistent assumption. Then the set of labels returned by the algorithm is compatible.*

Hence, the redundancies reported by the algorithm are compatible with each other and all redundancies can be removed simultaneously to get a delayed safe replacement.

Theorem 2.2 *Let n_i k_i-cycle stuck-at-v_i redundant, for all $1 \le i \le n$, be the set of redundant faults reported by the algorithm. Let $K = \Sigma_{1 \le i \le n} k_i$. Then, the circuit obtained by setting net $n_i = v_i$ forall $1 \le i \le n$, is a K-delay safe replacement of the original.*

Proof: From Lemma 2.5, we know from that for all $1 \le i \le n$, n_i is k_i-cycle stuck-at-v_i redundant in the circuit obtained by setting $n_j = v_j$ for all $j \neq i$. It has been shown in [5] that for any circuits C, D and E, if C is an a-delay replacement for D and D is a b-delay replacement for E then C is $(a + b)$-delay replacement for E. The desired result follows easily by induction on n from this property of delay replacements. ∎

Circuit	Redundancy Removal				With Recursive Learning			
Name	red	LR	A1	%	red	LR	A2	%
s349	0	0	345	0	0	0	345	0
s382	0	0	437	0	0	0	437	0
s386	0	0	251	0	0	0	251	0
s499	0	0	605	0	0	0	605	0
s526	1	0	472	0.4	1	0	472	0.4
s820	0	0	499	0	1	0	492	1.4
s832	0	0	456	0	1	0	443	2.8
s953	0	0	920	0	2	0	717	22.6
s1238	0	0	998	0	3	0	890	5.4
s1269	0	0	1140	0	20	0	1100	3.5
s1488	0	0	1034	0	0	0	1034	0
s1512	0	0	1337	0	0	0	1337	0
s3271	0	0	2828	0	0	0	2828	0
s3384	0	0	3775	0	1	0	3767	0.2
s4863	0	0	3368	0	0	0	3368	0
s5378	29	3	3538	2.1	30	4	3504	3.0
s9234	0	0	2854	0	2	0	2795	2.1
s13207	11	3	7590	0.4	22	5	7509	1.4
s15850	103	0	10571	1.3	113	0	10495	2.0
s35932	64	0	32006	0.3	64	0	32006	0.3
s38417	183	2	32746	0.9	219	41	31917	3.4
s38584	144	0	29069	0.5	124	0	28167	3.6
cordic	80	0	10636	0.5	81	0	16360	0.5

For legend see Table 2.

Table 1: Experimental results for combinational redundancies

3 Experimental Results

We present some experimental results for this algorithm. We demonstrate that our approach of identifying sequential redundancies yields significant reduction in area and is better than the approach which removes only combinational redundancies. We also show that for most examples, recursive learning gives better results then the simple implication propagation scheme. In fact for many circuits, recursive learning could identify redundancies where the simple implication propagation scheme is unable to find any.

This algorithm was implemented in SIS [11]. The circuit was first optimised using `script.rugged` which performs combinational optimisation on the network. The optimised circuit was mapped with a library consisting of 2-input gates and inverters. The sequential redundancy removal algorithm was run on the mapped circuit. The propagation of implications was allowed to propagate 15 time steps forward and 15 timesteps backward from the global symbolic time. Table 2 shows the mapped (to MCNC91 library) area of the circuits obtained by running `script.rugged` and that obtained by starting from that result and applying redundancy removal algorithm. For very large circuits (s15850 and larger), BDD operations during the `full_simplify` step in `script.rugged` were not performed. We report results for those circuits on which our algorithm was able to find redundancies.

As mentioned earlier, our algorithm starts with an assumption on the nets and implies values on other nets of the circuit. We implemented two flavors of selection of assumptions. In

Circuit	Attributes				Redundancy Removal						With Recursive Learning					
Name	PI	PO	L	A	red	C	LR	A1	%	time	red	C	LR	A2	%	time
s349	9	11	15	345	0	0	0	345	0	0.5	9	105	0	330	4.3	1.0
s382	3	6	21	436	0	0	0	437	0	0.9	2	5	0	434	0.7	2.1
s386	7	7	6	251	1	1	0	245	2.5	0.5	1	2	0	245	2.5	0.9
s499	1	22	22	605	19	32	0	583	3.6	5.1	16	30	0	581	4.0	9.9
s526	3	6	21	480	1	1	0	472	0.4	0.8	1	0	0	472	0.4	2.3
s820	18	19	5	499	0	0	0	499	0	1.7	1	0	0	492	1.4	2.9
s832	18	19	5	456	0	0	0	456	0	1.6	2	0	0	431	5.5	2.6
s953	16	23	29	920	0	0	0	920	0	3.7	3	0	10	632	31.3	6.3
s1238	14	14	18	998	0	0	0	998	0	3.5	3	0	0	890	10.1	5.4
s1269	18	10	37	1140	0	0	0	1140	0	2.9	21	8	0	1094	4	3.8
s1488	8	19	6	1034	0	0	0	1034	0	5.0	154	149	0	863	16.5	8.7
s1512	29	21	57	1337	2	2	0	1333	0.3	3.3	147	146	9	1092	18.3	5.6
s3271	26	14	116	2828	0	0	0	2828	0	12.9	6	5	0	2801	1.0	25.5
s3384	43	26	183	3775	0	0	0	3775	0	15.8	4	1	1	3745	0.8	18.7
s4863	49	16	83	3386	80	160	0	3319	1.9	23.1	82	164	0	3313	2.2	33.4
s5378	35	49	163	3616	574	1995	25	2959	19.6	22.0	1145	6992	58	2261	37.5	19.9
s9234	19	22	138	2854	102	1414	0	2752	3.8	22.8	102	1414	0	2752	3.8	22.4
s13207	31	121	453	7681	49	518	28	7035	8.4	66.9	92	733	70	6317	17.8	32.1
s15850*	14	87	540	10704	199	1841	6	10415	2.7	272.4	163	1650	43	9380	10.0	493.7
s35932*	35	320	1728	32092	64	0	0	32006	0.3	1339.4	64	0	0	32006	0.3	5010.3
s38417*	28	106	1464	33055	591	887	42	31943	3.4	1139.4	1129	9245	97	29718	10.1	1763.7
s38584*	12	278	1285	29252	102	168	0	29016	0.8	1193.5	114	400	5	28656	2.9	2157.4
cordic*	35	9	271	10688	81	73	0	10636	0.5	251.8	66	56	8	8939	16	242.6

* `full_simplify` not run.
All times reported on an Alpha 21164 300MHz dual processor with 2G of memory.

PI	number of primary inputs	PO	number of primary outputs
L	number of latches	A	Mapped area after `script.rugged`
A1	Mapped area after redundancy removal	A2	Mapped area after redundancy removal with recursive learning
red	number of redundancies removed	LR	Number of latches removed
C	Upper bound on *c*, where the new circuit is a *c*-delay replacement	time	CPU time
%	Percentage area reduction		

Table 2: Experimental results for sequential redundancies

the first case a conflicting assignment was assumed on one net and values were implied on other nets. The second case was similar to the first except that once the implications could not propagate for an assumption on a net, we performed a naïve version of case splitting only on the net which was closest to the original net from which the implications were propagated and implications common in the two cases were also added in the set of implications learnt for the original net.[4] This enabled us to propagate implications over a larger set of nets in the network and hence to discover more redundancies at the expense of CPU time. Table 2 indicates the area reduction obtained both by simple propagation and by performing this recursive learning. We find that even for this naïve recursive learning we get reduction in area in most of the circuits over that obtained without case split. For instance, for S5378 we were able to obtain 37.5% area reduction with recursive learning as against 19.6% without it. For most of the medium sized circuits we were not able to obtain any reduction in area without recursive learning. For large circuits also we were able to obtain approximately 5-10% area reduction. S35952 was an exception where we did not obtain any more reduction in area. Except for this circuit the CPU time for recursive learning was less than twice the CPU time for redundancy removal without it. This suggests that more sophisticated recursive learning based techniques could yield larger area reduction without prohibitive overhead in terms of CPU time.

[4]If a node is unjustified during forward propagation of implications then case-split is performed by setting the output net to 0 and 1. If the node is unjustified during backward propagation case split is achieved by setting one of the two inputs to the input controlling value (0 for (N)AND gate and 1 for (N)OR gate) at a time and propagating the implications backward.

Since our algorithm also identified combinational redundancies, we wanted to quantify how many of the redundancies were purely combinational. To verify this we ran our algorithm on the circuits for combinational redundancy removal only. Table 1 shows the area reduction due to combinational redundancies only with and without recursive learning. In most cases, the number of redundancies identified in Table 2 is significantly larger than the set of combinational redundancies identified by our algorithm. Only for S35952 and S953 did the combinational redundancy removal result in approximately the same area reduction as the sequential redundancy case.

For the example circuits presented here we were able to achieve 0-37% area reduction. In a number of cases the algorithm was able to remove a significant number of latches. In all cases, the new circuit is a *C*-delay safe replacement of the original circuit. The *C* reported in Table 2 is actually an upper bound. For most of the delay replaced circuits $C < 10000$. However most practical circuits operate at speeds exceeding 100 MHz in present technology. $C < 10000$ for a circuit would require the user to wait for at most 100 μs before useful operation can begin. This is not a severe restriction.

We are unable to compare sequential redundancy removal results with the previous work of Entrena and Cheng [8] because as we noted earlier, their notion of sequential replacement, which is based on the conservative 0,1,X-valued simulation, is not compositional (unlike the notion of delay replacement that we use).

4 Future Work

Our redundancy removal algorithm does not find the complete set of redundancies. We can extend this scheme in several ways to identify larger sets. For instance, instead of analyzing two assumptions due to a case split on a single net we could case split on multiple nets and intersect the implications learnt on this larger set of assumptions. One such method is to incrementally select those which are at the frontier where the first phase of implications died out. Additionally, if we split on multiple nets it is possible to detect pairs of nets such that if one is replaced with another the circuit functionality does not change. With our current approach, because we split on a single net, one of the nets in this pair is always a 1 or a 0, which means that we are only identifying stuck-at-constant redundancies.

For this algorithm we map a given circuit using a library of two input gates and inverters. A different approach would be to use the original circuit and propagate the implications forward and backward by building the BDD's for the node function in terms of it's immediate fanins. We intend to compare the running times and area reduction numbers of our approach with such a BDD based approach. In addition, BDD based approaches may allow us to do redundancy removal for multi-valued logic circuits as well in a relatively inexpensive way. We can extend the notion of redundancy for multi-valued circuits to identify cases where a net can take only a subset of its allowed values. Then latches of this kind can be encoded using fewer bits.

5 Acknowledgements

We had useful discussions with Mahesh Iyer during the course of this work. Miron Abramovici gave us the example of Figure 1. The comments by the referees also helped to improved the paper.

References

[1] M. A. Iyer, D. E. Long, and M. Abramovici, "Identifying Sequential Redundancies Without Search," in *Proc. of the Design Automation Conf.*, (Las Vegas, NV), pp. 457–462, June 1996.

[2] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney, "The Transduction Method - Design of Logic Networks Based on Permissible Functions," in *IEEE Trans. Computers*, Oct. 1989.

[3] H. Savoj, *Don't Cares in Multi-Level Network Optimization*. PhD thesis, University of California Berkeley, Electronics Research Laboratory, College of Engineering, University of California, Berkeley, CA 94720, May 1992.

[4] W. Kunz and D. K. Pradhan, "Recursive Learning: A New Implication Technique for Efficient Solution to CAD Problems - Test, Verification and Optimization," *IEEE Trans. Computer-Aided Design*, Sept. 1994.

[5] V. Singhal, C. Pixley, A. Aziz, and R. K. Brayton, "Exploiting Power-up Delay for Sequential Optimization," in *Proc. European Design Automation Conf.*, (Brighton, Great Britain), pp. 54–59, Sept. 1995.

[6] S. Qadeer, V. Singhal, R. K. Brayton, and C. Pixley, "Latch Redundancy Removal without Global Reset," in *Proc. Intl. Conf. on Computer Design*, (Austin, TX), pp. 432–439, Oct. 1996.

[7] M. Chatterjee, D. K. Pradhan, and W. Kunz, "LOT: Logic Optimization with Testability - New Transformations using Recursive Learning," in *Proc. Intl. Conf. on Computer-Aided Design*, (San Jose, CA), pp. 318–325, Nov. 1995.

[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, *On Redundancy and Untestability in Sequential Circuits*. PhD thesis, Illinois Institute on Technology, 1995.

[10] I. Pomeranz and S. M. Reddy, "On Removing Redundancies from Synchronous Sequential Circuits with Synchronizing Sequences," *IEEE Trans. Computers*, vol. 45, pp. 20–32, Jan. 1996.

[11] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldanha, H. Savoj, P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "SIS: A System for Sequential Circuit Synthesis," Tech. Rep. UCB/ERL M92/41, Electronics Research Lab, Univ. of California, Berkeley, CA 94720, May 1992.