Symbolic Computation of Logic Implications for Technology-Dependent Low-Power Synthesis [*]

R. I. Bahar [†] M. Burns [‡] G. D. Hachtel [‡] E. Macii [§] H. Shin [‡] F. Somenzi [‡]

[†] Brown University
Division of Engineering
Providence, RI 02912

[§] Politecnico di Torino
Dip. di Automatica e Informatica
Torino, ITALY 10129

[‡] University of Colorado
Dept. of ECE
Boulder, CO 80309

Abstract

This paper presents a novel technique for re-synthesizing circuits for low-power dissipation. Power consumption is reduced through redundancy addition and removal by using learning to identify indirect logic implications within a circuit. Such implications are exploited by adding gates and connections to the circuit without altering its overall behavior and thereby enabling us to eliminate other, high power dissipating, nodes. We propose a new BDD-based method for computing indirect implications in a logic network; furthermore, we present heuristic techniques to perform redundancy addition and removal without destroying the topology of the mapped circuit. Experimental results show the effectiveness of the proposed technique in reducing power while keeping within delay and area constraints.

1 Introduction

Excessive power dissipation in electronic circuits reduces reliability and battery life. The severity of the problem increases with the level of transistor integration. Therefore, much work has been done on power optimization techniques at all stages of the design process (e.g., high-level, logic-level, and circuit-level). At the logic level–the focus of this paper–the main objective of low-power synthesis algorithms is the reduction of the switching activity of the logic, weighted by the capacitive load. Early work only considered optimization at the technology-independent stage of the synthesis flow. However, since it is difficult to measure the power dissipation of technology independent circuits with a dependable level of accuracy, techniques such as technology decomposition [1], technology mapping [1, 2, 3] and gate re-sizing [4] have shown to be effective.

In this paper we propose a method that can be applied to technology mapped circuits, and that is based on the idea of reducing the total switching activity of the network through redundancy addition and removal. We start from a circuit which is already implemented in gates from a technology library, and perform power analysis on it to identify its high power dissipating nodes. We use a sophisticated learning mechanism (related to those of [6, 7, 8]) to find logic implications in the circuit in the neighborhood of these nodes. Such implications are used to identify network transformations which add and remove connections in the circuit, with the objective of eliminating the high power nodes or connections from them. Although other redundancy addition and removal techniques have been proposed (e.g., [5, 8, 10]), our method is innovative in two main aspects; first, it uses a powerful learning procedure based on symbolic calculations which allows the identification of very general forms of logic implications; second, it operates at the technology dependent level; this allows more accurate power estimates to drive the overall re-synthesis process. Experimental results, obtained on a sample of `Mcnc'91` benchmarks [9], show the viability and the effectiveness of the proposed approach.

[*]This work is supported in part by NSF/DARPA grant MIP-9115432 and SRC contract 94-DJ-560.

2 Redundancy Addition and Removal

A circuit may fail due to a wire being physically connected to the power source or ground, where the failure may be observed as a *stuck-at fault*; under this failure, the circuit behaves as if the wire were permanently stuck-at-1 or 0. Let C be a combinational circuit, and let C_f be the same circuit in which fault f is present. Fault f is *redundant* if and only if the output behaviors of C and C_f are identical for any input vector applied to both C and C_f.

Any automatic test pattern generation program may be used to detect redundant faults (e.g., [11]). The computed information may be used to simplify the network by propagating the constant values (0 or 1), due to stuck-at connections, throughout the circuit. The effectiveness of redundancy removal greatly depends on the number of redundancies. For circuits that are 100% testable, redundancy removal does not help. For this reason, techniques based on redundancy addition and removal have been proposed.

We illustrate this idea through an example. Consider the circuit shown in Figure 1(a), in which gate $G9$ is initially an inverter. All stuck-at-faults in the circuit are testable. Therefore, no simplification through redundancy removal is possible on the network as is. However, consider in particular the testable stuck-at-1 fault f at the output of gate $G2$. If the inverter $G9$ is transformed into a 2-input NAND gate and the additional input is connected from the output of gate $G2$ (see dotted line in the figure), then the behavior of the circuit is unchanged and fault f becomes redundant. The circuit now can be simplified as shown in Figure 1(b).

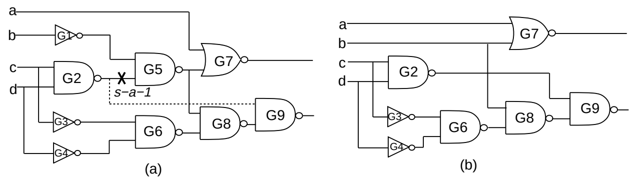

Figure 1: Circuit with Added and Removed Redundancies.

In general, there may be a large variety of choices available in selecting the new connection (and logic gates) that may be added to the original circuit so as to introduce redundancies. Kunz and Menon have proposed an effective solution for selecting these connections through a method derived from *recursive learning* [8]. This process is equivalent to finding *implications* in the circuit; that is, finding all the value assignments necessary in order that a given set of value assignments are satisfied.

Consider again the circuit in Figure 1(a), where the signal assignment $G8 = 1$ has been made. We may optionally assign either $G5 = 0$ or $G6 = 0$ to satisfy this assignment, however neither assignment is *necessary.* We therefore conclude that no additional assignment can be *directly implied* given the assignment $G8 = 1$. However, upon closer inspection, we can determine that the value assignment $G8 = 1$ *indirectly implies* the value assignment $G2 = 1$. If we temporarily assign $G6 = 0$, $G3 = G4 = G2 = 1$ and $c = d = 0$ are all assignments necessary to satisfy $G6 = 0$. Likewise, by temporarily assigning $G5 = 0$ we find that $G1 = G2 = 1$ and $b = 0$ are necessary assignments. In either case, $G2 = 1$ is a necessary assignment so we conclude that $G2 = 1$ is a necessary assignment for, and is indirectly implied by, $G8 = 1$.

Of particular interest to us are these *indirect implications*, which cannot be found through simple propagation of signal values in the circuit and must instead be obtained through recursive learning.

3 Computing Implications Symbolically

We now introduce our symbolic procedure to compute logic implications through recursive learning.

Suppose we are given a set $T = \{T_j\}$ of relations $T_j \subseteq B^n \times B^n$, where $B = \{0, 1\}$, on the implied universe defined by n Boolean variables $y = (y_0, \ldots, y_{n-1})$. In our context, we can think of each T_j as the transition relation $y_j \equiv f_j(y)$ for a single gate j. For example, if j were a NAND gate with inputs y_i and y_k, $T_j = y_j(y_i y_k)' + y_j'(y_i y_k)$. T_j is then a set of consistency assignments for the gates such that the variables are assigned values consistent with the behavior of the gates. In this way, if the Boolean variables y_i are assigned such that $T_j = 0$, then the variable assignments are inconsistent with the correct behavior of the gate.

Suppose further that we are given an initial assignment, or assertion, $A(y)$. We may compute its "implications" on T as follows:

$$\begin{aligned} I(y) &= A(y) \prod_j T_j(y), \\ c(y) &= \text{Essential}(I(y)). \end{aligned} \tag{1}$$

Here procedure Essential computes the literal factors of $I(y)$, that is, the variable assignments $y_i = 1$ or $y_i = 0$ $(0 \leq i < n)$ that are required to make the assertion consistent with all the relations in the set T. We have three cases:

1. $c(y) = 0$ (empty cube). $A(y)$ is inconsistent with one or more of the T_j's.
2. $c(y) = 1$ (tautology or cube of no literals). $A(y)$ does not require any variable to assume a constant value.
3. $c(y)$ denotes the product of one or more single variable functions. For example, $c(y) = y_1 \overline{y}_3$ means that consistency of the assertion with the relations requires $y_1 = 1$ and $y_3 = 0$.

3.1 Direct Implications

At the heart of the implication computation is procedure impSatDirect of Figure 2, which finds direct satisfiability implications of a given logical assertion, with respect to a specified set of logic relations. There are two inputs to impSatDirect. The first is the assertion, or premise, $A(y)$. The second is the relation set $T = \{T_j\}$. The procedure returns a cube c^0, representing the set of variables that are directly implied to constant values by the assertion A. In our implementation, T, $A(y)$, and c^0 are all represented by BDDs.

```
     **Procedure** impSatDirect($A$,  $T$)  {
1.       $c^0$ = Essential($A$)
         $A^0$ = Cofactor($A, c^0$)
         $T^0 = T$
2.       $Q = \{k | T_k^0$ is a fanout of some $x_j \in c^0\} \cup \{x_j | x_j \in c^0\}$
         **while** ($Q \neq \emptyset$) {
             $k$ = select_and_remove_one($Q$)
             $T_k^0$ = Cofactor($T_k, c^0$)
3.           **if** ($T_k^0 \equiv zero$)  **return**($zero, zero, \{T_i^0\}$)
4.           $t$ = Essential($T_k$)
5.           **if** ($t \equiv one$)  **continue**
             $A^0$ = Cofactor($A^0, t$)
6.           **if** ($A^0 \equiv zero$)  **return**($zero, zero, \{T_i^0\}$)
7.           $u$ = Essential($A^0$)
             $A^0$ = Cofactor($A^0, u$)
8.           $t = u \cdot t$
9.           $c^0 = c^0 \cdot t$
10.          $Q = Q \cup \{k_t | k_t$ fanout of $x_j \in t\} \cup \{x_j | x_j \in t\}$
         }
         **return**($c^0, A^0, \{T_i^0\}$)
     }
```

Figure 2: Procedure impSatDirect.

Procedure Essential finds a cube, c^0, such that $A = c^0 \cdot A^0$, and A^0 is cube-free. If $c^0 = one$ (i.e., there are no single literal functions that factor A), we skip the **while** loop and immediately return. Otherwise, we consider, one at a time, all possible direct implications on the gates from the set Q, containing all fanouts of variables in the support of the cube c^0, plus all the variables in c_0 itself.

The first step of the **while** loop selects one gate from Q, and removes it from the set. Inside the **while** loop procedure Essential is called, according to Equation 1, to discover any new implication literals (lines 4 and 7). We break out of the **while** loop and return whenever T_k^0 or A^0 reduces to *zero*, implying the assertion A is logically inconsistent with one of the relations in the set of transition sub-relations $\{T_i^0\}$.

The literal function t in line 8 represents newly discovered direct implications of the original assertion A. On each pass through the **while** loop, these new implications are added to the global implications cube c^0 (line 9), effectively accumulating all the implications of the original assertion A. As these implication variables are found, they are also appended to the set Q (line 10).

3.2 Indirect (Recursively Learned) Implications

Procedure impSatBexp, shown in Figure 3, finds all satisfiability implications of a given assertion $A(y)$. This is done by first calling impSatDirect to find the direct implications, and then by performing a recursive Boole expansion, with impSatDirect as the terminal case and procedure cubeIntersect as the merge step. Implications common to all leaves of the recursion are retained as recursively learned implications.

```
    Procedure impSatBexp($A$, $T$, $level$) {
1.      $(c^0, A^0, T^0)$ = impSatDirect($A, T$)
2.      if ($A^0 \equiv zero$) return($zero, zero$)
3.      if ($\forall i, (T_i^0 \equiv one)$) return($c^0, one$)
4.      if ($level \geq$ MAXLEVEL) return($c^0, one$)
5.      $y_j$ = selectVariable($c^0$)
6.      $(c^+, c^{ind+})$ = impSatBexp($A^0 \cdot y_j,\ T^0,\ level+1$)
7.      $(c^-, c^{ind-})$ = impSatBexp($A^0 \cdot \overline{y}_j,\ T^0,\ level+1$)
8.      $c^{ind}$ = cubeIntersect($c^+, c^-$)
9.      $impCube = c^0 \cdot c^{ind}$
        return($impCube, c^{ind}$)
    }
```

Figure 3: Procedure impSatBexp.

As with procedure impSatDirect, impSatBexp takes as input the assertion A and the relation set T. In addition, it takes an input, *level*, representing the recursion level of the recursive call, initially set to 0. When *level* exceeds a specified limit MAXLEVEL, the search for further implications is abandoned. The first output of impSatBexp is the cube *impCube*, representing the set of all variables that are implied to constant values. The procedure also returns c^{ind}, the cube consisting of only the indirectly implied variables.

The efficiency of impSatBexp depends highly on the criterion used to select the splitting variable for the expansion. The heuristic we use considers all the variables y_i in the support of cube c^0. The splitting variable y_j is chosen such that it is in the support of the transition relation of some variable y_i in c^0 but is not contained in the cube c^0 itself. This heuristic effectively selects some fanin to a gate whose output, but not all its input values, have been determined by implication.

Using BDDs to store and compute indirect implications may seem inefficient compared to doing a simple analysis of the topology of a circuit. This may in fact be true if all we are interested in are single-variable implications derived from satisfiability assignments for single-literal assertions (for example, $(y_i = a) \Rightarrow (y_j = b)$ for $a, b \in \{0, 1\}$.) However, by computing the implications symbolically, we are able to handle more general implications as easily as simple ones. Assertions and implications need not be restricted to cubes and may be given as complex functions.

4 Redundancy Addition and Removal

We now describe our learning-based, re-synthesis procedure for reducing power dissipation. The procedure consists of six steps, described in detail in the following Sections 4.1– 4.6.

4.1 Building the Sub-Network

Computing all the indirect implications of a network as shown in Section 3 can be very computationally expensive. While the learning procedure can be pruned by limiting the recursion level (see Figure 3), this may not sufficiently reduce the time spent learning. Therefore, instead of searching the entire network for implications, we extract a sub-network and find implications within it. Since reducing power dissipation is our goal, we first consider sub-networks that contain a high power-dissipating gate.

Indirect implications are present specifically in circuits containing *reconvergent fanout.* Reconvergent fanout is defined as two or more distinct paths with a common input gate (or *fanout stem*) and leading to a common output gate. That is, given input gate i and output gate j, there exist at least two paths, $i \to k_1 \to k_2 \cdots \to j$, and $i \to l_1 \to l_2 \cdots \to j$, where k_m, l_n are gates and $k_m \neq l_n$, $\forall\ m, n$. The gate where the paths reconnect (gate j) is called the *reconvergence gate.* In order to spend our time learning where it is most useful, we limit our search for indirect implications to sub-networks containing reconvergent fanout. The sub-networks are found using procedure findSubnet shown in Figure 4.

```
    Procedure findSubnet($net$, $target$) {
1.    $fanouts = findGateFanouts(net, target)$
2.    $fanins = findGateFanins(net, target)$
3.    $stem\_list = findStems(fanins)$
      foreach ($stem \in stem\_list$) {
4.      $outbranch\_list = findOutbranch(net, stem)$
5.      $reconv\_list = findReconv(net, outbranch\_list, fanouts)$
6.      $stem\_fanouts = findStemFanouts(net, stem)$
        foreach ($gate_k \in reconv\_list$){
7.        $reconv\_fanins = findReconvFanins(net,\ gate_k)$
8.        $subgr_k = subgr_k \bigcup (stem\_fanouts \bigcap reconv\_fanins)$
        }
      }
9.    return ($subgr$)
    }
```

Figure 4: Procedure findSubnet.

The procedure accepts as parameters the circuit network and a target gate around which the sub-networks will be built. It returns a list of sub-networks, where each sub-network is given as a list of gates. The sub-network(s) containing that target gate are found by identifying common input gates (fanout stems) in the transitive fanin of the target gate, and their associated reconvergence gates in the transitive fanout of the target gate.

We perform a forward and backward search from the target gate (lines 1 and 2) up to a specified depth limit to obtain the lists *fanouts* and *fanins.* In line 3 we find the gates in *fanins* that have multiple fanout (i.e., fanout stems). For each *stem*, we identify gates fanning out from the stem and not contained in the list *fanins* (i.e., outbranches). These outbranch gates are used in line 5 to find any reconvergence gate. If a reconvergence gate exists we extract the transitive fanout list of the stem (*stem_fanouts*) in line 6 and the transitive fanin list of the reconvergence gate (*reconv_fanins*) in line 7. The sub-network then consists of the intersection of *stem_fanouts* and *reconv_fanins*. As it finds all sub-networks around the target gate, the procedure merges any sub-networks containing the same reconvergence gate. It then returns a list of all merged sub-networks found for that target gate.

For example, in the circuit of Figure 1(a), consider gate $G5$ as the target gate. Searching backwards from $G5$, one sees that inputs c and d are the only nodes with more than one fanout, so they are the only candidates for common inputs. Looking first at input c, there is one path to the transitive fanout of the target gate that does not go through the target gate itself (i.e., $c \rightarrow G3 \rightarrow G6 \rightarrow G8$). Since $G8$ can be reached from c by two paths, one of which includes the target gate $G5$, it is a reconvergence gate. The sub-network is then found by intersecting the transitive fanout of c with the transitive fanin of $G8$, yielding the sub-network $\{c, G2, G3, G5, G6, G8\}$.

Starting from node d, one finds the same reconvergence gate, $G8$, through a slightly different path. Rather than generating a second sub-network, procedure findSubnet appends the new gates to the old sub-network. In this manner, only one sub-network is generated per reconvergence gate. In our example, the final sub-network returned by findSubnet is $\{c, d, G2, G3, G4, G5, G6, G8\}$.

4.2 Finding the Indirect Implications

Once a sub-network containing the target gate has been extracted, we can proceed to search the sub-network for indirect implications. We start by identifying the reconvergence gate in the sub-network and use this gate to produce our assertion A which is passed to the implication procedure outlined in Figure 3. The reconvergence gate is asserted to both logic values 1 and 0. If the given assertion A is inconsistent within the sub-network, then the implication procedure, impSatBexp, returns $c^{ind} = zero$. This occurs if the reconvergence gate has a redundant connection. If, for the given assertion, no indirect implication is found, procedure impSatBexp returns $c^{ind} = one$. Otherwise, all the indirect implication gates contained in c^{ind} are saved and considered separately to guide our re-synthesis.

4.3 Finding the Right Addition

Once we find the indirect implications for the given assertion on the reconvergence gate, we use this information to add gates and/or connections to the circuit while preserving its original I/O behavior. We use a method similar to *data flow* analysis [6] to determine what modifications can be made for a given assertion $(x = i)$ and its implication $(y = j)$ for $i, j \in \{0, 1\}$, where the implication gate y is in the transitive fanin of assertion gate x.

Consider first the case where $(x = 0) \Rightarrow (y = 0)$. This implication can also be expressed as $\overline{x} \cdot y = 0$. Given the function $F(x) = x$, the implication can be expressed as the don't care condition, $F(x)_{DC}$, for $F(x)$. That is, $\overline{x} \cdot y \in F(x)_{DC}$. We may transform F to $\tilde{F}$ by adding this don't care term to the output of F without changing the behavior at the primary outputs of the circuit:

$$\tilde{F}(x) = F(x) + \overline{x} \cdot y = x + y = x, \text{ since } y = 0 \text{ if } x = 0.$$

The original circuit is modified by ORing the don't care term (i.e., the implicant gate y) with the output of gate x.

For the case where $(x = 1) \Rightarrow (y = 1)$, instead of using the don't care expression $x \cdot \overline{y} \in F(x)_{DC}$, we use the analogous expression $\overline{x} + y = 1$ and transform F to $\tilde{F}$ as:

$$\tilde{F}(x) = F(x) \cdot (\overline{x} + y) = x \cdot y = x, \text{ since } y = 1 \text{ if } x = 1.$$

In Figure 1(a) we found implication $(G8 = 1) \Rightarrow (G2 = 1)$. Identifying this implication allows us to modify the circuit by inserting the AND function $\tilde{G8} = G8 \cdot G2$ without changing the logical behavior at the primary outputs. Notice that this AND gate added to the network is "absorbed" by the inverter $G9$ which now becomes a 2-input NAND gate.

4.4 Finding the Redundancies

Once the redundant circuitry is added, we use the automatic test pattern generation (ATPG) procedure implemented in SIS [11] to find the new redundancies created in the network. Although gates are added to the network based on implications in the *sub-network*, finding and removing redundancies should be done on the *entire* network.

We generate a list of possibly redundant connections. Since the newly added gates are themselves redundant we need to make sure they are not included in the list. The result of redundancy removal is order dependent. Since our primary objective is reducing power dissipation, we sort the redundant connections in order of decreasing power and remove them starting at the top of the list.

4.5 Removing the Redundancies

After ATPG, the identified redundant faults can be removed with the ultimate goal of eliminating fanout connections from the targeted high power dissipating node. Redundancy removal procedures such as the one implemented in SIS cannot be used for this purpose for two main reasons. First, optimization occurs through restructuring of the Boolean network. As a consequence, even if redundancy removal operates on a technology mapped design, the end result of the optimization is a technology independent description that requires re-mapping onto the target gate library. This may lead to significant changes in the structure of the original network. This is undesirable in the context of low-power re-synthesis, since the network transformations made during re-synthesis are based on the original circuit implementation. Second, redundancy removal usually targets area minimization, and this may obviously affect circuit performance.

We have implemented our own redundancy removal algorithm, which resembles the Sweep procedure implemented in SIS, but operates on the *gates* of a circuit rather than on the nodes of a Boolean network. In addition, it performs a limited number of transformations. Namely, the procedure (a) simplifies gates whose inputs are constant, and (b) collapses inverter chains, while at the same time preserving the original circuit structure and performance.

Gate Simplification: Three simplifications are applicable to a given gate, G, when one of its inputs is constant:

1. If the constant value is a controlling value for G, then G is replaced by a connection to either V_{dd} or *Ground*, depending on the function of the gate.
2. If the constant value is a non-controlling value for G, and G has more than two inputs, then G is replaced by a gate, $\tilde{G}$, taken from the library and implementing the same logic function as G but with one less input.
3. If the constant value is a non-controlling value for G, and G is a two-input gate, then G is replaced by an inverter or buffer.

Usually, cell libraries contain several gates implementing the same function, but differing by their sizes and, therefore, by their delays, loads, and driving capabilities. We select, as replacement gate $\tilde{G}$, the gate that has approximately the same driving strength as the original gate G.

Inverter Chain Collapsing: Inverter chains are commonly encountered in circuits, especially in the cases where speed is critical. Collapsing inverters belonging to these "speed-up" chains, though advantageous from the point of view of area and, possibly, power, may have a detrimental effect on the performance of the circuit. On the other hand, the simplification of gates due to redundancy removal may produce inverter chains that may be easily eliminated without slowing down the network. We eliminate inverter chains only in the cases where the transformation does not increase the critical delay of the original circuit. For each inverter, $\tilde{G}$, obtained through simplification of a more complex gate, we first checks if $\tilde{G}$ belongs to a chain which can be eliminated. If certain constraints are satisfied, both the inverter $\tilde{G}$ and the companion inverter in the chain (i.e., the inverter feeding $\tilde{G}$ or the inverter fed by $\tilde{G}$) are removed. In particular, in order to safely remove the inverter chain:

1. The first inverter cannot have multiple fanouts.
2. The load at the output of the inverter chain is not greater than the load currently seen on the gate preceding the inverter chain.

The first restriction may be unnecessarily conservative; however, removing it implies that sometimes extra inverters need to be inserted on some of the fanout branches of the first inverter, thereby possibly introducing area, power, and delay degradation.

4.6 Choosing the Best Network

For the given assertions on the reconvergence gate, we create a new network for every indirect implication discovered by our learning procedure. We select the best network among them and replace our original network with it. Since reducing power dissipation is our primary goal, our first choice is the new network with the lowest power dissipation; however, our choice is also constrained by the critical delay of the new network. Therefore, we accept the new network with the lowest power dissipation that does not increase the critical delay of the original circuit by more than a fixed percentage—typically 5%.

5 Experimental Results

We have applied our optimization procedure to combinational circuits from the `Mcnc'91` suite [9]. Experiments were run within the SIS environment on a DEC Station 5000/200 with 88 MB of memory. We start with a circuit that has been first optimized using the SIS script `script.rugged` and then mapped for either delay (using `map -n 1 -AFG`) or area (using `map`.) The library used for the mapping consists of NAND, NOR, and inverter gates, each of which has 5 different drive options and up to 4 inputs.

After the circuit has been mapped, gates are selectively re-sized with smaller gates with no circuit delay penalty. The method of [4] is used for this purpose. The re-sized circuit is the starting point for the experiments. By re-sizing gates before applying our algorithm we have a more reliable indication that we are targeting nodes effectively (i.e., nodes are not targeted for re-synthesis because they are improperly sized). The initial statistics for these circuits are reported in Table 1. We show the number of gates, area (μm^2), delay ($nsec$), and power dissipation (μW) of the circuit before we apply our re-synthesis procedure. If there are any initial redundancies in the circuit they are first removed using the redundancy removal method outlined in Section 4.5. The power dissipation after this step is shown in column labeled *RR Power*. redundancy removal step has a negligible effect on power

We then apply our learning procedure to find indirect implications in the circuit and use them for redundancy addition and removal. After this step, we again re-size the circuit where possible without increasing the critical delay. Table 2 reports the final circuit statistics. In the experiments, only indirect implications that led to a decrease in power dissipation without increasing the critical delay of the circuit by more than 5% were retained. The number of accepted implications is shown in the columns labeled *I*. The relative changes in power, delay and area are shown in columns labeled ΔP, ΔD, and ΔA respectively.

Results show that power dissipation can be reduce substantially in some circuits by adding and removing redundancy through learned implications. In the case of mapping for area, a 23% power reduction was obtained for circuit `bw`. Of course, if no useful implications exist in the circuit, then this method will not reduce the power dissipation significantly (e.g., circuits `inc` and `misex2` mapped for speed). However, enough examples produced real improvements to indicate promise in our approach.

6 Conclusions and Future Work

Although there have been other techniques presented on redundancy addition and removal, our procedure outlined in this paper differs from them in the following ways. The effectiveness of the method in [5] is limited by the fact that a gate must already exist in the network in order to add redundant connections, whereas we have the flexibility of adding connections and gates in order to optimize a circuit. Our method for adding redundancies more closely matches that of [8]; however, we have set up a mechanism better suited to finding more general implications than Kunz. In addition, he may permit gate transformations that actually increase the power and/or delay. Finally, the main difference between our work and that proposed in [10] consists of

Circuit	*Initial Statistics (Mapping for Area)*					*Initial Statistics (Mapping for Speed)*				
	Gates	*Area*	*Delay*	*Power*	*RR Power*	*Gates*	*Area*	*Delay*	*Power*	*RR Power*
`5xp1`	106	142912	31.47	429	429	148	222488	20.20	1107	1107
`9sym`	159	236176	17.79	629	629	276	404608	10.92	1747	1747
`b12`	71	96512	9.96	277	277	97	130036	8.18	447	447
`bw`	124	172608	32.65	529	527	215	309256	17.95	1397	1388
`clip`	104	146624	17.43	427	427	167	249632	12.29	1205	1205
`cps`	897	1286208	40.09	1932	1932	1272	1694412	13.68	2772	2772
`inc`	81	116000	27.80	329	329	128	176784	17.81	628	628
`misex1`	50	66352	13.79	192	192	76	111244	9.94	525	525
`misex2`	74	104864	10.26	272	272	133	183164	7.55	551	552
`rd84`	125	174464	19.23	436	436	226	316796	12.69	1233	1233
`sao2`	105	148944	20.89	432	433	171	244992	15.19	966	966

Table 1: Circuit Statistics Before Learning.

Circuit	*Final Statistics (Mapping for Area)*								*Final Statistics (Mapping for Speed)*							
	Gates	*Area*	*Delay*	*Power*	*I*	ΔP	ΔD	ΔA	*Gates*	*Area*	*Delay*	*Power*	*I*	ΔP	ΔD	ΔA
`5xp1`	103	139200	28.55	394	5	0.92	0.91	0.97	141	213672	20.21	890	5	0.80	1.00	0.96
`9sym`	163	239424	18.46	617	4	0.98	1.04	1.01	277	405536	10.96	1737	1	0.99	1.00	1.00
`b12`	72	96512	9.98	272	3	0.98	1.00	1.00	95	126092	8.38	422	1	0.94	1.02	0.97
`bw`	134	175392	27.94	407	23	0.77	0.86	1.02	210	296380	18.75	1299	6	0.93	1.04	0.96
`clip`	103	141520	17.77	392	12	0.92	1.02	0.97	162	236756	12.29	1094	6	0.91	1.00	0.95
`cps`	911	1297808	40.05	1899	24	0.98	1.00	1.01	1272	1689424	14.17	2730	6	0.98	1.04	1.00
`inc`	94	127136	28.08	285	21	0.87	1.01	1.10	130	178640	17.95	625	1	1.00	1.01	1.01
`misex1`	52	68208	13.82	176	3	0.92	1.00	1.03	76	108460	9.92	499	1	0.95	1.00	0.97
`misex2`	73	103472	10.24	252	4	0.93	1.00	0.99	133	183164	7.55	551	0	1.00	1.00	1.00
`rd84`	130	177248	19.38	412	6	0.94	1.01	1.02	223	311344	12.80	1180	4	0.96	1.01	0.98
`sao2`	109	151264	21.88	417	4	0.97	1.05	1.02	172	245456	15.85	955	2	0.99	1.04	1.00
Average						0.93	0.99	1.01						0.95	1.01	0.98

Table 2: Circuit Statistics After Learning.

the way logic implications, or *valid clauses*, are found. Their approach uses an analysis tool to identify permissible transformations on the network which may reduce power dissipation. Unlike our approach, their technique for finding permissible transformations is simulation-based; invalid clauses are eliminated from a list using bit-parallel fault simulation and the remaining *potentially valid clauses* are checked for validity using ATPG. As more complex clauses are included, the number of clauses to consider can increase dramatically. On the other hand, the complexity of our learning-based approach does not increase if more complex clauses (i.e., more general implications) are considered while searching for valid transformations.

Although our symbolic algorithm for computing indirect implications will find more general implications, our procedure only exploits the simple, single variable implications. If we compare our results to those reported in [10] we see that we may not always be able to obtain as high a percentage of power savings. This is due in part to the fact that they are exploiting 2-variable implications (e.g., $x = 1 \Rightarrow y \cdot z = 0$) and also because they are including observability conditions when generating their implications. As future work, we would like to take advantage of these more general implications and also include observability conditions on our symbolic algorithm. Both these enhancements should allow us to reduce power dissipation further. In addition, we are working on identifying more powerful transformations that may be applied to the circuit in order to further reduce area and power at no delay cost.

Acknowledgments

We thank Wolfgang Kunz for providing us Hannibal, for supporting its use, and for interesting discussions.

References

[1] C. Y. Tsui, M. Pedram, A. M. Despain, "Technology Decomposition and Mapping Targeting Low Power Dissipation," *Design Automation Conference*, pp. 68-73, Dallas, TX, June 1993.

[2] V. Tiwari, P. Ashar, S. Malik, "Technology Mapping for Low Power," *Design Automation Conference*, pp. 74-79, Dallas, TX, June 1993.

[3] B. Lin, H. de Man, "Low-Power Driven Technology Mapping under Timing Constraints," *Intl. Conf. on Computer Design*, pp. 421-427, Cambridge, MA, Oct. 1993.

[4] R. I. Bahar, G. Hachtel, E. Macii, F. Somenzi, "A Symbolic Method to Reduce Power Consumption of Circuits Containing False Paths", *Intl. Conf. on Computer Aided Design*, pp. 368-371, San Jose, CA, Nov. 1994.

[5] L. A. Entrena, K. T. Cheng, "Sequential Logic Optimization by Redundancy Addition and Removal," *Intl. Conf. on Computer Aided Design*, pp. 310-315, Santa Clara, CA, Nov. 1993.

[6] L. Trevillyan, W. Joyner, L. Berman, "Global Flow Analysis in Automatic Logic Design," *IEEE Transactions on Computers*, Vol. C-35, No. 1, pp. 77-81, Jan. 1986.

[7] J. Jain, R. Mukherjee, M. Fujita, "Advanced Verification Techniques Based on Learning," *Design Automation Conference*, pp. 420-426, San Francisco, CA, June 1995.

[8] W. Kunz, P. R. Menon, "Multi-Level Logic Optimization by Implication Analysis," *Intl. Conf. on Computer Aided Design*, pp. 6-13, San Jose, CA, Nov. 1994.

[9] S. Yang, "Logic Synthesis and Optimization Benchmarks User Guide Version 3.0," *Technical report, Microelectronics Center of North Carolina*, Research Triangle Park, NC, Jan. 1991.

[10] B. Rohfleish, A. Koelbl, B. Wurth, "Reducing Power Dissipation After Technology Mapping by Structural Transformations," *Design Automation Conference*, Las Vegas, NV, June 1996.

[11] E. M. Sentovich, K. J. Singh, C. W. Moon, H. Savoj, R. K. Brayton, A. Sangiovanni-Vincentelli, "Sequential Circuits Design Using Synthesis and Optimization," *Intl. Conf. on Computer Design*, pp. 328-333, Cambridge, MA, Oct. 1992.

[12] A. Ghosh, S. Devadas, K. Keutzer, J. White, "Estimation of Average Switching Activity in Combinational and Sequential Circuits," *Design Automation Conference*, pp. 253-259, Anaheim, CA, June 1992.