Implementation and Use of SPFDs in Optimizing Boolean Networks

Subarnarekha Sinha   Robert K. Brayton

Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley, CA 94720

Abstract

Yamashita et. al.[1] introduced a new category for expressing the flexibility that a node can have in a multi-level network. Originally presented in the context of FPGA synthesis, the paper has wider implications which were discussed in [2]. SPFDs are essentially a set of incompletely specified functions. The increased flexibility that they offer is obtained by allowing both a node to change as well as its immediate fanins. The challenge with SPFDs is (1) to compute them in an efficient way, and (2) to use their increased flexibility in a controlled way to optimize a circuit. In this paper, we provide a complete implementation of SPFDs using BDDs and apply it to the optimization of Boolean networks. Two scenarios are presented, one which trades literals for wires and the other rewires the network by replacing one fanin at a node by a new fanin. Results on benchmark circuits are very favorable.

1 Introduction

At ICCAD'96, an interesting paper by Yamashita et. al. [1] was presented which develops a new way to express flexibility when implementing a node in a multi-level circuit. Classically, don't cares or incompletely specified functions (ISFs) are used to derive the flexibility at a node. These don't cares have been shown to express fully the flexibility of a node due to the non-controllability or non-observability of a node. Satisfiability don't cares (SDCs) express that certain input patterns to a node cannot appear. Observability don't cares (ODCs) express that for certain primary input patterns, the output of a gate is not observable at any of the primary outputs. ODCs plus SDCs are equivalent to ATPG testing techniques used for redundancy removal. Since ODCs are expensive to compute, various subsets have been proposed such as CSPFs (or CODCs) introduced by Muroga. CODCs combined with image computations are implemented in SIS in the command **full_simplify**. CODCs can be computed relatively simply by propagating CODCs backwards through the network. At a multiple fanout point, wire CODCs on the fanouts are intersected to obtain the node CODC. Besides being computationally efficient, CODCs are independent; a CODC at one node can be used without affecting the CODC at another node. ODCs do not have this property. SPFDs are a generalization of CODCs.

To categorize different kinds of flexibility for combinational networks, Sentovich [3, 4] used three categories, ISFs, Boolean relations, and sets of Boolean relations and showed various situations where each occur. SPFDs provide a new useful category. They are essentially sets of ISFs. They can be computed efficiently and are independent, like CODCs, but express more flexibility. They are not a subset of any Boolean relation.

In this paper we define SPFDs in Section 2 and in Section 3, we give an overview on computing SPFDs and introduce some notation. In Section 4, we discuss how to compute them efficiently using BDDs. In Section 5, we provide two different methods for using them to optimize a network. Section 6 gives some experimental results and Section 7 concludes the paper and discusses future developments.

2 SPFDs

Definition 1 *A function f is said to* **distinguish** *a pair of functions g_1 and g_2 if either one of the following two conditions is satisfied:*

$$g_1 \leq f \leq \overline{g}_2 \tag{1}$$
$$g_2 \leq f \leq \overline{g}_1 \tag{2}$$

Note that this definition is symmetrical between g_1 and g_2. We can think of g_1 as the onset and g_2 as the offset in condition 1 or vice-versa for condition 2.

Having to satisfy only one of the two conditions provides the freedom to implement a function or its complement. This is one source of additional flexibility provided by SPFDs.

Definition 2 *An* **SPFD**

$$\{(g_{1a}, g_{1b}), \ldots, (g_{na}, g_{nb})\}$$

represents a Set of Pairs of Functions to be Distinguished.

Definition 3 *A function f* **satisfies** *an SPFD, if f distinguishes each pair of the set, i.e.*

$$[((g_{1a} \leq f \leq \overline{g}_{1b}) + (g_{1b} \leq f \leq \overline{g}_{1a})] \wedge \ldots \wedge$$
$$[(g_{na} \leq f \leq \overline{g}_{nb}) + (g_{nb} \leq f \leq \overline{g}_{na})]$$

An SPFD represents flexibility that can be used to implement a node in a network - the only condition required is that the function implemented at the node satisfy its node SPFD.

A trivial case is where the set is a single pair. In this case the SPFD represents two incompletely specified functions (ISF) where one is the complement of the other. If each of the $\{(g_{1a}, g_{1b}), (g_{2a}, .), \ldots, (., g_{nb})\}$ are pairwise disjoint, then the SPFD represents 2^n ISFs[1].

It is instructive to consider how to represent a given ISF. Think of each minterm m_i in its onset as a function and each minterm m_j in its offset as a function. Then each (m_i, m_j) is a pair of functions to be distinguished, i.e. we have to find a function f such that $f(m_i) \neq f(m_j)$. With ISFs it is necessary

[1]Note that an SPFD cannot represent a single function, it always represents at least a pair. Thus it cannot represent the function 1.

that for all m_i in the onset, we have $f(m_i) = 1$ and for all m_j in the offset, we have $f(m_j) = 0$. With SPFDs this is not imposed, just that f distinguish all the pairs in the list.

This leads to the representation that we use in this paper, of an SPFD as a symmetric bipartite graph i.e. a symmetric relation $R(x, x')$ (which can be represented compactly as a BDD) where $R(x_i, x'_j) = 1 \Leftrightarrow R(x_j, x'_i) = 1$. The variable x is in some space X. If a pair of minterms $(m_i, m_j) \in R$ (i.e. is an edge in the bipartite graph), then we seek a function f such that $f(m_i) \neq f(m_j)$.

Figure 1 illustrates the bipartite graph representation of an SPFD. This shows that 000 has to be distinguished from 010 and 011 and 001 has to be distinguished from 100, but no requirements are imposed on the remaining pairs of minterms. Hence flexibility in optimizing f is provided. If the bipartite

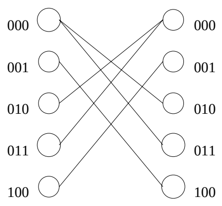

Figure 1: *Bipartite graph representation of* $\{(000, 010), (000, 011), (001, 100)\}$

graph R has a single strongly connected component (SCC) it is a pair of ISFs. If it has k SCCs, it is a set of 2^k ISFs. In the above example, for instance, there are two SCCs which represents 4 ISFs.

Classically, in computing the flexibility at a node in a Boolean network, don't cares are computed which represent a single ISF. These computations can be generalized so that SPFDs are obtained, which provide much more freedom in optimizing the node.

3 Overview

3.1 Notation

We consider a Boolean network with primary inputs $x = (x_1, \ldots, x_n)$, internal variables $y = (y_1, \ldots, y_m)$ and internal nodes $(\eta_1, \ldots, \eta_m)$. Each internal node, η_i, is associated with a function, f_i, a Boolean function of the variables x, y. The relation $y_i = f_i$ is imposed. Since the Boolean network is acyclic, each f_i can be expressed as a global function, $g_i(x)$, of the primary inputs only, by recursively substituting $y_j = f_j$ until only primary inputs remain. In the sequel, we use $g_i(x)$ to denote f_i expressed as a function of x.

The space of local input variables for f_i is denoted Y_i which may consist of both x and y variables. For example, if $f_i = y_3\overline{y_5}+x_6$, then $Y_i = \{y_3, y_5, x_6\}$. The functions in the transitive fanin of f_i define a mapping from X to Y_i, denoted by $G_i(x) : B^n \to B^k$ where $k = |Y_i|$.

The fanout space of node η_i, denoted Z_i, is the union of the input spaces of the fanouts of i, i.e.

$$Z_i = \cup_{j \in \text{ fanout}(i)} Y_j$$

From now on, in this paper, we think of an SPFD as a symmetric bipartite graph where the vertices are minterms in some space (in the above, this is the space Y_i). This is represented by a relation, R, which is just the set of edges of the graph. In the implementation of our algorithms, each R is represented by the BDD of its characteristic function.

3.2 Computing SPFDs

SPFDs can be computed for an entire network by starting at the primary outputs. The computation is a two-step process done in reverse topological order.

1. At each node, the edges of its SPFD are distributed to its input wires, creating wire SPFDs.

2. Once all the fanout wire SPFDs of a node are available, the node SPFD is computed as the union of the wire SPFDs.

To start, one can assume that at each primary output, a compatible don't care set is given. This translates easily into a corresponding SPFD for each output node. Suppose o is a primary output and its SPFD is given as $R_o(X, X')$ i.e. as a bipartite graph in terms of the primary inputs[2]. We first map this into a SPFD $R_o(Y_o, Y'_o)$ in terms of o's input space. Next each pair of minterms to be distinguished (edge) is assigned to one of the inputs wires. The set of pairs assigned to input wire (j, o) is denoted $R_{jo}(Y_o, Y'_o)$. At an internal node η_j assume we have already computed its SPFD, $R_j(Y_j, Y'_j)$. Then its edges are distributed to each input wire (k, j) to obtain $R_{kj}(Y_j, Y'_j)$.

Once all of the fanouts of node η_i have been processed, we have $R_{ij}(Y_j, Y'_j)$ for each wire (i, j), $j \in$ fanout(i). Since the function at η_i must satisfy all the $R_{ij}(Y_j, Y'_j)$ we must have

$$R_i(Z_i, Z'_i) = \cup_{j \in \text{ fanout}(i)} R_{ij}(Y_j, Y'_j)$$

The next step is to translate this into $R_i(Y_i, Y'_i)$ using the mappings,

$$G_j(x), j \in \text{ fanout}(i) \text{ and } G_i(x)$$

Repeating this procedure in reverse topological order from outputs to inputs we obtain SPFDs for all nodes in the network.

3.3 Re-encoding the fanins of a function

Consider a node η_i and an associated SPFD, $R_i(Y_i, Y'_i)$. Then, f_i can be any function which distinguishes all $(y^k, y^l) \in R_i$. Each minterm $y^k \in Y_i$ represents a function of the primary input variables x, i.e. $q_{y^k}(x) = \{x | y^k = G_i(x)\}$. So $R_i(Y_i, Y'_i)$ induces a corresponding SPFD $R_i(X, X')$, i.e.

$$R_i(X, X') = \\ \{(x^j, x^p) | x^j \in q_{y^k}, x^p \in q_{y^l}, (y^k, y^l) \in R_i(Y_i, Y'_i)\}$$

We note that G_i represents the current implementation of the transitive fanin of η_i. But we will be interested in changing G_i (using the computed SPFDs) to improve the circuit. Any new implementation whose mapping $\hat{G}_i$ satisfies

$$\hat{G}_i(x^k) \neq \hat{G}_i(x^l), \forall (x^k, x^l) \in R_i(X, X')$$

[2]Usually an output don't care is given in terms of the primary inputs.

is allowed. If the change from G_i to $\hat{G}_i$ is made, f_i must change to reflect this re-encoding. Figure 2 illustrates the change needed in f_i where conceptually an encoding function $Y_i = E(\hat{Y}_i)$ is inserted in front of f_i. The new function $\hat{f}_i(\hat{Y}_i) = f_i(E(\hat{Y}_i))$ then replaces f_i.

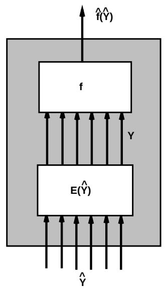

Figure 2: *Function $\hat{f}$ is the original function f under the new encoding of the inputs $E(\hat{Y})$*

We also note that in order for a function at η_i to distinguish y^k from y^l it is only necessary that one of its inputs has different values on these two minterms. Thus, as done in the previous section, different pairs of minterms can be assigned to different wires each of which is given the task of distinguishing the pairs assigned to it. The set of edges assigned to the input wires then constitutes a new SPFD for that wire. This may require that the function at η_i be changed. Thus more flexibility is achieved, by allowing both the function and its inputs to be changed. Note also that the SPFDs generated on the wires are independent; each wire has an assigned task, and it does not matter what any other wire does as long as each satisfies its own task.

3.4 Controlling change

Note that in using the flexibility provided by SPFDs both the function f_i as well as its transitive fanin may be changed, creating new functions $\hat{f}_i$ and a new fanin mapping, $\hat{G}_i(x)$. However, since it would probably be too expensive to change the entire transitive fanin of f_i, we restrict the change to only the immediate fanin of f_i as follows.

We assign a minterm pair $(y^a, y^b) \in R_i(Y_i, Y_i')$ to a fanin η_j only if

$$g_j(x^r) \neq g_j(x^s) \forall x^r \in q_{y^a}(x), x^s \in q_{y^b}(x)$$

In other words, g_j must already have the power to distinguish the two minterms. This insures that only the immediate fanin may need to change to get the new encoding in the Y_i space.

Another concern is that if a node η_i is changed then it may be necessary to propagate this change through all of its transitive fanouts. In using SPFDs with LUT FPGAs, this is not really a concern, since in all the tables in the transitive fanout of an LUT, even though the function may change, the fanins to each LUT remains the same. For general Boolean networks, in the implementation that we have done so far, we have confined the propagated changes to just two nodes, η_i and one of its fanins. This is done for each η_i. This can be done using the CODC of i computed in the normal way instead of its SPFD. This is translated into an SPFD at that node, which is distributed to its inputs. One input, say η_k, is chosen to be changed which may imply that the function at η_i must be changed, but because the CODC at the output of η_i was used, no other node in its transitive fanout needs to be changed. In addition, if η_k is changed, this may require other functions in the transitive fanout of η_k to change. We again block this by requiring that the new function at η_k satisfy the CODCs on the fanout wires of η_k (other than the wire to η_i). Thus in effect, we select a region to be changed[3] and surround it with a frontier of CODCs to block propagation of any changes beyond this region. Of course these restrictions limits the optimizations that can be done, and in the future, we may experiment with allowing the propagated changes to include a larger region.

4 Computing the SPFDs of a network

In the following, we give the resulting computations without proofs, although the reader may be persuaded of their correctness by the fact that for the benchmark circuits of Section 6, we verified the equivalence between the original and the optimized circuits.

4.1 Computing the SPFDs at the fanin wires

To calculate the SPFD at the fanins of a node, we assume that the SPFD, $R_j(Y_j, Y_j')$, has been computed[4]. Suppose we have selected an ordering on the input connections of η_j. We assign an edge of R_j (a pair to be distinguished) to the first input in the ordering which distinguishes the pair. The result is that the SPFD of each fanin wire (i, j) is the set of minterms that are distinguished by fanin η_i but not by any of the fanin wires earlier in the ordering. Thus, the SPFD of fanin wire (i, j) is

$$R_{ij}(Y_j, Y_j') = \\ R_j(Y_j, Y_j')\{\prod_{k_l \in \text{ fanin}(j), k_l < i} (y_{k_l} = y_{k_l}')\}(y_i \neq y_i')$$

The BDD $R_{ij}(Y_j, Y_j')$ represents the SPFD at input wire η_i of node η_j . Each edge of $R_j(Y_j, Y_j')$ has been assigned to one of $\{R_{ij}(Y_j, Y_j')\}, i \in \text{ fanin}(j)$. The above equation is a straightforward BDD calculation.

This calculation essentially distributes the edges in the bipartite graph $R_j(Y_j, Y_j')$ among the fanins of η_j so that each edge is assigned to the first fanin in the ordering that distinguishes the two vertices of the edge.

The following example illustrates how the edges are distributed. Suppose we are given a node Z which has two fanins a and b and a is before b in the ordering. Let the SPFD of Z be given as $\{(00, 01), (00, 10), (00, 11)\}$. Then, the SPFDs of the wires (a, Z) and (b, Z) are $\{(00, 10), (00, 11)\}$ and $\{(00, 01)\}$ respectively. Note that the edge $(00, 01)$ was not assigned to a because $a = 0$ on both minterms. Figure 3 below shows the SPFDs of the wires (a, Z) and (b, Z) due to the SPFD at Z.

4.2 Forming the SPFD at a node

The SPFD at a node is obtained by the union of all the SPFDs of the fanout wires of the node. For a node η_j with multiple fanout wires, the SPFD can be obtained as follows. Let

$$Z_j = \cup_{i \in \text{ fanout}(j)} Y_i$$

[3]In the current implementation, this is just the two nodes η_i and η_k.

[4]All SPFDs are represented using BDDs.

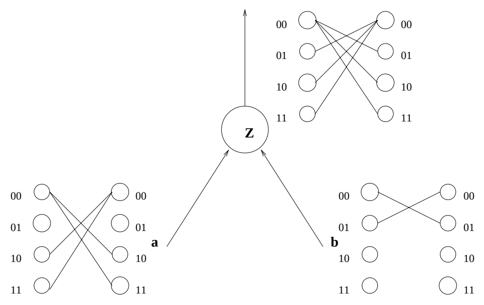

Figure 3: *Computing the SPFDs of (a, Z) and (b, Z) from the SPFD of Z*

be the fanout space and $\mathcal{G}_j(X, Y_j)$ be the relation

$$\mathcal{G}_j(X, Y_j) = \prod_{y_k \in Y_j} (y_k = g_k(x))$$

Let the SPFD associated with the ith fanout wire be $R_{ji}(Y_i, Y_i')$. Then, the SPFD associated with η_j can be obtained by taking the union of the SPFDs of the fanout wires and then taking its image in the Y_j space. Let

$$R_j(Z_j, Z_j') = \cup_{i \in \text{ fanout}(j)} R_{ji}(Y_i, Y_i')$$

Then the SPFD of η_j is given as

$$R_j(Y_j, Y_j') = \exists_{X, X'} \mathcal{G}_j(X', Y_j') R_j(Z_j(X), Z_j'(X')) \mathcal{G}_j(X, Y_j)$$

where $R_j(Z_j(X), Z_j'(X'))$ is obtained from $R_j(Z_j, Z_j)$ by composing each $z_k \in Z_j$ by $g_k(x)$. This technique of computing the SPFD at the output of a node in a single step may be expensive since during this computation, the entire space $\{Y_j \cup Y_j' \cup X \cup X'\}$ has to be considered. Moreover, the intermediate BDD $R_j(Z_j(X), Z_j'(X'))$ may be big.

An alternative approach is to map the SPFD on each fanout wire into the input space, Y_j, and then do the merging. Given the SPFD of the ith fanout wire, $R_{ji}(Y_i, Y_i')$, its image in the local input space, Y_j, can be obtained as follows:

1. Substitute $y_i = g_i(x)$ in $R_{ji}(Y_i, Y_i')$ for each $y_i \in Y_i$ This gives the modified SPFD $R_{ji}(X, Y_i')$.
2. Compute $R_{ji}(Y_j, Y_i') = \exists_X \mathcal{G}_j(X, Y_j) R_{ji}(X, Y_i')$
3. Substitute $y_i' = g_i(x)$ in $R_{ji}(Y_j, Y_i')$ for each $y_i' \in Y_i'$ to obtain the SPFD, $R_{ji}(Y_j, X)$.
4. Compute $R_{ji}(Y_j, Y_j') = \exists_X \mathcal{G}_j(X, Y_j') R_{ji}(Y_j, X)$

This process is repeated for the SPFDs of all the fanout wires of η_j. The SPFD of η_j, is then obtained by

$$R_j(Y_j, Y_j') = \cup_{i \in \text{ fanout}(j)} R_{ji}(Y_j, Y_j')$$

Note that while merging the SPFDs in the local input space, Y_j, the largest space that has be considered at any time is $(Y_j \bigcup Y_j' \bigcup X)$, which is definitely smaller than the largest space encountered by the first method. However, note that each R_{ji} is handled separately.

4.3 Computing the function at a node

As mentioned previously, if the fanin mapping of a node is changed, then the function at the node may have to be changed to reflect this re-encoding. We compute the SPFDs for all the nodes in the network in a reverse topological order from outputs to inputs. Then, we obtain the implementations of the functions at the nodes in topological order from inputs to outputs. So, at any step, when the function at a node is being derived, the new global functions at the input wires are already known $i.e.$ the fanin mapping is known. To compute a set of functions that can be implemented at η_j, the modified SPFD of the node under the new encoding at the inputs is obtained. The new encoding $\hat{Y_j}$ is related to the old encoding Y_j as:

$$E(Y_j, \hat{Y_j}) = \exists_X \mathcal{G}_j(X, Y_j) \hat{\mathcal{G}}_j(X, \hat{Y_j})$$

The modified SPFD is then

$$R_j(\hat{Y_j}, \hat{Y_j'}) = \exists_{Y_j, Y_j'} R_j(Y_j, Y_j') E(Y_j, \hat{Y_j}) E(Y_j', \hat{Y_j'}).$$

Figure 4 illustrates how the new encoding of the inputs changes the original SPFD.

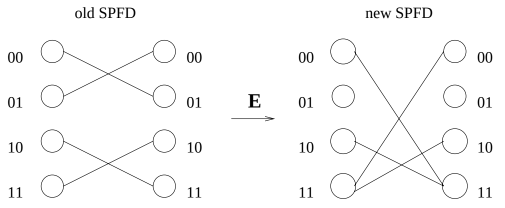

Figure 4: *The modified SPFD under the encoding* $E = \{00 \to 11, 01 \to 10, 10 \to 00, 11 \to 11\}$

Each strongly connected component of $R_j(\hat{Y_j}, \hat{Y_j'})$ encodes the pairs of minterms that have to be distinguished. However, the minterms in one SCC do not have to be distinguished from those in another. This is similar to the graph coloring problem where any two vertices that are connected by an edge cannot be assigned the same color. In the case when $R_j(\hat{Y_j}, \hat{Y_j'})$ is bipartite, we can color all the vertices of the graph using only two colors. So, for each SCC, this corresponds to placing one set of minterms in the onset of the function and the other set in the offset. In this way, a new ISF is obtained at η_j. Thus, if there are k strongly connected components in $R_j(\hat{Y_j}, \hat{Y_j'})$, then there are 2^k functionally different ISFs that can be implemented at η_j. The new implementation at a node is chosen to be the minimum of the minimum covers of all the 2^k ISFs.

Non-bipartition:

There could be situations where $R_j(\hat{Y_j}, \hat{Y_j'})$ is not bipartite, even though $R_j(Y_j, Y_j')$ is. Figure 5. illustrates one such example. In such a situation, the result is a general graph. If we can color the graph using k colors, we can encode the new function using $\log k$ bits. Thus, the original node is replaced by $\log k$ nodes, all of whose fanouts are the same as the original node. This situation is undesirable since the number of fanins of the fanout nodes may increase. We are looking at ways to constrain the SPFD propagation through the network so that under any encoding, the graph $R_j(\hat{Y_j}, \hat{Y_j'})$ remains bipartite.

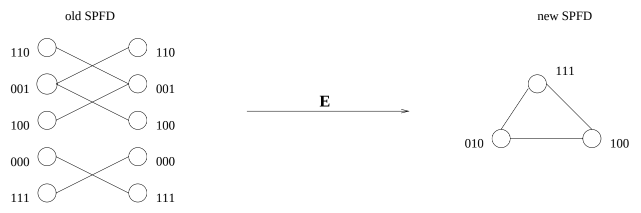

Figure 5: *The modified SPFD under the encoding* $E = \{110 \rightarrow 111, 101 \rightarrow 010, 001 \rightarrow 100, 000 \rightarrow 010, 111 \rightarrow 111\}$

It should be emphasized that in both cases all the minterms in any SCC of the original SPFD are still distinguished. But the new encoding can cause two SCCs of the original SPFD to become incompatible and hence destroy the bipartite structure. This situation can only arise if the SPFD at a node has more than one strongly connected component. Our intial experience is that non-bipartite structures occur rarely. In the experiments described in Section 5, this cannot happen since there is only one SCC at each node.

Finding the SCCs:

Given an SPFD, $R(z, z')$, the individual SCCs can be obtained as follows. Initially, the two step graph $R_2(z, z') = \exists_y R(z, y)R(y, z')$ and the set of left-side nodes $N(z) = \exists_y R(z, y)$ in the bipartite graph are obtained. Then the following steps are performed.

1. Pick $z_0 \in N(z)$.

2. Compute the fixpoint, $E_1(z)$, which is all the nodes that can be reached from z_0 using R_2. Compute $E_0(z) = \exists_y R(y, z)E_1(y)$, the set of nodes that are connected by an edge to a node in $E_1(z)$. Store (E_1, E_0) as an SCC pair.

3. Let $N(z) = N(z)\overline{E_1(z) + E_0(z)}$. If $N \neq \emptyset$, go to 1.

5 Circuit Optimization using SPFDs

5.1 Minimum fanin computation

The objective here is given a node η_j determine if any of the fanin wires can be removed, or if not can any of the fanins be simplified enough to obtain an improved cost function.

In order to simplify a fanin, η_k, as much as possible, we compute the minterms that are distinguished uniquely by that fanin. The minimum SPFD of the wire (k, j), R_{kj}^{min} gives the minterms that need to be distinguished at the output of η_j and can be done only by the wire (k, j). Given the SPFD, $R_j(Y_j, Y_j')$, of η_j,

$$R_{kj}^{min}(Y_j, Y_j') = R_j(Y_j, Y_j')\{\prod_{y_i \in Y_j, i \neq k} (y_i = y_i')\}(y_k \neq y_k')$$

If R_{kj}^{min} is nil, we could remove the wire, but we do not do this until it is determined that the new function at η_j is small enough. If $R_{kj}^{min} \neq \phi$, then a new function is obtained at η_k that satisfies R_{kj}^{min}. We then test if the result also satisfies the CODCs of all the other fanouts of η_k. If so, a new encoding at the inputs of η_j due to the new function at η_k is computed. This is related to the old encoding Y_j as:

$$E(Y_j, \hat{Y}_j) = \exists_X \mathcal{G}_j(X, Y_j)\hat{\mathcal{G}}_j(X, \hat{Y}_j)$$

Note that if $R_{kj}^{min} = \phi$, wire (k, j) can be removed. This is equivalent to setting the term $\hat{y}_k = g_k(x)$ to 1 in the computation of $\hat{\mathcal{G}}_j(X, \hat{Y}_j)$.

The modified SPFD of η_j under the new encoding, $E(Y_j, \hat{Y}_j)$, is

$$R_j(\hat{Y}_j, \hat{Y}_j') = \exists_{Y_j, Y_j'} R_j(Y_j, Y_j')E(Y_j, \hat{Y}_j)E(Y_j', \hat{Y}_j')$$

which is minimized as discussed in the previous section. The *gain* of these changes is computed as

$$\begin{aligned} gain_{kj} &= l_n^{old} + l_f^{old} \Leftrightarrow \{l_n^{new} + l_f^{new}\} \\ &\quad + \alpha * \{w_n^{old} + w_f^{old} \Leftrightarrow \{w_n^{new} + w_f^{new}\}\} \end{aligned}$$

where l stands for the literal count in the factored form, w the number of fanins of the node, the superscripts, old and new, refer to the original and new implementations respectively and the subscripts, n and f, refer to the node and fanin respectively. The parameter α above relates the value of a wire relative to a literal.

Since it is not practical to do this computation for each fanin wire of η_j, in our current experiments, only a subset of the fanins are considered; only those fanins whose own fanins are a subset of the fanins of η_j are selected. This procedure is repeated for each candidate fanin and the one giving the largest non-negative gain, $gain_{kj}$, is chosen. Then the original function at η_j and the corresponding fanin function are replaced by the new ones. This procedure of trying to simplify the fanin of a node as much as possible is called **fanin_simplify**.

5.2 Alternative wire computation

The objective here is to replace wire (k, j) to node η_j with a wire from another node η_s, originally not a fanin of node η_j, such that a new function $\hat{f}_j$ can be found which depends on η_s but not on η_k. The function $\hat{f}_j$ must still satisfy the SPFD at η_j and some gain should be obtained by this replacement.

A wire (s, j) can replace the wire (k, j) if all the minterms in $m_i \in X$ uniquely distinguished by the wire (k, j) are also distinguished by η_s. The procedure for replacing one wire by another is explained below.

Suppose $R_j(Y_j, Y_j')$ has been computed. The minimum SPFD of the wire, (k, j), is computed. Now we seek candidate nodes $\{\eta_s\}$ in the network that can distinguish all the minterms in $R_{kj}^{min}(Y_j, Y_j')$. A necessary and sufficient condition for this is that $H(y_s) = \phi$, where $H(y_s)$ is derived by the following steps.

1. Substitute $y_j = g_j(x)$ in $R_{kj}^{min}(Y_j, Y_j')$ for each $y_j \in Y_j$ to obtain $R_{kj}^{min}(x, Y_j')$.

2. Compute $R_{kj}^{min}(y_s, Y_j') = \{\exists_x (y_s = g_s(x))R_{kj}^{min}(x, Y_j')\}$.

3. Substitute $y_j' = g_j(x)$ in $R_{kj}^{min}(y_s, Y_j')$ for each $y_j' \in Y_j'$ to obtain $R_{kj}^{min}(y_s, x)$.

4. Compute $H(y_s) = \{\exists_x (y_s = g_s(x)) R^{min}_{kj}(y_s, x)\}$.

$H(y_s)$ has the property that if $H(y_s) \neq \emptyset$, then there exists at least one pair of minterms in $R^{min}_{kj}(Y_j, Y'_j)$ that are not distinguished by η_s and hence η_s cannot be a candidate.

Since it is not practical to consider all the nodes in the network, only a subset is considered; only the fanins of η_k and the nodes in their transitive fanout are considered. Of course, nodes in the transitive fanout of η_j cannot be considered.

After the set of candidate nodes is obtained, the following procedure is repeated to obtain the node η_s from which a wire can be added that can replace wire (k, j). First, the modified SPFD of η_j is obtained in the new space

$$\hat{Y_j} = \{y_i \in Y_j, i \neq k\} \cup \{y_s\}$$

A new minimized function at node η_j is then obtained from this modified SPFD as previously described. If the number of literals in the factored form of the new function is less than the number in the factored form of f_j, the replacement is done[5]. In case of a tie in the number of literals, the replacement is also done if the level of η_s is less than the level of η_k. Otherwise, the next node in the candidate set is selected and the same procedure repeated. We call this procedure of replacing wires by other wires, **wire_replace**.

To illustrate the power of **wire_replace**, consider

$$\begin{aligned} z &= \overline{g}b + g\overline{b} \\ g &= \overline{a}b + a\overline{b} \end{aligned}$$

Running **full_simplify** on this example results in no simplification. Now consider **wire_replace**. Wires (a, g) and (b, g) have no alternative wires and hence g remains unchanged. For wire (g, z), the minimum SPFD is $A = \{(00, 10), (01, 11)\}$. (In the set A, each minterm is of the form gb). Now, if we express the minterms of A in terms of the primary inputs, a and b, we get $A' = \{(00, 10), (11, 01)\}$. (The minterms in A' are of the form ab). The primary input, a, can distinguish both pairs in A'. Hence, a is a candidate node and can be used to replace (g, z). Simplifying z we obtain $z = a$. It is also interesting to note that in g, $\{00, 11\}$ are in the offset and $\{01, 10\}$ are in the onset. However for a, $\{00, 01\}$ are in the offset and $\{11, 10\}$ are in the onset. Yet, a can be used to replace g. This ability to mix onset and offset minterms is another source of additional flexibility provided by SPFDs.

6 Results

The results for fanin simplification are shown in Table 1. The initial circuits were obtained from the various benchmarks by running **script.rugged** in SIS on them. The result served as our point of comparison. We then took this output and subjected it to an iteration of **fanin_simplify** until no gain was obtained or the cost function (which was equal to the gain in the number of literals plus twice the gain in the number of wires) became non-positive for the first time. In some circuits, the cost function kept oscillating and for these circuits we took the point at which the cost function first started to oscillate. We recorded, under the heading, **(fanin_simplify)***, both the number of connections in the circuit "wires" and the number of

[5]Of course, we could use the gain function given in Section 5.1 to control acceptance, but the experiment here is to reduce literals (and hopefully wires too).

	script.rugged		*$(fanin\_simplify)$**			
NAMES	wires	literals	wires	ratio	literals	ratio
apex6	650	741	633	0.974	731	0.987
apex7	222	245	210	0.946	241	0.984
b9	115	122	111	0.965	122	1
bbara	51	63	45	0.882	61	0.968
bbsse	140	140	138	0.986	140	1
c1908	378	540	357	0.944	534	0.989
c432	205	205	194	0.946	201	0.98
c499	344	552	296	0.86	554	1.004
c8	128	139	118	0.922	144	1.036
cht	165	165	164	0.994	165	1
cse	213	215	201	0.944	206	0.958
dk16	348	348	307	0.882	345	0.991
dk17	88	89	44	0.5	58	0.652
ex1	279	280	259	0.928	261	0.932
ex2	172	172	158	0.919	173	1.006
ex3	84	86	81	0.964	87	1.012
ex4	91	91	87	0.956	87	0.956
ex5	71	71	47	0.662	66	0.93
ex6	108	109	96	0.889	111	1.018
f51m	60	91	39	0.65	88	0.967
frg1	79	136	60	0.759	137	1.007
frg2	833	886	753	0.904	894	1.009
i6	391	457	391	1	457	1
i7	587	596	517	0.881	583	0.978
i8	1012	1015	989	0.977	999	0.984
i9	587	596	577	0.983	603	1.012
k2	1112	1120	1084	0.975	1100	0.982
kirkman	300	308	87	0.29	135	0.438
lal	89	105	83	0.933	103	0.981
planet	614	617	594	0.967	610	0.989
s1	429	430	356	0.83	375	0.872
sand	612	613	579	0.946	584	0.953
scf	983	985	974	0.991	977	0.992
sct	63	79	55	0.873	78	0.987
sse	140	140	109	0.779	117	0.836
styr	596	596	561	0.941	565	0.948
term1	130	179	104	0.8	174	0.972
too_large	266	347	255	0.959	349	1.006
ttt2	184	219	167	0.908	215	0.982
vda	611	615	606	0.992	612	0.995
x1	285	298	275	0.965	301	1.01
x2	44	48	42	0.955	49	1.021
x3	720	787	675	0.938	764	0.971
x4	367	386	353	0.962	384	0.995
z4ml	29	41	20	0.69	41	1
AVERAGE				0.887		0.962

Table 1: *Results for $fanin\_simplify$*

	script.rugged		*(*$wire\_replace$*)**				*(script.rugged,* $(wire\_replace$*)*)**			
NAMES	wires	literals	wires	ratio	literals	ratio	wires	ratio	literals	ratio
apex6	650	741	625	0.962	724	0.977	621	0.955	715	0.965
apex7	222	245	203	0.914	235	0.959	198	0.892	226	0.922
b9	115	122	111	0.965	119	0.975	112	0.974	122	1
bbara	51	63	46	0.902	60	0.952	49	0.961	61	0.968
bbsse	140	140	119	0.85	126	0.9	102	0.729	110	0.786
c1908	378	540	352	0.931	525	0.972	352	0.931	525	0.972
c432	205	205	186	0.907	222	1.083	186	0.907	222	1.083
c499	344	552	300	0.872	552	1	300	0.872	552	1
c8	128	139	127	0.992	138	0.993	125	0.977	136	0.978
cht	165	165	164	0.994	165	1	163	0.988	164	0.994
cse	213	215	201	0.944	204	0.949	162	0.761	183	0.851
dk16	348	348	316	0.908	321	0.922	187	0.537	245	0.704
dk17	88	89	40	0.455	53	0.596	37	0.42	51	0.573
ex1	279	280	255	0.914	258	0.921	219	0.785	229	0.818
ex2	172	172	142	0.826	160	0.93	134	0.779	151	0.878
ex3	84	86	82	0.976	85	0.988	45	0.536	62	0.721
ex4	91	91	82	0.901	85	0.934	71	0.78	78	0.857
ex5	71	71	60	0.845	67	0.944	26	0.366	51	0.718
ex6	108	109	92	0.852	103	0.945	89	0.824	96	0.881
f51m	60	91	39	0.65	83	0.912	45	0.75	70	0.769
frg1	79	136	44	0.557	127	0.934	51	0.646	127	0.934
frg2	833	886	696	0.836	792	0.894	690	0.828	735	0.83
i6	391	457	391	1	457	1	391	1	457	1
i7	518	584	517	0.998	583	0.998	517	0.998	583	0.998
i8	1012	1015	980	0.968	988*	0.973*				
i9	587	596	584	0.995	596	1	580	0.988	592	0.993
k2	1112	1120	1067	0.96	1082	0.966	*		*	
kirkman	300	308	137	0.457	198	0.643	85	0.283	126	0.409
lal	89	105	82	0.921	101	0.962	79	0.888	102	0.971
planet	614	617	586	0.954	593	0.961	555	0.904	589	0.955
s1	429	430	349	0.814	381	0.886	275	0.641	298	0.693
sand	612	613	566	0.925	574	0.936	521	0.851	550	0.897
scf	983	985	970	0.987	974	0.989	870	0.885	917	0.931
sct	63	79	57	0.905	78	0.987	55	0.873	75	0.949
sse	140	140	119	0.85	126	0.9	102	0.729	110	0.786
styr	596	596	550	0.923	555	0.931	431	0.723	482	0.809
term1	130	179	97	0.746	152	0.849	93	0.715	103	0.575
too_large	266	347	253	0.951	234	0.674	*		*	
ttt2	184	219	160	0.87	206	0.941	122	0.663	163	0.744
vda	611	615	607	0.993	612	0.995	571	0.935	579	0.941
x1	285	298	279	0.979	295	0.99	279	0.979	295	0.99
x2	44	48	43	0.977	48	1	39	0.886	46	0.958
x3	720	787	650	0.903	753	0.957	628	0.872	705	0.896
x4	367	386	347	0.946	381	0.987	332	0.905	367	0.951
z4ml	29	41	28	0.966	38	0.927	28	0.966	38	0.927
AVERAGE				0.888		0.936		0.807		0.871

Table 2: *Results for* $wire\_replace$

literals in the factored forms of the Boolean network. The ratio of these results to the output of **script.rugged** is also shown. At the bottom of the table we compute the average ratios for both the wires and the literals. We see, on average, a 11% reduction in the number of wires as well as a 3% reduction in literals. But since we assigned greater weights to wires ($\alpha = 2$), there is a greater reduction in wire count.

The results for wire replacement are shown in the Table 2. The initial circuits were obtained as in the previous case. We took the output of **script.rugged** and subjected it to an iteration of **wire_replace** until no gain was obtained. The number of wires, the number of literals in the factored form of the network and the ratio of these results to the output of **script.rugged** were stored under the heading **(wire_replace)***. For $i8$, the program ran out of memory before the iterations could converge, so we took the values of the previous iteration. The third set of columns was obtained by taking the result of **(wire_replace)*** and repeating **script.rugged** followed by **(wire_replace)*** until no gain was recorded. For $k2$ and too_large, the program ran out of memory even before the first iteration was over. At the bottom of the table we compute the average ratios for both experiments and for both wires and for literals. We see on average a 11% reduction in wires and 6% in literals after **(wire_replace)***, and still better results for the repetition of **script.rugged** and **(wire_replace)***, a 19% reduction in wires and 12% in literals.

The experiments were made with automation in mind. We wanted to devise an automatic script that could be run on any module without having to interact with it. The results are the "scripts", **(script.rugged(fanin_simplify)*)** and **(script.rugged(wire_replace)*)***.

In **(script.rugged (fanin_simplify)*)**, it is possible to trade off wires for literals or vice-versa by setting different costs to the literals and wires. In **(script.rugged (wire_replace)*)***, we could set conditions for choosing an alternative wire for a given wire. In addition to minimizing literals, we could use this to optimize different criteria like delay, maximum wire length, etc.

Since the computation at a node can take an arbitrarily long time, we have introduced timeouts to prevent the computation from hanging up. So, with every node in the network we associate a timeout interval and if the time of computation exceeds the interval, we just quit that node and move on to the next. Right now, we set a timeout interval of 20 sec., more for ruggedizing the algorithms than a concern for the run time. We are looking at a number of techniques to improve the run times.

In the future, we see this technology as important for improving delay and wireability for DSM designs.

7 Conclusions

We have implemented the computation of SPFDs using BDDs and have shown that their use is quite feasible in circuits of medium size. Roughly, any circuits where **script.rugged** can be used, SPFDs can also be used. Ultimately, both **fanin_simplify** and **wire_replace** can be made more rugged following suggestions by Savoj [5], by filtering out known problems and by better controlling time-outs on some of the BDD calculations. The initial experiments with **fanin_simplify** and **wire_replace** are very encouraging. In these experiments, we have strongly restricted the SPFDs computed, in order to control the change in the network. We want to experiment with relaxing these conditions while still controlling the gains made in a predictable way.

Ultimately, we plan to use SPFDs in DSM synthesis where the ability to assign costs to the wires during circuit optimization or to replace one wire by another should be very useful to alleviate wiring problems or to improve wire delay and noise problems.

Acknowledgements

This research was supported by the SRC under contract DC-324, and the California MICRO program with participating industrial sponsors, Cadence, Motorola, and Fujitsu.

References

[1] S. Yamashita, H. Sawada, and A. Nagoya. A New Method to Express Functional Permissibilities for LUT based FPGAs and Its Applications. In *International Conference on Computer Aided Design*, pages 254–261, November 1996.

[2] Robert K. Brayton. Understanding SPFDs: A New Method for Specifying Flexibility. In *IWLS*, 1997.

[3] E. M. Sentovich and R. K. Brayton. Multiple Boolean Relations. In *International Workshop on Logic Synthesis*, Tahoe City, CA, May 1993.

[4] Ellen M. Sentovich, Vigyan Singhal, and Robert K. Brayton. Boolean Function Sets. In *unpublished manuscript*, 1996.

[5] H. Savoj. Improvements in Technology Independent Optimization of Logic Circuits. In *IWLS*, 1997.