COMBINATIONAL LOGIC SYNTHESIS FOR DIVERSITY IN DUPLEX SYSTEMS

Subhasish Mitra and Edward J. McCluskey
Center for Reliable Computing
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California
http://crc.stanford.edu

Abstract

We describe logic synthesis techniques for designing diverse implementations of combinational logic circuits in order to maximize the data integrity of diverse duplex systems in the presence of common-mode failures. Data integrity means that the system either produces correct outputs or indicates errors when incorrect outputs are produced. Design diversity has long been used to increase the data integrity of duplex systems against common-mode failures. The conventional notion of diversity is qualitative and relies on "independent" generation of "different" implementations. In a recent paper, we presented a metric to quantify diversity among several designs. Our synthesis techniques described in this paper use the diversity metric as a cost function and maximize diversity while reducing the area overhead of the resulting diverse duplex system.

1. Introduction

Concurrent Error Detection (CED) techniques are widely used for designing systems with high data integrity. A duplex system is an example of a classical redundancy scheme that has been used in the past for concurrent error detection. There are many examples of commercial dependable systems from companies like Stratus and Sequoia using hardware duplication [Kraft 81, Pradhan 96]. Hardware duplication is also used in the IBM G5 processor [Webb 97, Spainhower 99]. Figure 1.1 shows the basic structure of a duplex system.

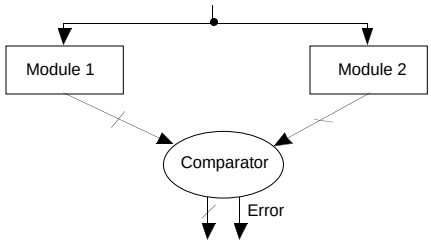

Figure 1.1. A Duplex System

In a duplex system there are two modules (shown in Fig. 1.1 as Module 1 and Module 2) that implement the same logic function. The two implementations can be the same or different. A comparator is used to check whether the outputs from the two modules agree. If the outputs disagree, the system indicates the presence of an error. *Data Integrity* means that the system either produces correct outputs or generates error signal when incorrect outputs are produced. For a duplex system, data integrity is maintained as long as both modules do not produce identical erroneous outputs (assuming that the comparator is fault-free).

In any redundant system, *common-mode failures* (CMFs) result from failures that affect more than one module at the same time, generally due to a single cause [Lala 94]. These include operational failures that may be due to external (such as EMI, power-supply disturbances and radiation) or internal causes. Common-mode failures in redundant VLSI systems are surveyed in [Mitra 00a]. *Design diversity* has been proposed in the past to protect redundant systems against common-mode failures. In [Avizienis 84], *design diversity* was defined as the "independent" generation of two or more software or hardware elements (e.g., program modules, VLSI circuit masks, etc.) to satisfy a given requirement. Design diversity has been applied to both software and hardware systems [Avizienis 77, Lyu 91, Briere 93, Riter 95]. Tohma proposed using the implementations of logic functions in true and complemented forms during duplication [Tohma 71]. The use of a particular circuit and its dual was proposed in [Tamir 84] to achieve diversity in order to handle common-mode failures. The basic idea is that, with different implementations, common failure modes will probably cause different error effects.

In an earlier paper [Mitra 99a], we developed a metric to quantify diversity among several designs and used this metric to perform reliability analysis of redundant systems. In this paper, we develop techniques to synthesize diverse combinational logic circuits using the diversity metric as a cost function during the synthesis process. Thus, in contrast to "independent" generation of "different" implementations, we guarantee maximum data integrity if the diverse implementations synthesized by our procedure are used in duplex systems.

The problem studied in this paper is as follows: Suppose that we are given the truth table of a combinational circuit to be implemented and one implementation of the combinational circuit. We call the given combinational circuit implementation N_1. Our goal is to synthesize another implementation (N_2) of the same combinational circuit such that the diversity of N_2 with respect to N_1 is maximized.

The faults considered are single-stuck-at faults in the two implementations. Assume that we are given two implementations (logic networks) of a logic function, an input probability distribution and faults f_i and f_j that occur in the first and the second implementations, respectively.

The *diversity* $d_{i,j}$ *with respect to the fault pair* (f_i, f_j) is the conditional probability that the two implementations do not produce identical errors, given that faults f_i and f_j have occurred [Mitra 99a]. In this paper we assume that all input combinations are equally probable.

For every fault f_i in N_1, we find the fault f_j in N_2 such that the value of $d_{i,j}$ with respect to the fault pair (f_i, f_j) is the minimum over all f_j's. The fault pair (f_i, f_j) is defined as a *worst-case fault pair*. Finally, we calculate the expectation of the $d_{i,j}$'s of the worst-case fault pairs assuming the worst-case fault pairs are equally probable to obtain the *expected diversity of* N_2 *with respect to* N_1.

In the problem statement, we assume that we are given one implementation N_1. This assumption is reasonable in a scenario where the user wants to use a particular implementation for area, performance or power consumption reasons. However, our techniques can still be applied in the absence of the above assumption.

While the main focus of this paper is on combinational logic circuits, the ideas presented can be extended to sequential circuits. Given the specification of a sequential logic circuit and an encoding of the internal states, the problem of synthesizing the sequential circuit can be mapped to a combinational logic synthesis problem. However, if we have the freedom to choose internal state encoding, then diversity can be created by encoding the internal states in different ways. This problem is outside the scope of this paper.

In Sec. 2, we provide a motivation for this work. Section 3 introduces some basic concepts about stuck-at faults in logic networks. We present the problem formulation in Sec. 4. Sections 5 and 6 describe two-level and multi-level logic synthesis techniques for designing diverse implementations. We conclude in Sec. 7.

2. Motivation

As observed in Sec. 1, while the concept of design diversity was well-known since the 1980's, no systematic technique was developed to guarantee sufficient diversity to maximize reliability of redundant systems with diversity. This is mainly due to the fact that the concept of diversity was qualitative. In [Mitra 99a, Mitra 99b] we developed a metric to quantify diversity and presented simulation results to demonstrate the effectiveness of using diverse implementations in protecting duplex systems against common-mode failures for some MCNC benchmark circuits. Moreover, in [Mitra 00b] we compared various concurrent error detection (CED) techniques (based on identical duplication, diverse duplication, parity prediction, etc.); and concluded that while the area overhead of diverse duplex systems is marginally more than that of parity prediction, diverse duplication provides better data integrity against multiple and common-mode failures compared to parity prediction. However, for all our simulation experiments, we didn't use any systematic approach to synthesize diverse duplex designs. We just synthesized truth tables of combinational logic circuits with true and complemented outputs using *Sis* [Sentovich 92]. This paper presents a systematic technique to synthesize diverse implementations of the same logic function so that system data integrity against common-mode failures (and worst-case multiple failures) is maximized.

3. Stuck-at Faults: Equivalence and Dominance

Research in the area of digital testing and diagnosis of combinational and sequential logic circuits has demonstrated the effectiveness of the logical *stuck-at* fault model [McCluskey 00]. In this model, the failures in a logic circuit behave as if as some lines in the circuit assume constant logical values, either 1 or 0, independent of the logic values on other lines of the circuit.

For the rest of this paper, we assume that all failures manifest themselves as single stuck-at faults in the circuit. For example, consider the network shown in Fig. 3.1. The function implemented by the network is $wx + y$. Consider a stuck-at-0 (s-a-0) fault on the line y, denoted by $y/0$. The function implemented by the network, in the presence of the fault, is wx. Thus, wxy = 101, when applied to the input of the logic circuit causes the faulty network to produce a 0 and the fault-free network to produce a 1. Therefore, the fault $y/0$ is detected by the pattern wxy = 101.

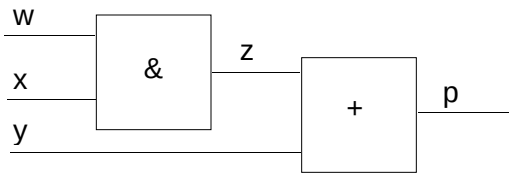

Figure 3.1. An example logic circuit

A fault is said to be *equivalent* to another fault if and only if the output function realized by the network with only the first fault present is equal to the function realized when only the second fault is present. For example, in the network of Fig. 3.1, in the presence of the fault $x/0$, the function implemented is y. In the presence of the fault $z/0$, the function implemented by the network is also y. Hence, the faults $x/0$ and $z/0$ are equivalent. If a fault f is equivalent to fault g, the set of test patterns that detect fault f is the same as the set of test patterns that detect g. A fault f *dominates* fault g if and only if all input combinations that detect g also detect f (i.e., the set of test patterns that detect g is a subset of the set of test patterns that detect f). In our example, the set of input combinations that detect $z/0$ is {wxy = 110}. The set of input combinations that detect $p/0$ is {wxy = 110, 111, 001, 011, 101}. Hence, fault $p/0$ dominates fault $z/0$. Techniques for obtaining equivalence and dominance relationships among various fault pairs have been described in [McCluskey 71] and [To 73].

4. Problem Formulation

In this section, we formulate the problem studied in this paper. We assume that we are given the truth table of a combinational circuit to be implemented and one implementation of the combinational circuit. We call the

given combinational circuit implementation N_1. Our goal is to synthesize another implementation (N_2) of the same combinational circuit so that the diversity of N_2 with respect to N_1 is maximized and the logic area (estimated by calculating the number of logic gates and the literal count) is minimized.

Conventional logic synthesis techniques consist of two steps: (i) two-level minimization [McCluskey 56, Brayton 84] followed by (ii) multi-level transformations [Rajski 92, De Micheli 94]. In this paper, we also follow the same flow for synthesizing implementation N_2.

5. Two-Level Synthesis

In this section, we describe two-level logic synthesis techniques for synthesizing the combinational logic network N_2 as stated in Sec. 4. We restrict ourselves to two-level logic circuits in AND-OR form. However, the entire discussion also holds for logic circuits in OR-AND form.

Theorem 1: For single-output functions, for any fault f in N_1, any two-level logic implementation of N_2 produces the same worst-case fault pair.

Proof: Any internal single-stuck-at fault (that does not affect the primary inputs or the output) in a two-level single-output logic circuit is dominated by or is equivalent to the output stuck-at 0 or the output stuck-at 1 fault. This is because, any single-stuck-at fault at the input of an AND (OR) gate is dominated by or is equivalent to a stuck-at fault at the output of the same AND (OR) gate. The dominating fault has a bigger detection set than a dominated fault. Hence, for any stuck-at fault f in the first implementation N_1, the dominating single-stuck-at fault in the second implementation has more chance of producing identical erroneous outputs compared to the dominated fault; thus, the fault in the second implementation that produces the worst-case pair with f affects the primary inputs or the output. The set of test patterns for the output or input stuck-at faults is independent of the implementation and only depends on the truth table of the function. Hence, any implementation of N_2 will produce the same set of worst-case fault pairs. Q.E.D.

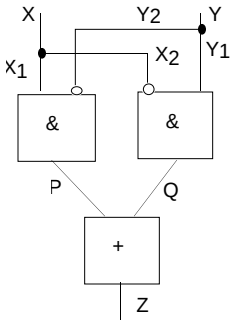

Figure 5.1. An example to illustrate Theorem 1

Theorem 1 can be explained using the example in Fig. 5.1. Consider the single-stuck-at fault X_1/1, for example. The fault X_1/1 is dominated by P/1. Again P/1 is equivalent to Z/1. Thus, the set of test patterns that detect Z/1 is a superset of the set of test patterns that detect X_1/1. Following the definition of $d_{1,2}$, the $d_{1,2}$ value of the fault pair (f, X_1/1) is greater than or equal to that of the fault pair (f, Z/1) because the set of test patterns of X_1/1 is a subset of that of Z/1. Hence, the diversity with respect to the fault pair (f, Z/1) is less than or equal to that with respect to (f, X_1/1). Since Z/1 is detected by all input combinations for which the logic function in Fig. 5.1 produces a 0, the set of test patterns for Z/1 is independent of the two-level implementation of the logic function.

It may be noted that Theorem 1 is not true for two-level multi-output logic circuits. For two-level multiple-output logic circuits we can control the amount of sharing of the AND gates among the outputs. Thus, we can effectively control the fanout structure of the implementation N_2 and enhance diversity between the two implementations. This corroborates our earlier observation in [Mitra 99a] that the fanout structure of a logic network plays a primary role in determining its diversity with respect to a given implementation. We explain this fact using the example in Fig. 5.2.

In Fig. 5.2a, we have two identical implementations N_1 and N_2 where the AND gate *abcd* is shared by Z_1 and Z_2. Let us consider a stuck-at-1 fault on signal line p, the output of the AND gate. For a duplex system with identical implementations, the fault in N_2 that creates the worst-case fault pair is also p stuck-at-1. The two implementations produce identical erroneous outputs for any input combination that either produces an error at Z_1 (Z_2) and the fault-free value of $Z_2 = 1$ ($Z_1 = 1$) or produces errors at both Z_1 and Z_2. The set of input combinations of a, b, c and d that produce identical errors is: {*abcd* = 0011, 0111, 0001, 0101, 0010, 0110, 0000, 0100, 1000, 1001, 1010, 1011, 1100, 1101, 1110}.

Next, we consider the duplex system in Fig. 5.2b. In the first implementation, the AND gate *abcd* is shared by Z_1 and Z_2, while in the second implementation, it is shared by Z_2 and Z_3. We consider the same fault (p/1) in N_1 and N_2. Fault p/1 in N_1 always produces $Z_1 = 1$ and $Z_2 = 1$; however, output Z_3 is not affected by the fault. Fault p/1 in N_2 always produces $Z_2 = 1$ and $Z_3 = 1$; however, output Z_1 is not affected by the fault. Hence, The two implementations produce identical errors in response to any input combination that produces an error at Z_2 and the fault-free value of $Z_1 = 1$ and $Z_3 = 1$. It is clear from Fig. 5.2b that there is no input combination other than *abcd* = 1111 that produces fault-free value of $Z_1 = 1$ and $Z_3 = 1$. However, with *abcd* = 1111, the fault is not excited and no error is produced at Z_2. Thus, because of diversity in the fanout structure, the fault effect propagates to different set of outputs for the example in Fig. 5.2b.

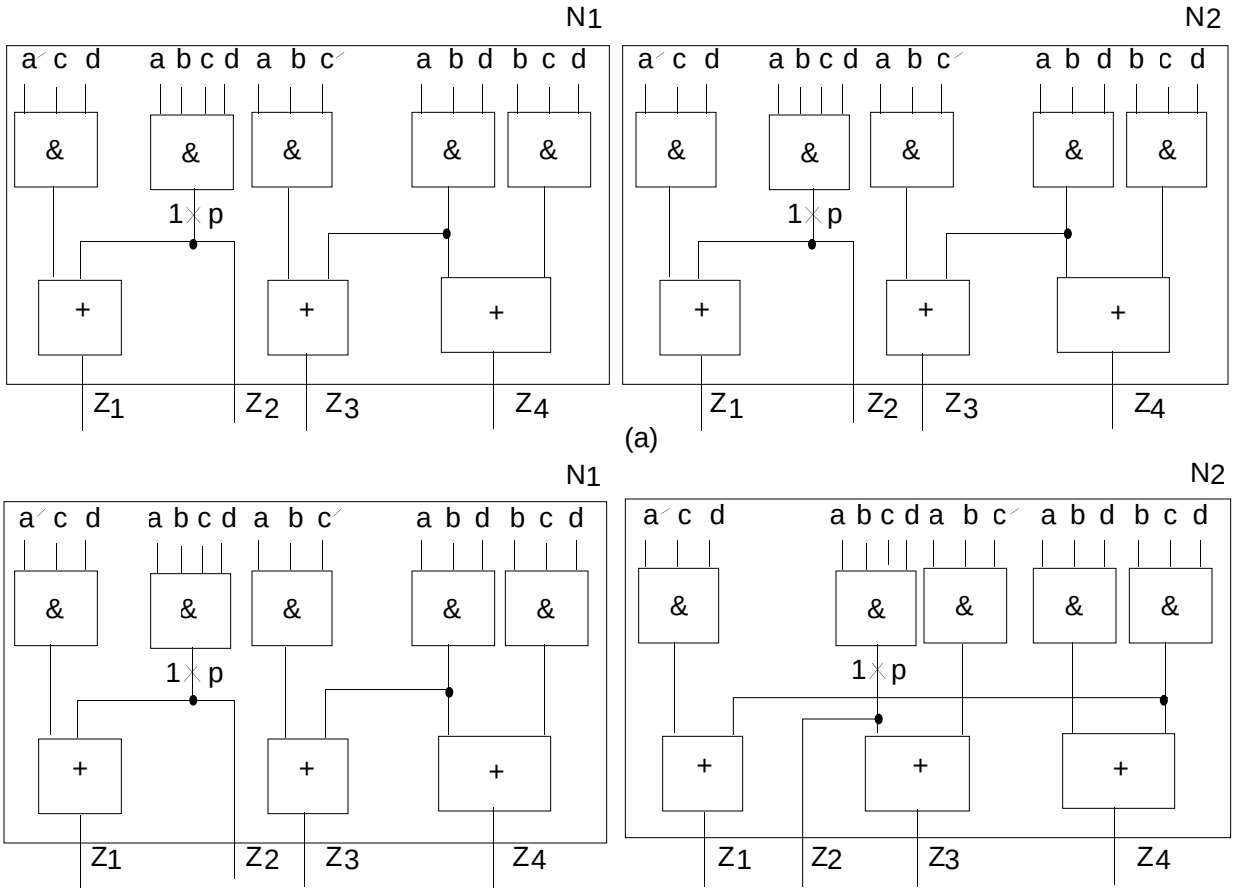

Figure 5.2. Example to illustrate the effect of fanout structure on diversity

Hence, there is no input combination for which the two implementations produce identical errors in the presence of these two faults. Thus, diversity in the fanout structure of the two implementations can be utilized for enhancing the data integrity of the duplex system against common-mode failures.

Classical algorithms for two-level logic synthesis can be used for creating diversity in the fanout structures of the two implementations in a duplex system. The classical Quine-McCluskey method for two-level minimization of single-output logic circuits [McCluskey 56] has been extended for multiple-output logic circuits [McCluskey 86]. The basic procedure consists of the following steps:

1. *Multiple-output Prime Implicant Generation*: Each minterm (fundamental product or cube) has a tag associated with it. For a function with *n* outputs, the tag has *n* entries. The *i*th entry is a 1 if the minterm is in the 1-set of the output function Z_i, otherwise the entry is 0. An input combination belongs to the 1-set (0-set) of a function if the function evaluates to 1 (0) for that input combination. A minterm can be looked upon as an implicant. Two implicants (with '-'s) that differ in a single bit position can be combined to form a new implicant (having a *don't care* ,'-', entry in the position in which the two implicants differ). The tag of the resulting implicant is obtained by bit-wise AND-ing the tags of the two implicants. Moreover, an implicant can be canceled if it can be combined with another implicant to form a new implicant and the tag of the resulting implicant is the same as the tag of the implicant under consideration. The procedure of combining implicants is repeated until no more new implicants can be formed. The implicants (excluding the canceled ones) thus obtained are called *multiple-output-prime-implicants* (MOPIs).

Suppose that the minterm 1111 is in the 1-sets of output functions Z_1, Z_2 and Z_3, 1101 is in the 1-set of output function Z_1, 1100 is in the 1-sets of output functions Z_1 and Z_3, and 1110 is in the 1-set of Z_1. Assuming there are only 3 outputs, the tag of 1111 is {1, 1, 1}, the tag of 1101 is {1, 0, 0} and the tags of 1100 and 1110 are {1, 0, 1} and {1, 0, 0}, respectively. We can find that 1111 and 1101 can be combined to form the implicant 11-1 with tag equal to {1, 0, 0}. Since the tag of 11-1 is the same as that of 1101, we can cancel 1101. Similarly, we can combine 1100 and 1110 to form 11-0 with tag equal to {1, 0, 0}. Finally, we can combine 11-1 and 11-0 to form 11-- with tag equal to {1, 0, 0}. Since the tag of

11-- is the same as that of 11-0 and 11-1, we can cancel 11-0 and 11-1.

2. *MOPI Covering Table*: Once all the MOPIs are found, the process of selecting the MOPIs to form multiple-output minimal sums is carried out by means of a MOPI covering table called the *multiple-output prime implicant table* [McCluskey 86]. Each row of the table corresponds to a multiple-output prime implicant. There is a column for each fundamental product (minterm) of each of the output functions. All fundamental products corresponding to the same output function are represented by a set of adjacent columns. If a fundamental product occurs in more than one of the functions, it is represented by more than one column in the table. There is an X in a particular table entry if the MOPI corresponding to the row covers the minterm corresponding to the column, and the MOPI has a 1 in its tag set corresponding to output function that the column (minterm) belongs to. Example 5.1 (taken from [McCluskey 86]) illustrates the formation of a MOPI covering table.

3. *MOPI Covering*: For MOPI covering, we have to select enough rows of the covering table such that there is at least one X in each column. However, we have to minimize the number of gates and number of literals. The corresponding cost function computation is described in [McCluskey 86].

Example 5.1: Consider two functions $Z_1(a, b, c, d) = \sum(0, 2, 5, 6, 13)$ and $Z_2(a, b, c, d) = \sum (0, 8, 12)$. The multiple-output prime implicants together with their tag sets are:

A = a′b′d′, {1, 0}; B = bc′d, {1, 0}; C = a′cd′, {1, 0}; D = ac′d′, {0, 1}; E = b′c′d′, {0, 1}; F = a′b′c′d′, {1, 1}. Table 5.1 shows the MOPI covering table.

Table 5.1. MOPI Covering Table

	Z_1					Z_2		
	0	2	5	6	13	0	8	12
A	X	X						
B			X		X			
C		X		X				
D							X	X
E						X	X	
F	X					X		

A simple high-level pseudo-code for the entire process of synthesizing the best implementation of N_2 given N_1 and maximizing the diversity with respect to the worst-case fault pairs and minimizing the area overhead is shown next. Note that, we implement the function with complemented outputs for N_2 and then add inverters at the outputs of the implementation. This is mainly because the cost function evaluation becomes computationally very intensive otherwise. Moreover, there are some potential gains because there is no apparent correlation between the MOPIs of a given function and the MOPIs of the same function with complemented outputs. Appendix 1 shows the cost function evaluation step for the case when we implement the given truth table without complemented outputs for N_2 and explains why it is computationally intensive.

The MOPI generation step is the same as in conventional synthesis. Finding a MOPI cover is also very similar to the covering step in classical two-level minimization. The above pseudo-code performs an exhaustive search to obtain the best cover for implementing N_2. The covering problem has exponential complexity in the worst-case and may be impractical for large circuits. However, heuristic algorithms for covering [Cormen 89] and branch-and-bound techniques [McCluskey 86] can be used to speedup covering.

```
Procedure: Generate "Best" N2
Input: Function Truth Table, Implementation N1
Output: Implementation of N2

Procedure:
1.  Generate a new truth table by complementing the outputs of the given truth table
2.  Generate MOPIs for the new truth table
3.  Construct MOPI covering table
4.  Min. # Gates = Infinite, Min. # Literals = Infinite, Max. Diversity = 0
5.  While (all covers not examined)
6.          Generate next cover C
7.          Calculate cost of C:
            # Gates = G, # Literals = L
            Expected diversity with respect to N1 = E
8.          If Cost of C "better" than Min. cost
        /* e.g., G < Min. # Gates, L < Min. # Literals, E > Max. Diversity */
9.                  Min. # Gates = G
10.                 Min. # Literals = L
11.                 Max. Diversity = E
12.                 Best_Cover = C
13.         Endif
14. Endwhile
15. Return Best_Cover with inverters at the outputs for implementing N2
```

It is clear from the pseudo-code that the cost function has two major components: (i) the area of the circuit (estimated by the number of gates and the number of literals) and (ii) the diversity component. The weights associated with these two components in the final cost function are flexible and depend on the application. For area constrained designs, it may be required to obtain the minimal area implementation and diversity can be a secondary component of the cost function. For some applications, there may be a bound on the maximum area that can be used for implementing N_2 and the diversity component may be maximized for all designs with the area cost less than the predetermined bound.

In classical two-level minimization of multiple-output logic circuits, we can reduce the size of the covering table by establishing dominance relationships among the rows and columns of the table. Column C_1 dominates C_2 if the prime implicants that cover the minterm corresponding to C_2 is a subset of the set of prime implicants that cover the minterm corresponding to C_1. If C_1 and C_2 correspond to

minterms of the same logic function, then the dominating column C_1 can be removed. This rule can be used for our current synthesis technique too. However, unlike the conventional synthesis procedure, we cannot remove equivalent rows from the table. If area (gate count and literal count) minimization is our primary goal, then we can remove dominated rows with higher area costs from the table.

For evaluating the cost of a MOPI cover, we have to do some extra processing in order to find the worst-case fault pairs and calculate the diversity with respect to the worst-case fault pairs. In Sec. 5.1, we describe techniques to calculate the cost function of a given cover consisting of multiple-output prime implicants.

5.1. Cost Function Computation

It is clear from our previous discussion that unlike conventional two-level minimization the cost function involved during the covering step has two components. (i) The area (estimated by calculating the number of logic gates and the literal count) of the synthesized design and (ii) The expected diversity of fault pairs with respect to the given implementation N_1. The calculation of the area component of the cost function remains the same as in conventional two-level minimization. A high-level sketch of the procedure to compute diversity is shown below.

Compute Expected Diversity
Input: Function Truth Table, Implementation N_1, a MOPI cover C for implementation of N_2
Output: Expected Worst-case Diversity

Procedure:
1. For each fault f_1 in N_1
2. For each fault f_2 in N_2
3. Find $d_{1,2}$ value of (f_1, f_2)
4. Choose an f_2 so that (f_1, f_2) has the least $d_{1,2}$
5. (f_1, f_2) constitutes a worst-case fault pair
6. Return the mean $d_{i,j}$ over the worst-case fault pairs

Let us consider the example in Fig. 5.3. Consider the fault f_1 to be $p/0$. We have to find fault f_2 in N_2 such that (f_1, f_2) is the worst-case fault pair. Since N_2 implements the truth table with complemented outputs, f_2 cannot be a stuck-at-0 fault. Since f_1 is $p/0$ in N_1, the value on Z_1 (or Z_1) may be changed from the fault-free value of 1 to the faulty value 0 (but not the other way). Similarly, if f_2 is a stuck-at-0 fault at the output of an AND gate (or input of an OR gate) in N_2, then the values on the outputs of N_2 can get changed from the fault-free value of 1 to the faulty value of 0 (but not the other way). Since the outputs of N_2 are complements of the outputs of N_1, the two implementations will never produce identical erroneous outputs when f_2 is a stuck-at-0 fault. Thus, f_2 must be a stuck-at-1 fault at the output of a gate in N_2. Let $f_2 = q/1$ be a candidate for the worst-case fault pair. This fault produces $Z_2 = 0$ and $Z_3 = 0$ in N_2. Since the output Z_1 is not affected by $q/1$ in the second implementation, N_2 will not produce any error on Z_1. This means that, N_1 can produce an error only at Z_2 with the fault-free values $Z_1 = 1$ (for the fault to be excited p must be equal to 1; this means that the fault-free value of Z_1 is 1) and $Z_3 = 0$ that can be identical to an erroneous response from N_2. The computation of the $d_{1,2}$ value of the fault pair (f_1, f_2) = ($p/0$, $q/1$) is shown next.

Find $d_{1,2}$ value of (f_1, f_2) = ($p/0$, $q/1$)
V_1 = {input combinations covered *only* by the AND gate with output p for function Z_2 in N_1}
V_2 = {input combinations covered by the AND gate with output p and also by some other MOPI for Z_1 in N_1}
$V_3 = V_1 \cap V_2$ /*Input combinations producing error at Z_2 and fault-free value of $Z_1 = 1$ in N_1*/
$V_4 = V_3 \cap$ {input combinations for which fault-free value of Z_3 is 0}
V_5 = {input combinations for which (fault-free value of Z_1 is 1) AND(fault-free value of Z_2 or Z_3 or both is 1)}
$V = V_4 \cap V_5$ /* Input combinations for which N_1 and N_2 produce identical errors */
$d_{1,2} = 1 - \frac{|V|}{2^n}$, n is the number of primary inputs of the multi-output logic function

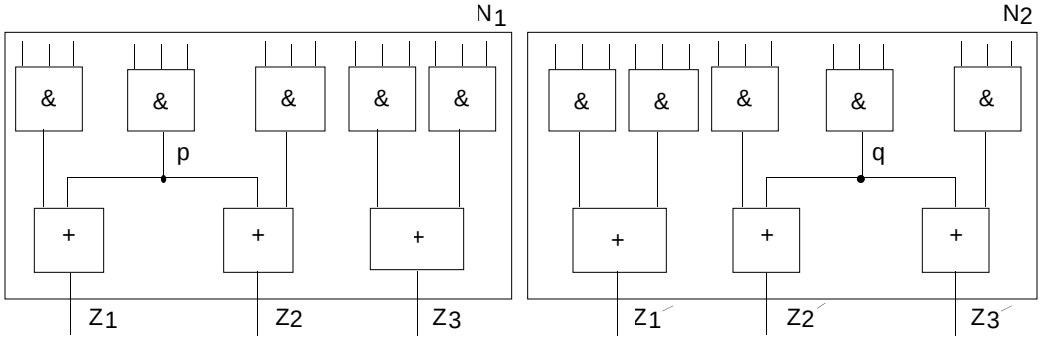

Figure 5.3. Computation of the diversity cost function

The reader is reminded that N_2 is implemented by synthesizing the truth table obtained by complementing the outputs in the truth table of N_1 and then adding inverters at the outputs of the synthesized design. Since we are considering only two-level logic circuits (in AND-OR form), for a given fault f_1 in N_1, the candidates for f_2 are faults at the outputs of the AND and the OR gates. This is because, any fault at the input of an AND (OR) gate is equivalent to or dominated by a fault at the output of that AND (OR) gate. Careful analysis of the above computation shows that in this case $V = V_4$. The case where f_1 is a stuck-at-1 fault is symmetric because, in that case f_2 must be a stuck-at-0 fault in N_2. It is clear from the above discussion that, for both stuck-at-0 and stuck-at-1 faults, we have to consider only one erroneous output combination with all 1's at the outputs affected by the stuck-at-1 fault. This implies a potential decrease in the number of input combinations for which both N_1 and N_2 produce identical erroneous outputs.

5.2. Example

In this section, we illustrate the above technique with the help of an example. Consider the truth table of the combinational logic function shown in Table 5.2. Figure 5.4 shows the implementation N_1 of the logic function of Table 5.2. Our aim is to synthesize the second implementation N_2 for the above logic function in order to maximize diversity. For using our technique, we first list all the MOPIs for the logic function in Table 5.3 which is obtained by complementing the outputs of Table 5.2. For the logic function of Table 5.3, the MOPIs together with the tags are shown in Table 5.4. We have to choose a set of MOPIs that minimizes the cost of the implementation and maximizes diversity between the implementations.

Table 5.2. Logic function

a b c	f_1 f_2
0 0 0	1 0
0 0 1	0 0
0 1 0	0 0
0 1 1	1 1
1 0 0	0 0
1 0 1	1 1
1 1 0	1 1
1 1 1	0 1

Table 5.3. Complemented outputs

a b c	g_1 g_2
0 0 0	0 1
0 0 1	1 1
0 1 0	1 1
0 1 1	0 0
1 0 0	1 1
1 0 1	0 0
1 1 0	0 0
1 1 1	1 0

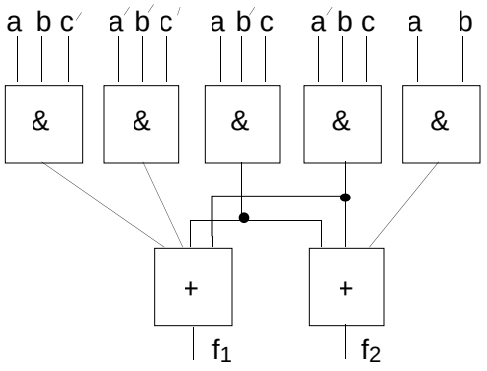

Figure 5.4. An implementation of the function in Table 5.2.

As shown in Table 5.4, there are 6 MOPIs for the logic function of Table 5.3. The MOPI covering table is shown in Table 5.5. Note that MOPIs B, C, E and G must always be selected for implementing the function g_1. Let us consider a set of MOPIs that correspond to the implementation shown in Fig. 5.5.

Table 5.4. Multiple-output prime implicants for the logic function of Table 5.3.

	a b c	Tag (g_1, g_2)
A	0 0 -	0 1
B	0 0 1	1 1
C	0 1 0	1 1
D	0 - 0	0 1
E	1 0 0	1 1
F	- 0 0	0 1
G	1 1 1	1 0

Table 5.5. MOPI Covering Table

	g_1				g_2			
	1	2	4	7	0	1	2	4
A					X	X		
B	X					X		
C		X					X	
D					X		X	
E			X					X
F					X			X
G				X				

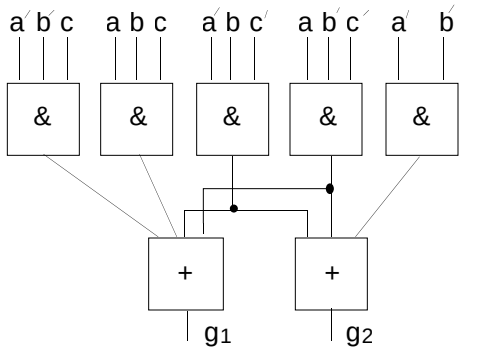

Figure 5.5. An implementation of the function in Table 5.3.

Let us consider the faults in the implementation of Fig. 5.4. The "interesting" faults are the ones at the outputs of the AND gates that have fanouts. This is because, by Theorem 1, the faults at the output leads of the implementation in Fig. 5.5 form worst-case pairs with the faults at the inputs or outputs of the AND gates without any fanout in Fig. 5.4. Consider a stuck-at-1 fault at the output of the AND gate $a'bc$ in implementation N_1 (Fig. 5.4). The set of input combinations for which N_1 produces incorrect output combination 11 is {000, 001, 010, 100, 111}. For implementation N_2 (Fig. 5.5), the candidate faults that produce the worst-case fault pairs are $g_1/0$, $g_2/0$ and stuck-at-0 faults at the outputs of the AND gates $a'bc'$ and $ab'c'$. For $g_1/0$, the set of input combinations for which N_2 produces incorrect output combination 00 is given by {111}. For $g_2/0$, the set of input combinations for which N_2 produces incorrect output combination 00 is given by {000}. The set of input

combinations for which the N_2 produces erroneous output combination 00 in the presence of a stuck-at-0 fault at the output of the AND gate $a'bc'$ is {010}. Similar case holds for a stuck-at-0 fault at the output of the AND gate $a'bc'$. Thus, if we use the implementation in Fig. 5.5 for the second module instead of replicating the implementation in Fig. 5.4, we find that the $d_{i,j}$ value for the worst-case fault pair with a stuck-at-1 fault at the output of the AND gate $a'bc$ in N_1, changes from 0.375 (1-$\frac{5}{8}$) to 0.875 (1-$\frac{1}{8}$) in the worst-case.

6. Multi-Level Logic Synthesis

Conventional techniques for synthesis of multi-level logic circuits rely on a set of logic transformations that are applied systematically to the input Boolean network to obtain a final network. Since the multi-level transformations change the structure of the Boolean network under consideration, they can potentially increase or decrease the degree of diversity between two logic networks. Hence, it is important to examine the effects of these transformations on a Boolean network as far as its diversity with respect to a given Boolean network is concerned. There are four main types of transformations that are used during multi-level logic synthesis. They are: (i) Single-cube extraction, (ii) Double-cube extraction, (iii) Re-substitution and (iv) Vertex elimination [Rajski 92, De Micheli 94]. These transformations are also present in the widely used *rugged script* that is provided by the *Sis* [Sentovich 92] logic optimization system. In this paper, we analyze each of these logic transformations. For most of the cases, we will follow the definitions in [Rajski 92]. Our goal is to identify a set of transformations that we can use safely without sacrificing the gains obtained from two-level synthesis. By applying some of these transformations, we can possibly increase the diversity between two implementations. This observation leads to interesting optimization problems not studied in here.

6.1. Single-Cube Extraction

"Single-cube extraction is the process of extracting cubes which are common to two or more cubes" [Rajski 92]. For example, let us suppose that we have a logic function $f = abA_1 + abA_2 + \ldots + abA_n$. When single-cube extraction is performed, we have an intermediate node (AND gate) $C = ab$ and $f = CA_1 + CA_2 + \ldots + CA_n$. Figure 6.1 illustrates this procedure.

Let us consider a stuck-at-0 fault at the AND gate output C. This implies that the value of the output f is 0 in the presence of this fault. Hence, $C/0$ is equivalent to $f/0$. Similarly, consider the fault $C/1$. Any input combination that detects this fault at output f also detects the fault $f/1$. Hence, $f/1$ dominates $C/1$. All the other stuck-at faults are tested by the same set of input combinations as the original circuit. This observation is very significant. Suppose that, we have a network K as a possible implementation of module N_2 in a duplex system. With respect to the first module N_1, the implementation K has a particular value of diversity. If we apply single-cube extraction on network K to obtain a new implementation for N_2, for any fault g in N_1, the fault in N_2 that forms the worst-case pair remains unchanged. This notion is somewhat (but not fully) similar to the notion of test-set preserving transformations [Rajski 92].

6.2. Double-Cube Extraction

"The double-cube extraction transformation consists of extracting a double cube from a single-output sum-of-products sub-expression, $AC + CB \Rightarrow C(A + B)$" [Rajski 92]. Figure 6.2 shows an example of double-cube extraction.

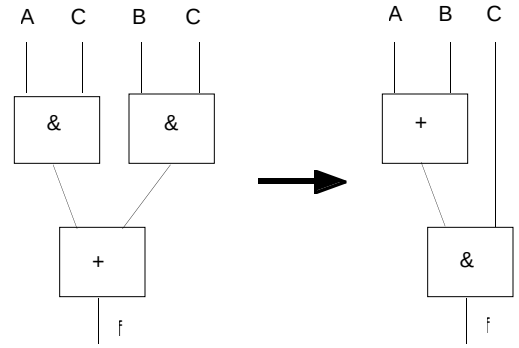

Figure 6.2. Illustration of double-cube extraction

It is very easy to see from Fig. 6.2 that all faults in the original network and the transformed network are dominated by or equivalent to stuck-at faults on f. Thus, for any fault g in N_1, the fault in N_2 that forms the worst-case pair remains unchanged.

6.3. Re-substitution

Re-substitution is a transformation that replaces all nodes corresponding to a particular logic function with a single node. It has been proved in [Rajski 92] that re-substitution is not test set preserving in general. It can be shown that, under the re-substitution transformation, it is not guaranteed that the diversity of the transformed network will not decrease. Thus, it is tricky to apply the re-substitution transformation for generating multi-level logic networks while preserving the diversity measure between two designs. However, application of synthesis scripts with and without the re-substitution transformation shows that fairly area-efficient logic circuits can be without using the re-substitution transformation [Mitra 00c].

6.4. Elimination

In the elimination transformation, an internal node of a logic network is eliminated from the Boolean network and the variable corresponding to that node is replaced by the corresponding expression in all its occurrences in the logic network [De Micheli 94]. This transformation guarantees that the diversity with respect to the worst-case fault pairs in the transformed circuit is never less than that in the circuit on which the transformation is applied. Intelligent use of this transformation can possibly increase diversity. This is a new optimization problem not considered in this paper.

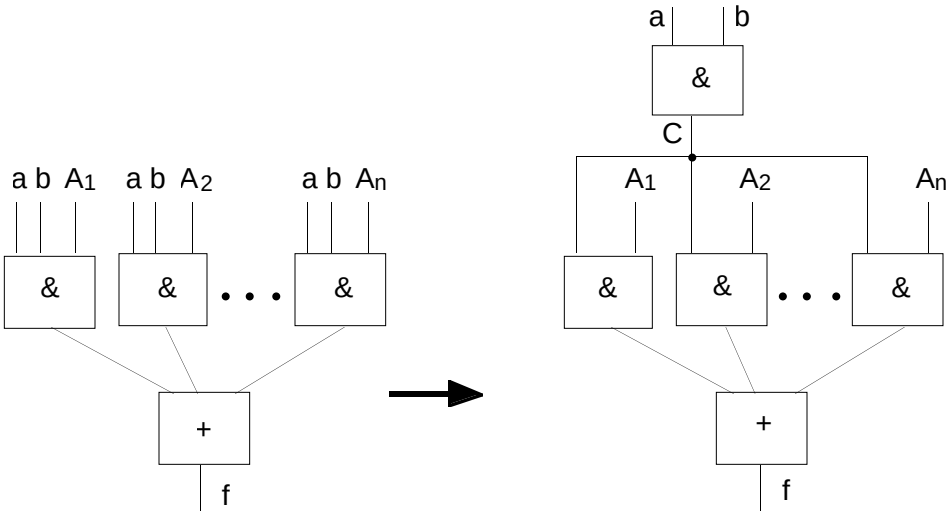

Figure 6.1. Illustration of single-cube extraction

6.5. Simplification

The simplification transformation performs two-level minimization of internal nodes of a Boolean network (if the nodes represent complex Boolean functions). The internal nodes of a Boolean network can be treated as single-output logic functions. From Theorem 1 in Sec. 5, it is clear that for any fault g in N_1, two-level minimization (simplification) of internal nodes in N_2 will not have any effect on the worst-case fault pair. The simplification transformation can be invoked using the *simplify* and *full_simplify* commands in *Sis*.

7. Summary and Conclusions

In this paper, for the first time, we have developed systematic techniques to synthesize diverse implementations of combinational logic circuits for duplex systems so that the data integrity of the resulting diverse duplex system is maximized. Unlike conventional logic synthesis (where the cost function of logic minimization is mainly determined by the area and delay of the resulting logic), our cost function has a diversity component that has to be maximized.

Unlike previous approaches for designing diverse implementations that depended on independent generation, we have identified a concrete diversity cost function in this paper. We modified the conventional two-level logic synthesis procedure to maximize the diversity component of the cost function without drastically affecting the area component. Next, we identified a set of multi-level logic transformations that can be applied without sacrificing diversity in the resulting logic structure with respect to a given implementation. We observed that there are further opportunities to increase diversity in the resulting implementation through multi-level transformations. Future research should focus on identifying new multi-level logic transformations and using the conventional multi-level transformations (that do not preserve diversity) appropriately to enhance the diversity in the resulting implementation.

The cost function for diversity, considered in this paper, is based on the $d_{i,j}$ value of the worst-case fault pairs. However, consider the case where, for a fault f_1 in N_1, there are 5 possible f_2's in N_2 and 2 possible f_2's in another implementation N_3 that produce the same diversity with respect to the worst-case fault pairs. In our current analysis, for fault f_1 in N_1, the diversity between N_1 and N_2 is the same as that between N_1 and N_3. Our current cost function can be modified to incorporate this subtle difference between N_2 and N_3. Another interesting extension is to synthesize sequential logic circuits with diversity in the encoding of the internal states.

8. Acknowledgments

This work was supported by Defense Advanced Research Projects Agency (DARPA) under Contract No. DABT63-97-C-0024. Thanks to Philip Shirvani of Stanford CRC for his comments.

9. References

[Avizienis 77] Avizienis, A. and L. Chen, "On the implementation of N-version programming for software fault-tolerance during program execution," *Proc. Intl. Computer Software and Appl.*, pp. 149-155, 1977.

[Avizienis 84] Avizienis, A. and J. P. J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, pp. 67-80, Aug. 1984.

[Briere 93] Briere, D. and P. Traverse, "Airbus A320/A330/A340 Electrical Flight Controls: A family of fault-tolerant systems," *Proc. FTCS*, pp. 616-623, 1993.

[Cormen 89] Cormen, T. H., C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, The MIT Press and McGraw-Hill Book Company, 1989.

[De Micheli 94] De Micheli, G., *Synthesis and Optimization of Digital circuits*, McGraw-Hill, 1994.

[Lala 94] Lala, J. H. and R. E. Harper, "Architectural

principles for safety-critical real-time applications," *Proc. of the IEEE*, vol. 82, no. 1, pp. 25-40, Jan. 1994.

[LSI 96] *G10-p Cell-Based ASIC Products Databook*, LSI Logic, May 1996.

[Lyu 91] Lyu, M. R. and A. Avizienis, "Assuring design diversity in N-version software: a design paradigm for N-version programming," *DCCA*, pp. 197-218, 1991.

[McCluskey 56] McCluskey, E.J., "Minimization of Boolean Functions," *Bell System Tech. Journal*, Vol. 35, No. 5, pp. 1417-1444, Nov. 1956.

[McCluskey 71] McCluskey, E. J. and F. W. Clegg, "Fault Equivalence in combinational logic networks," *IEEE Trans. Computers*, Vol. C-20, No. 11, pp. 1286-1293, Nov. 1971.

[McCluskey 86] McCluskey, E. J., *Logic Design Principles with Emphasis on Testable Semicustom Circuits*, Prentice-Hall, 1986.

[McCluskey 00] McCluskey, E. J., and C. Tseng, "Stuck-Fault Tests vs. Actual Defects," *Intl. Test Conf.*, 2000.

[Mitra 99a] Mitra, S., N. Saxena and E. J. McCluskey, "A Design Diversity Metric and Reliability Analysis for Redundant Systems," *Proc. Intl. Test Conf.*, pp. 662-671, 1999.

[Mitra 99b] Mitra, S., N. Saxena and E. J. McCluskey, "A Design Diversity Metric and Analysis of Redundant Systems," *Technical Report, Center for Reliable Computing, CRC-TR 99-4*, Stanford University, 1999. (http://crc.stanford.edu)

[Mitra 00a] Mitra, S., N. Saxena and E. J. McCluskey, "Common-Mode Failures in Redundant VLSI Systems: A Survey," *IEEE Trans. Reliability*, 2000, To appear.

[Mitra 00b] Mitra, S., and E. J. McCluskey, "Which Concurrent Error Detection to Choose ?," *Proc. International Test Conf.*, 2000, To appear.

[Mitra 00c] Mitra, S., "Diversity Techniques for Concurrent Error Detection," *PhD Thesis*, Department of Electrical Engineering, Stanford University, 2000. (http://crc.stanford.edu)

[Pradhan 96] Pradhan, D. K., *Fault-Tolerant Computer System Design*, Prentice Hall, 1996.

[Rajski 92] Rajski, J. and J. Vasudevamurthy, "The testability-preserving concurrent decomposition and factorization of Boolean expressions," *IEEE Trans. CAD*, Vol. 11, No. 6, pp. 778-793, June 1992.

[Riter 95] Riter, R., "Modeling and Testing a Critical Fault-Tolerant Multi-Process System," *Proc. FTCS*, pp. 516-521, 1995.

[Sentovich 92] Sentovich, E. M., *et al.*, "SIS: A System for Sequential Circuit Synthesis," *ERL Memo. No. UCB/ERL M92/41*, EECS, UC Berkeley, CA 94720.

[Siewiorek 92] Siewiorek, D. P. and R. S. Swarz, *Reliable Computer Systems: Design and Evaluation*, Digital Press, 1992.

[Spainhower 99] Spainhower, L. and T. A. Gregg, "S/390 Parallel Enterprise Server G5 fault tolerance," *IBM Journal of Res. & Dev.*, Vol. 43, pp. 863-873, 1999.

[Tamir 84] Tamir, Y. and C. H. Sequin, "Reducing common mode failures in duplicate modules," *Proc. ICCD*, pp. 302-307, 1984.

[To 73] To, K., "Fault Folding for Irredundant and Redundant Combinational Circuits," *IEEE Trans. Computers*, Vol. C-22, No. 11, pp. 1008-1015, Nov. 1973.

[Tohma 71] Tohma, Y. and S. Aoyagi, "Failure-tolerant sequential machines with past information," *IEEE Trans. Computers*, Vol. C-20, No. 4, pp. 392-396, April 1971.

[Trivedi 82] Trivedi, K.S., Probability and statistics with reliability, queuing, and computer science applications, Prentice Hall, Englewood Cliffs, NJ, USA, 1982.

[Webb 97] Webb, C. F., and J. S. Liptay, "A High Frequency Custom S/390 Microprocessor," *IBM Journal Res. and Dev.*, Vol. 41, No. 4/5, pp. 463-474, 1997.

Appendix 1. A Computationally Intensive Problem

In this section, we illustrate the difficulties in the cost function computation when the second implementation (N_2) is synthesized in AND-OR form directly from the truth table of N_1 (without complementing the outputs). For our current discussion, we assume that the truth table of N_1 is directly used to synthesize N_2 (unlike in Sec. 5 where the truth table with complemented outputs is used to synthesize N_2). Since we are considering only two-level logic circuits (in AND-OR form), for a given fault f_1 in N_1, the candidates for f_2 are faults at the outputs of the AND and the OR gates. This is because, any fault at the input of an AND (OR) gate is equivalent to or dominated by a fault at the output of that AND (OR) gate. The problem of calculating the $d_{1,2}$ values when f_1 is a stuck-at-0 fault is tricky and is computationally intensive. Here we illustrate the complications in finding the worst-case fault pair for a stuck-at-0 fault using the example in Fig. 5.2b.

When f_1 is a stuck-at-0 fault at the output of an AND gate p, the implementation N_1 can produce the following incorrect combinations at outputs Z_1 and Z_2: (a) $Z_1 = 0$, $Z_2 = 0$ and (b) $Z_1 = 1$ and $Z_2 = 0$. Similarly, when f_2 is a stuck-at-0 fault at the output of AND gate p in the second implementation, N_2 can produce the following incorrect combinations at outputs Z_2 and Z_3: (a) $Z_2 = 0$, $Z_3 = 0$ and (b) $Z_2 = 0$, $Z_3 = 1$. Thus, for calculating the $d_{1,2}$ value for the fault pair, we must consider each of the following erroneous combinations on the outputs Z_1, Z_2 and Z_3: (a) $Z_1 = 0$, $Z_2 = 0$, $Z_3 = 0$, (b) $Z_1 = 0$, $Z_2 = 0$, $Z_3 = 1$, (c) $Z_1 = 1$, $Z_2 = 0$, $Z_3 = 0$ and (d) $Z_1 = 1$, $Z_2 = 0$, $Z_3 = 1$.

This computation can be intensive because the number of erroneous output combinations to be considered can be exponential in the number of outputs that depend on the gates whose inputs or outputs are stuck at 0. However, as we have shown in Sec. 5.1, we can eliminate this problem of considering an exponential number of error patterns if we use the truth table of N_1 with complemented outputs for implementing N_2.