NON-SELF-TESTABLE FAULTS IN DUPLEX SYSTEMS

Subhasish Mitra, Nirmal R. Saxena and Edward J. McCluskey

Center for Reliable Computing (http://crc.stanford.edu)
Departments of Electrical Engineering and Computer Science
Stanford University, Stanford, California

Abstract

Adaptive Computing Systems (ACS) provide new opportunities to implement different concurrent error detection and fault-tolerance techniques to build dependable systems with high reliability and availability. In this paper, we first review the opportunities for implementing design diversity in dependable adaptive computing systems. The main contribution of this paper is to identify non-self-testable fault pairs in a duplex system and enhance the number of self-testable fault pairs in the system through test point insertion. This enhancement has a direct effect on increasing the availability of the dependable system. We show that our algorithm for identifying non-self-testable fault pairs has orders of magnitude improvement in the execution time over other competitive techniques. Our results also reveal a significant reduction in the number of test points required for duplex systems with diverse implementations of the same logic function compared to duplex systems with identical implementations.

1. Introduction

Adaptive Computing Systems (ACS) represent a new emerging technology that is derived by combining microprocessor, memory and configurable logic. ACS designs deliver ASIC-like performance and cost, with the time-to-market and flexibility of *Programmable Logic Devices* (PLDs), and the ease of programming of microprocessors [Chameleon 99]. For using ACS applications in nuclear reactors, fly-by-wire systems, satellites and other remote systems, and life-critical systems, *dependability* (availability, reliability, testability, maintainability and fault-tolerance) of these systems is an important requirement. The work presented in this paper is a part of the DARPA-funded ROAR project at the Stanford University's Center for Reliable Computing. ROAR is an acronym for *Reliability Obtained by Adaptive Reconfiguration*. The details of the ROAR project can be obtained from [Saxena 98] and [ROAR 99].

Figure 1.1 shows the system architecture that we are considering in our project. We call this the *ROAR architecture*. The ROAR architecture consists of a multi-threaded processor, a configurable coprocessor (implemented using *field programmable gate arrays* (FPGAs)), a memory and an input/output system.

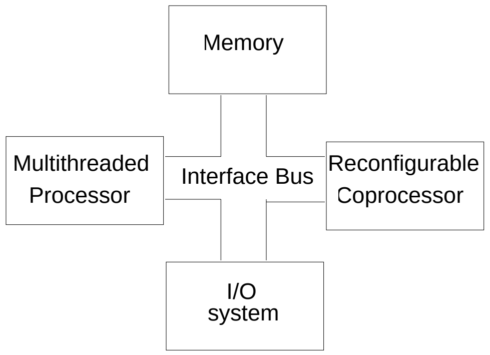

Figure 1.1. ROAR architecture

For designing dependable systems using the above architecture, we plan to use *design diversity* while incorporating redundancy in the applications running on the multi-threaded processor and the designs mapped to the configurable coprocessor. Design diversity has long been used to protect redundant systems against common-mode failures [Avizienis 84][Lala 94]. Common-mode failures result from failures that affect more than one module of a redundant system at the same time, generally due to a common-cause. The conventional notion of diversity relies on "independent" generation of "different" implementations. For the configurable coprocessor, the programmability of the FPGAs opens up new opportunities to create diversity in the designs mapped to FPGAs. In an ACS environment, we can create diversity by synthesizing and downloading different implementations into FPGAs at any time. Thus, there is no need to manufacture multiple diverse ASICs. For applications running on the processor, we can create diversity among the redundant software modules — a technique to achieve this is called *N-version programming* [Avizienis 77][Lyu 91]. In the micro-architecture level, with proper compilation techniques, we can create diversity among the different redundant threads of computation running on the multi-threaded processor and the FPGAs. Thus, diversity can be applied to the different computation models [ROAR 99] of our system.

For making choices in designing redundant systems using diversity, it is important to quantify the diversity and its effects on the dependability of the redundant systems. The conventional concept of diversity, however, is qualitative and does not provide a basis to compare the reliabilities of two redundant systems having diversity. At Stanford, we have developed a metric for design diversity and used that metric to analyze the reliability of redundant systems [Mitra 99a]. In this paper, we will discuss the effects of diversity on self-testing properties of faults in duplex systems in Sec. 2. In Sec. 3, we present techniques to identify non-self-testable fault pairs in a duplex system. Section 4 describes test point insertion techniques to increase the self-testability of duplex systems. We present the results in Sec. 5. In Sec. 6 we discuss some techniques to further minimize the number of test points. Finally, we conclude in Sec. 7.

2. Self-testing Properties of Fault Pairs in Duplex Systems

In this section, we discuss the effects of having design diversity on the self-testing properties of fault pairs of a duplex system. Consider a duplex system containing two implementations N_1 and N_2 of the same logic function and a comparator comparing the outputs obtained from these implementations. The duplex system is *self-testing* with respect to a fault pair (f_1, f_2) (f_1 affecting N_1 and f_2 affecting N_2) if and only if, there exists an input combination for which the two implementations produce different outputs in the presence of the faults. The corresponding fault pair is said to be *self-testable*. If the two implementations produce different outputs in the presence of the fault pair, then the comparator will produce a *Mismatch* signal so that repair action can be initiated.

Table 2.1. Self-testing properties of duplex systems

Circuit Name	Implementations	# SSF pairs (millions)	% escapes
Z5xp1	Identical	0.30	0.73
	Diverse	0.36	0.02
clip	Identical	0.49	0.58
	Diverse	0.46	0.02
inc	Identical	0.26	0.84
	Diverse	0.25	0.03
rd84	Identical	0.16	1.1
	Diverse	0.23	0.04

In Table 2.1 we show some simulation results comparing the number of self-testable fault pairs in duplex systems with identical and diverse implementations. For the purpose of the simulation, we assume that the failures show up as single-stuck faults in each of the two implementations under consideration. The different implementations were obtained by synthesizing the logic circuits in different ways. More detailed information about the synthesis of different implementations can be obtained from Sec. 5. As mentioned earlier, the self-testing property ensures that, in the presence of failures that affect the two implementations under consideration, we can detect the presence of the failures. This property has a direct effect on increasing the system availability, as illustrated in [Mitra 99b]. A system with a high percentage of self-testable fault pairs has the following advantages:

1. During idle cycles when the system is not used, special input combinations (test patterns) can be applied. Having a self-testable fault pair guarantees that it will be detected using the existing setup of duplex systems, when these special input combinations are applied.
2. It is easier to perform an off-line test on the system and detect the presence of faults.
3. It may be easier to locate the faulty unit and perform repair.

Thus, for a given duplex system, it is important to find the fault pairs that are not self-testable — we have to use some special techniques to handle these fault pairs. In the next section, we describe a technique to identify these fault pairs. In the remainder of this paper, we will consider single-stuck at faults.

3. Identifying Non-self-testable Fault Pairs

In this section, we describe our technique to identify fault pairs that are not self-testing in a duplex system. We call the two implementations in a duplex system N_1 and N_2. For each fault in each of the implementations, we calculate a *signature* corresponding to that fault. A fault pair is not self-testable if the two faults under consideration have the same signature. The algorithm for identifying the non-self-testable fault pairs (Algorithm 1) is shown below. The technique to calculate the signature of each fault is described in the next paragraph.

```
ALGORITHM 1
For each input combination $p$
        Perform logic simulation on $N_1$
        Store response $R_1$
        For each fault $f_1$ in $N_1$
                Inject $f_1$ in $N_1$ and store response $R_1(f_1)$
                If $R_1$ and $R_1(f_1)$ are different
                        Update signature of $f_1$
        Endfor
        Perform logic simulation on $N_2$
        Store response $R_2$
        For each fault $f_2$ in $N_2$
                Inject $f_2$ in $N_2$ and store response $R_2(f_2)$
                If $R_2$ and $R_1(f_2)$ are different
                        Update signature of $f_2$
        Endfor
Endfor
For each fault $f_1$ in $N_1$ and $f_2$ in $N_2$
        If signatures of the two faults are different
                The fault pair ($f_1$, $f_2$) is self-testing
        Else
                The fault pair ($f_1$, $f_2$) is not self-testing
Endfor
```

For calculating the signature associated with each single-stuck fault, we use a Multiple-Input Signature Register (MISR) and a counter. For example, suppose that we want to calculate the signature associated with a particular fault *f* in N_1. For each input combination, if the response of N_1 in the presence of f is different from the fault-free response, then the counter is incremented and the faulty response is compacted into the MISR. The counter counts the number of test patterns that detect a particular fault. The signature of a fault is the given by the pair <*content of the MISR, value of the counter*>. Our results show that using the counter or the MISR alone results in highly sub-optimal results. The sub-optimality arises from the fact that, two faults f_1 and f_2 may have the same signature but still the fault pair (f_1, f_2) may be self-testable. In this situation, our algorithm will declare the fault pair (f_1, f_2) to be non-self-testable. This situation is referred to as *signature aliasing*. However, as the results in Sec. 5 indicate, using both the counter and the MISR, we obtain very close to optimum and often fully optimum results with negligible aliasing.

If all the 2^n input combinations are applied and if we assume that there no aliasing in the signatures, then two faults f_1 and f_2 are not self-testing if and only if they have the same signature. This follows from the following fact. If two faults f_1 and f_2 have the same signature, then they are detected by the same set of test patterns and the corresponding faulty implementations (N_1 and N_1) produce identical faulty outputs in response to these test patterns. Thus, in a duplex system, in the presence of these faults, there will be no disagreement at the outputs of the two implementations. Thus, the fault pair is not self-testable. Conversely, if two faults f_1 and f_2 are not self-testable, then the corresponding faulty implementations always produce identical outputs. Thus, the two faults will always have the same signature.

It may be noted that any aliasing arising from signature computation will result in declaring a self-testable fault pair as being not self-testable. Thus, aliasing results in one-sided error and does not produce false positives (unlike fault detection where aliasing may cause a defective part to be treated as a fault-free part).

A similar argument holds for the number of input patterns that must be applied to identify the self-testable fault pairs. In the worst case, we have to apply all the 2^n input combinations for identifying self-testable fault pairs that are detected by very few input (possibly one) combinations. Note that, the self-testable fault pairs, that are not detected in course of our simulation with the reduced number of input combinations, will be declared as being not self-testable. Thus, using a reduced number of input combinations during simulation results in one-sided error and does not produce any false positive. We may declare a self-testable fault pair to be non-self-testable. However, the reverse situation will never occur. Thus, depending on the number of inputs of the circuits, the execution time that we can spend and the level of accuracy we want, we can tune the number of input combinations that we should consider in our technique.

The above feature of our algorithm makes it distinct from using conventional automatic test pattern generation (ATPG) tools to find whether a fault pair is self-testable or not.

A conventional ATPG tool can be used to serve the purpose in the following way. We can inject faults f_1 and f_2 in implementations N_1 and N_2, respectively and perform Exclusive-OR operations on the corresponding outputs of the two modules. Finally, we can logically OR the outputs of the exclusive-or gates — if we can detect a stuck-at 0 fault at the output of the OR gate, then the fault pair under consideration is self-testable. Otherwise, the fault pair is not self-testable. However, this problem translates to the redundancy identification problem which is NP-complete and has an exponential complexity in the worst case. Figure 3.1 shows the ATPG-based approach.

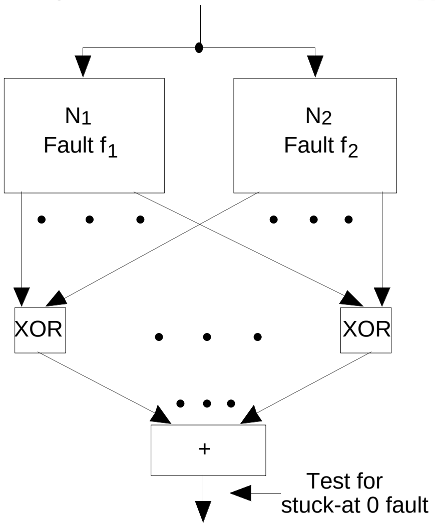

Figure 3.1. An ATPG-based approach

We can further reduce the running time of Algorithm 1 by using deductive fault simulation techniques [Abramovici 90][Armstrong 72]. In that case, we can perform word-level operations and bring down the complexity of fault-simulation by a factor of the word length of the machine on which the program is implemented. The experimental results presented in Sec. 5 clearly show a distinct advantage of using Algorithm 1 compared to the ATPG-based approach, so far as the execution time is concerned. The results also show that the quality of the results produced by Algorithm 1 are not eclipsed by aliasing if we use both the MISR and the counter for signature analysis.

In the next section, we will describe a technique to choose test points so that all fault pairs that are identified as being non-self-testable (using techniques described in this section) become self-testing.

4. Enhancing Self-testing Properties Using Test Points

In the literature on digital testing, test points have been used to enhance the fault-coverage of logic circuits

[Eichelberger 83]. In this section, we will discuss test point insertion techniques to enhance the self-testability of duplex systems. It is well-known that there are two types of test points, viz., control points and observation points. In Sec. 4.1 and 4.2, we describe self-testability enhancement techniques using control and observation points, respectively.

4.1. Self-testability Enhancement Through Control Points

Consider a duplex system consisting of two implementations of a combinational logic function. The outputs from the two implementations are compared bit-by-bit and a mismatch signal is generated when any output pair does not match. In response to the mismatch signal, subsequent repair and recovery actions can be taken.

In this discussion, we assume that the failures manifest themselves as single-stuck faults in the two modules of the duplex system under consideration.

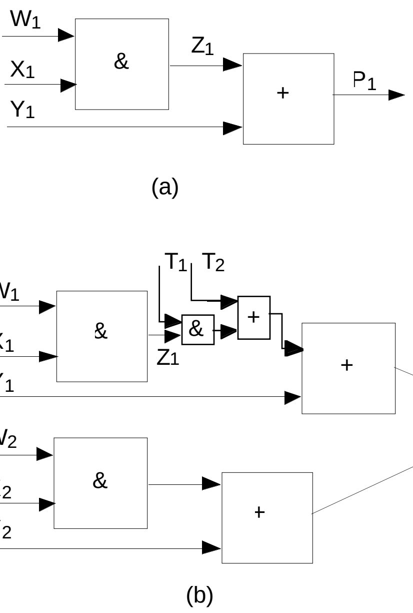

Figure 4.1. Test Control Points

Consider a duplex system with two identical modules each implementing the logic circuit shown in Fig. 4.1a. Consider the fault pair in the presence of which, the signal line Z_1 is stuck-at-0 in both the modules. It is obvious that the duplex system will never produce any mismatch signal in the presence of these two faults — thus, the fault pair under consideration is not self-testable. Now suppose that for one of the two modules, we add the test points T_1 and T_2 as shown in Fig. 4.1b. We make T_1 = 0 and T_2 = 0 and *apply a test pattern* for Z_1 stuck at 0. If the fault pair is not present, a mismatch signal will be produced — if the fault pair is present, no mismatch signal will be produced. This observation can be used to detect the presence of the fault pair. A similar case arises for the fault pair Z_1 is stuck at 1 in both the modules. Thus, control test points can enhance the self-testability of fault pairs in a duplex system. It may be noted that, in a duplex system with two identical implementations of the logic circuit in Fig. 4.1a, the fault pairs affecting the same leads in the two implementations are not self-testable. Thus, we have to add these test points at each lead of the circuit in Fig. 4.1a. Now, we discuss the advantages and disadvantages of using control points to enhance the self-testability of duplex systems.

The primary advantage of using control points is that we can utilize the available resources (comparators) of the duplex system for observing the response of the system in response to an input combination. Thus, we do not have to store simulated fault-free responses and compare the system response with these pre-stored responses to detect the presence of faults. However, we have to ensure that when the test points are activated, we apply a test vector for the untestable stuck-at fault pair under consideration. This can be achieved by using deterministic test patterns or pseudo-random patterns using LFSRs. If LFSRs are used, some technique similar to the mapping logic technique [Touba 95] can be used for test point activation. For detecting the presence of faults when the test points are activated, we can XOR the *mismatch* signal output of the comparator with a *Test* signal that is 1 when one of the test points is activated. The *Test* signal may be generated externally or by the test controller.

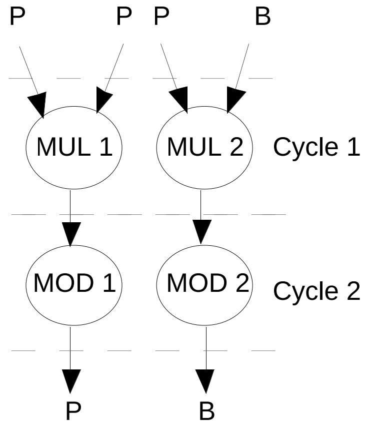

Figure 4.2. L-Modular exponential algorithm implementation

As a consequence of the previous observation, we can utilize the idle cycles of the duplex system in the following way. During the idle cycles of the system, we can apply input combinations and activate appropriate test points (if necessary) to detect different fault pairs. We explain the corresponding effect in the architecture level using the following example. Consider the example of computing *L*-modular exponential for encryption purposes reported in [Saxena 98] and shown in Fig. 4.2. We can see that the multipliers MUL1 and MUL2 are used in every alternate cycle (Cycle 1, Cycle 3, Cycle 5, etc.). Now, suppose that we have a duplex implementation of the multipliers. Thus, during every even cycle, we can apply test patterns to the multiplier inputs. This can be done by adding extra instructions if the algorithm is implemented in the processor, or by using an LFSR if it is implemented in FPGA. The basic advantage here is that, we do not need any extra mechanism to process the response of the multipliers during the idle cycles.

The drawbacks of using the control points are the following. They require extra area and may affect the performance of the circuit. Moreover, this technique requires more design effort.

4.2. Self-testability Enhancement Through Observation Test Points

Instead of adding control points, we can increase the self-testability of the duplex system under consideration by observing the node Z_1 of the circuit in Fig. 3.1(a). For observation purposes, we can perform signature analysis or directly observe the node under consideration. This approach has a distinct advantage because we do not have to add extra gates unlike control points. However, we have to route the observation points to signature analyzers and perform comparison of the computed signature of the logic values on the observation test points with the golden signature. An implementation of test observation points on a configurable computing test-bed demonstrates no performance overhead due to the insertion of test observation points. For self-testable fault pairs, we can steal idle cycles of the system to apply test patterns. However, for the non-self-testable patterns we have to observe the logical values on the added test points (possibly through signature analysis). Thus, fault simulation and storage of fault-free signatures are necessary.

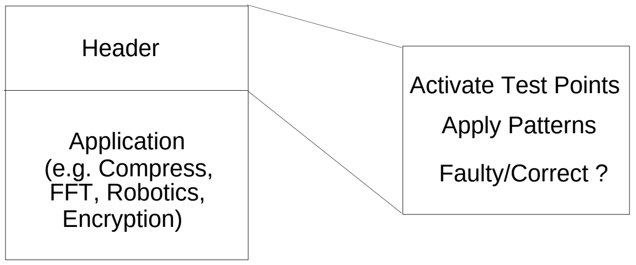

Figure 4.3. Applications preceded by a testing phase

For applying the idea of test points to ACS, each application can be preceded by a testing phase. Recent results show large speedups for different applications using ACS. For example, at Stanford Center for Reliable Computing, an ACS implementation of the DES encryption algorithm [FIPS 77] shows more than a factor of 20 speed-up over the best software implementation. Hence, it is expected that the use of a testing phase as a header for each application will not adversely affect the system performance. A high level block diagram for an application preceded by a testing phase is shown in Fig. 4.3. Input patterns may be applied using *LFSRs* (Linear Feedback Shift Registers) or by compiling deterministic finite state machines.

It may be argued that since we are using an off-line test as a preamble for every application, it is sufficient to observe the outputs of the individual modules of the duplex system during the off-line test rather than putting intermediate test points. However, there are cases where the number of test points is lower than the number of outputs. Moreover, with test points, the faults can be detected more easily with a large number of patterns — we just have to excite the fault and not worry about sensitizing the fault effect. Thus, for detecting the presence of the non-self-testable fault pairs, we have to toggle the individual points rather than propagating the fault effects to the outputs — thus, calculating the fault-free signature on the observation points is also very easy; the value is a 0 or a 1 if we target a stuck-at 1 or a stuck-at 0 fault, respectively. Finally, test points help us to perform quick fault-location and self-repair. Thus, use of test points (control and observation) is helpful for detect common-mode and multiple failures in duplex systems. Both these techniques have their relative merits and demerits. An interesting optimization problem can be formulated to determine the control and observation points that should be used for a given application.

4.3. Choice of Test Points

For duplex systems with identical implementations, the test points to be inserted can be determined very easily. In such a system with m leads in each implementation, there must be at least $2m$ single stuck-at fault pairs that are not self-testable in a duplex system. These are the same faults on the corresponding lines of the two identical implementations. Then we need m test points so that all the single stuck-at fault pairs are detected in a duplex system.

For a duplex system with diverse implementations, we can find the non-self-testable fault pairs using Algorithm 1. Now, we have to choose the minimum number of test points that make all these fault pairs self-testable. This problem can be formulated as a *Covering* problem. A non-self-testable fault pair (f_1, f_2) can be detected if we put a test point on the signal line that is affected by f_1 or f_2. Thus, for each non-self-testable fault pair, we have two candidate signal lines and at least one of these signal lines have to be selected in order to detect that fault pair. Finding the minimum number of test points that must be selected to detect all the non-self-testable fault pairs is the same as the Vertex Covering Problem [Cormen 89]. We explain the idea with the help of the following example.

Let us suppose that we have 5 non-self-testable fault pairs, (A_1, B_1), (A_1, B_2), …, (A_3, B_4), as shown in Table 4.1. Here, (A_1, B_1) stands for fault A_1 in the first implementation (N_1) and for fault B_1 in the second implementation (N_2). For detecting this fault pair, we have to add a test point at the signal line corresponding to

A_1 in the first implementation (N_1) or at the signal line corresponding to B_1 in the second implementation (N_2). The rows of Table 4.1 are the candidate test points and we put a X in an entry of the table if the fault pair in the column of that entry can be detected by inserting a test point at the signal line corresponding to the row of the entry. Now, we have to select the minimum number rows so that we have X's under every column of the table. This is the same as the classical covering problem encountered while finding the minimum number of prime implicants to represent a Boolean function [McCluskey 56] and has been studied extensively. It is also well-known that this problem is NP-complete and has exponential complexity in the worst case.

Table 4.1. Test point insertion example

	(A_1, B_1)	(A_1, B_2)	(A_1, B_3)	(A_2, B_4)	(A_3, B_4)
A_1	X	X	X		
A_2				X	
A_3					X
B_1	X				
B_2		X			
B_3			X		
B_4				X	X

Hence, we implemented a simple heuristic algorithm to solve the problem. The heuristic is to sort the test point candidates in decreasing order of the number of fault pairs that become testable when a particular test point is selected. We pick the candidate at the top of this sorted list and remove the columns (fault pairs) that are detected by this test point. We repeat the above procedure until all the fault pairs are detected. For example, for Table 4.1, there are three fault pairs that are detected by test point on the signal line corresponding to A_1. All other test points detect only one or two fault pairs. Thus, we choose to put a test point on the signal line corresponding to A_1. Due to the insertion of this test point, all fault pairs involving A_1 are detected — we remove the corresponding columns from the table. Now, the new table has only two columns corresponding to (A_2, B_4) and (A_3, B_4). Both these fault pairs can be detected by choosing a test point corresponding to B_4 in N_2. Thus, we need only two observation test points (or four control test points) to detect all the fault pairs. The heuristic algorithm is shown below.

```
Heuristic Test Point Selection
While there are columns in the table
        Sort rows in decreasing order of the number of X's
        Choose the row at the top of the sorted list
        Remove columns having X's in the selected row
Endwhile
```

Since the algorithm is based on a heuristic, it can be shown that it does not produce the minimum number of test points for all cases. However, experimental results reported in Sec. 5 show that the algorithm produces results that are very close to results produced by an exact algorithm — however, the heuristic algorithm has a polynomial complexity and runs extremely (orders of magnitude) fast compared to an exact algorithm. Other heuristic algorithms for this covering problem [Cormen 89] can be used to further enhance the quality of the solution i.e., to minimize the number of test points.

5. Experimental Results

In Tables 5.1(a) and 5.1(b), we show some results on the number of test points needed to make all the fault pairs self-testable in a duplex system. For generating *different* designs, we minimized the truth tables corresponding to some MCNC benchmark circuits (*clip*, *inc*, *Z5xp1* and *rd84*) using *espresso*. Then, we synthesized logic circuits after applying multi-level optimizations using the *rugged* script available in *sis* [Sentovich 92]. We subsequently mapped the multi-level logic circuits to the LSI Logic G-10p technology library [LSI 96]. These implementations are referred to as "T" in Table 5.1(a). Next, we complemented the outputs in the truth tables of the benchmark circuits to generate new truth tables. We used the same synthesis procedure for these new truth tables. Finally, we added inverters at the outputs of the new designs obtained. These implementations are referred to as "C" in Table 5.1(a). In the third column of Table 5.1(a), we show the total number of single stuck-at fault pairs (in millions) in a duplex system containing the two implementations under consideration. Column 4 presents the percentage of fault pairs that are not self-testable when no test points are used. In the next three columns we show the number of observation test points needed to detect all the single stuck-at fault pairs using different techniques. As mentioned in Sec. 3, the ATPG-based technique can find the minimum number of test points. The ATPG tool available in *Sis* is used for this purpose. However, the running time of the technique is extremely high for large designs as shown in Table 5.1(b). The entries marked with '—' are the cases where the exact technique ran for more than a day without producing any result. The next column shows the number of test points needed if we use only the MISR for signature analysis in Algorithm 1. We also show the number of test points that are inserted if we consider both the MISR and the counter during signature analysis in Algorithm 1. For solving the covering problem we used the heuristic algorithm presented in Sec. 4.2 because the exact algorithm could not find the solution after running for one day for most of the examples. Note that, it follows from our discussions in Sec. 3 that the number of control test points is twice the number of observation test points. In the columns of Table 5.1(b) show the execution time required for finding the non-self-testable fault pairs using Algorithm 1 and an ATPG-based approach. All the programs were executed on a Sun Ultra-Sparc-2 workstation. We used the ATPG tool available in *Sis* for this purpose.

Table 5.1(a). Test points for different duplex systems

Circuit Name	Copy	# pairs (Million)	% escapes (no test point)	# Observation Test Points ATPG-based	MISR	MISR & counter	% escapes (with test points)
Z5xp1	T, T	0.30	0.73	275	275	275	0
	C, C	0.37	0.65	305	305	305	0
	T, C	0.36	**0.02**	**9**	**12**	**9**	0
clip	T, T	0.49	0.58	349	349	349	0
	T, C	0.46	**0.02**	**13**	**33**	**13**	0
inc	T, T	0.24	0.79	243	243	243	0
	C, C	0.26	0.84	253	253	253	0
	T, C	0.25	**0.03**	**12**	**13**	**12**	0
apex4	T, T	93	0.1	—	4818	4818	0
	C, C	74	0.2	—	4289	4289	0
	T, C	83	**0.01**	**—**	**46**	**46**	0
rd84	T, T	0.32	0.74	284	284	284	0
	C, C	0.16	1.1	199	199	199	0
	T, C	0.23	**0.04**	**10**	**22**	**11**	0

Table 5.1(b). Execution time using different techniques

Circuit	# pairs (Million)	ATPG-Based	Algorithm 1
Z5xp1	0.36	2 min	**6 sec**
clip	0.46	4 min	**34 sec**
inc	0.25	1 min	**4 sec**
apex4	83	> 1 day	**85 min**
rd84	0.23	2 min	**6 sec**

The following observations can be made from the experimental data presented in Tables 5.1(a) and 5.1(b). It is overwhelmingly clear from Table 5.1(a) that by adding only a very few test points in a duplex system with different implementations, we can make all the fault pairs self-testable. The number of test points needed for duplex systems with different implementations is orders of magnitude lower than those needed for duplex systems with identical implementations.

For minimizing aliasing (and hence, reducing the number of test points to be added), we recommend using both the MISR and the counter during signature analysis in Algorithm 1. As shown in table 5.1(b), we obtain very good speedup of our algorithm using the signature-based approach (Algorithm 1) compared to the conventional ATPG-based approach.

6. Fault Equivalence Relationships And Test Point Insertion

In our test point insertion technique presented in Sec. 4, we did not consider the effects of equivalence relationships among the different faults. A fault is said to be functionally *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, for any AND gate, all single stuck-at-0 faults at the inputs and the output of the AND gate are equivalent. Similarly, for an OR gate, all single stuck-at-1 faults at the inputs and the output of the OR gate are equivalent. For an inverter, a stuck-at-0 (1) fault at the input of the inverter is equivalent to a stuck-at 1 (0) fault at its output. For more discussion on fault equivalence, the reader is referred to [McCluskey 71]. Fault equivalence relationships can be used to further reduce the number of test points to make all the fault pairs self-testable in a duplex system.

For example, consider the case of observation test point insertion. Let us suppose that a fault pair (f_1, f_2) is not self-testable in a duplex system. Let us suppose that fault f_1 is a stuck-at-0 fault at the input of an AND gate. Also suppose that fault f_3 is a stuck-at-0 fault at the output of the same AND gate. Since f_1 and f_3 are equivalent faults, (f_3, f_2) is also a non-self-testable fault pair. In this case, by inserting an observation test point on the output of the AND gate, we can detect the fault pair (f_1, f_2). However, inserting an observation test point at the input of the AND gate does not detect the stuck-at-0 fault at its output.

During control test point insertion, let us consider a similar scenario as before. The fault pair (f_1, f_2) is not self-testable in a duplex system. Fault f_1 is a stuck-at-0 fault at the input of an AND gate. Also suppose that fault f_3 is a stuck-at-0 fault at the output of the same AND gate. Since f_1 and f_3 are equivalent faults, (f_3, f_2) is also a non-self-testable fault pair. In this case, we can insert a control test point either at the input or at the output of the AND gate.

These observations provide further opportunities to minimize the number of test points that are required to make all the fault pairs self-testable in a duplex system.

7. Conclusions

In this paper, we have shown the usefulness of using diverse implementations in enhancing the self-testability of fault pairs in duplex systems. Our technique of finding the non-self-testable fault pairs in duplex systems show orders of magnitude improvement in run time compared to other competitive techniques. We have also described test point insertion techniques to increase the number of self-testable faults in duplex systems using test points and provided experimental results. The results show a distinct advantage of using duplex systems with different implementations so far as the number of test points is concerned. This enhancement allows us to detect the non-self-testable fault pairs and thereby increase the system availability. We are further investigating techniques to synthesize different implementations so that the number of self-testable fault pairs is maximized. The test point insertion techniques reported in this paper can be combined with other test point insertion techniques used in the context of digital testing [Touba 96] to reduce the test length and detect different fault pairs more efficiently.

8. Acknowledgments

This work was supported by Defense Advanced Research Project Agency (DARPA) under Contract No. DABT63-97-C-0024 (ROAR project).

9. References

[Abramovici 90] Abramovici, M., M. Breuer and A. Friedman, *Digital Systems Testing and Testable Design*, IEEE Press, 1990.

[Armstrong 72] Armstrong, D. B., “A Deductive Method of Simulating Faults in Logic Circuits,” *IEEE Trans. Computers*, Vol. C-21, No. 5, pp. 464-471, May, 1972.

[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. Conf.*, 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.

[Chameleon 99] *Chameleon Systems Inc.*, http://www.cmln.com

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

[Eichelberger 83] Eichelberger, E. B. and E. Lindbloom, “Random-Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test,” *IBM Journal of Res. and Dev.*, Vol. 27, No. 3, pp. 265-272, May 1983.

[FIPS 77] FIPS PUB 46, “Data Encryption Standard,” *FIPS Publication, US Department of Commerce/National Bureau of Standards*, National tech, Info. Service, Springfield, Virginia, 1977.

[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," *Proc. 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. On Computers*, Vol. C-20, No. 11, pp. 1286-1293, Nov. 1971.

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

[Mitra 99b] Mitra, S., N. Saxena and E. J. McCluskey, “Design Diversity for Redundant Systems,” *CRC TR 99-4*, Center for Reliable Computing, Stanford Univ., 1999.

[ROAR 99] The ROAR project, http://crc.stanford.edu /projects/roar/roarSummary.html

[Saxena 98] Saxena, N.R., and E.J. McCluskey, "Dependable Adaptive Computing Systems," *Proc. IEEE Systems, Man and Cybernetics Conf.*, San Diego, pp. 2172-2177, 1998.

[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.

[Touba 95] Touba, N.A., and E.J. McCluskey, "Synthesis of Mapping Logic for Generating Transformed Pseudo-Random Patterns for BIST," *Proc. Intl. Test Conf.*, pp. 674-682, 1995.

[Touba 96] Touba, N.A., and E.J. McCluskey, "Test Point Insertion Based on Path Tracing," *Proc. VLSI Test Symp.*, pp. 2-8, 1996.