SPaRe: Selective Partial Replication for Concurrent Fault Detection in FSMs

Petros Drineas * and Yiorgos Makris
Departments of Computer Science and Electrical Engineering
Yale University
{petros.drineas, yiorgos.makris}@yale.edu

Abstract

We propose a non-intrusive methodology for concurrent fault detection in FSMs. The proposed method is similar to duplication, wherein a replica of the circuit acts as a predictor that immediately detects potential faults by comparison to the original FSM. However, instead of duplicating the FSM, the proposed method selects a few prediction functions which only partially replicate it. Selection is guided by the objective of minimizing the incurred hardware overhead without compromising the ability to detect all faults, yet possibly introducing fault detection latency. Furthermore, in contrast to concurrent error detection approaches which presume the ability to re-synthesize the FSM and exploit parity-based state encoding, the proposed method does not interfere with the encoding and implementation of the original FSM. Experimental results indicate that the proposed method achieves significant hardware overhead reduction over duplication, while detecting more than 99% of all permanent faults with very low average fault detection latency.

1. Introduction

Concurrent test methods provide circuits with the ability to self-examine their operational health during normal functionality and indicate potential malfunctions. While such an indication is highly desirable, designing a concurrently self-testable circuit which, at the same time, conforms to the rest of the design specifications is not a trivial task. Issues to be addressed include the hardware cost and design effort incurred, potential performance degradation due to interaction between the circuit logic and the concurrent self-test logic, as well as the level of assurance required.

In this paper, we focus our interest on Finite State Machines (FSMs) and we explore the trade-offs between the aforementioned parameters, in order to devise a non-intrusive design method for concurrent fault detection. Non-intrusiveness implies that hardware may only be added in parallel to the given FSM which is encoded, optimized, and implemented to meet specific requirements and may not be modified. The additional logic is expected to detect all circuit faults. Moreover, self-test has to be performed concurrently with the operation of the FSM and may not degrade it.

In concurrent test, additional hardware is added to the circuit in order to monitor its inputs during normal operation and generate an *a priori* known property that is expected to hold for the circuit output. A property verifier is subsequently utilized to identify and indicate any violation of the expected property, thus detecting potential circuit malfunctions. An important requirement in concurrent test is that the normal operation of the circuit may not be interrupted by *false alarms*; in other words, the concurrent test output indicator of the property verifier may not be asserted unless a malfunction is detected in the circuit.

The simplest approach is to duplicate the circuit, thus imposing an identity property between the original circuit output and the replica output, which may be simply examined by a comparator operating as the property verifier. With the exception of common-mode failures, duplication will immediately detect any error in the circuit. However, it incurs significant hardware overhead that exceeds 100% of the original circuit, which may or may not be justifiable depending on the application that the circuit is intended for. While expensive schemes such as duplication detect all functional *errors*, simpler properties detecting only structural *faults* in a prescribed fault model exist. For example, the method proposed in [1] reduces the functionality of the duplicate so that it only predicts the output of the circuit for a set of test vectors adequate to detect all stuck-at faults. The latter, however, allows functional errors to go undetected until the structural fault that causes it is eventually detected. The concept of *fault detection latency*, the time difference between appearance of an error and detection of the causing fault is thus introduced.

Since electronic circuits are employed in a wide spectrum of applications, ranging from mission-critical to simple commodity devices, concurrent test methods of various cost and efficiency are required. Related work is reviewed in section 2 and SPaRe, a concurrent fault detection method based on selective partial replication is proposed in section 3. Experimental results in terms of hardware overhead, fault coverage, and fault detection latency are provided in section 4.

2. Related Work

Related research efforts in concurrent test can be roughly classified in one of the following two categories:

Concurrent Error Detection (CED): Approaches in this category require that all functional errors be detected with zero (or very small, bounded) latency. Duplication is the simplest CED method, limited however by its expensive hardware overhead. Reducing the area overhead below the cost of duplication typically requires redesign of the original circuit, thus leading to intrusive methodologies. One of the first successful attempts along this direction is described in [2], where resynthesis is employed to favorably encode the circuit, incorporating parity information and employing TSC checkers. Structural limitations requiring an inverter-free design were alleviated in [3], where a single parity bit and partitioning is employed. Multiple parity bits are used in [4]. While these methods are intrusive, they render totally self-checking circuits, guarantee zero latency, and typically provide hardware savings in the range of 15% over duplication.

*The author is supported in part through NSF grant CCR-9820850.

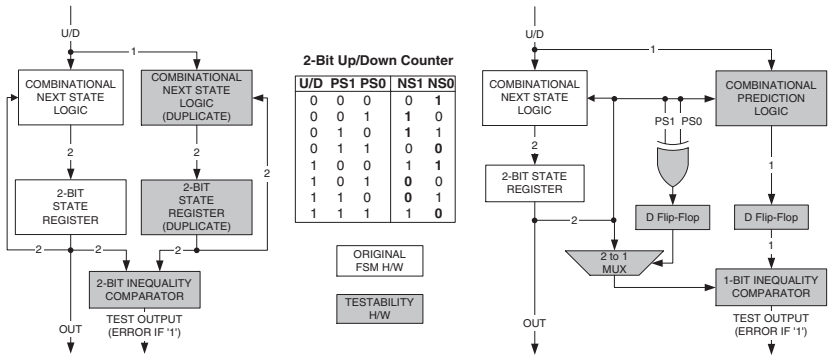

Figure 1. Duplication-based CED and SPaRe-based CFD on a 2-Bit Up/Down Counter

Concurrent Fault Detection (CFD): Approaches in this category require that for every structural fault in a prescribed fault model there exists at least one input combination that will detect the fault. Yet it is not guaranteed that every input combination that activates a fault will also detect it. While fault detection latency is, thus, introduced, this relaxation allows for significant hardware savings of CFD over CED methods. Among the few existing CFD schemes, properties specific to non-linear adaptive filters are used in [5], achieving a 30% cost reduction with near-zero latency. Frequency response of linear filters is used as an invariance property in [6], achieving a 50% cost reduction but introducing significant latency. Finally, a CFD approach exploiting transparency of RTL components is described in [7], achieving over 90% fault security with 40% hardware overhead.

3. Proposed Method

SPaRe, a CFD method based on Selective Partial Replication is proposed in this section. The key idea supporting SPaRe is presented through a small example, followed by an extensive description and analysis of the proposed method.

3.1. Motivation

Consider the 2-bit Up/Down Counter described in the table of figure 1. If the objective is to detect all *errors* occurring during normal operation, the duplication-based CED scheme shown on the left side of the figure will achieve this by comparing the two outputs of the FSM[1] to the two outputs of its replica. If, however, the objective is to detect all *faults*, allowing possible fault detection latency, it is not necessary to compare both FSM outputs at every clock cycle. When we implemented the counter we noticed that by observing only one bit per state transition (shown in boldface in the table of figure 1), we detect all faults. Therefore, for the purpose of CFD it is sufficient to replicate only partially the FSM, appropriately selecting which bits to predict for each state transition in order to detect all faults. Partial FSM replication implies cost reduction over duplication.

This observation is the basis for the SPaRe methodology which is shown on the right side of figure 1 for the 2-bit Up/Down Counter. A combinational prediction logic is used to implement the 1-bit function that generates for each state transition the value shown in boldface in the table of figure 1. This value is stored in a D Flip-Flop and compared to the corresponding bit of the FSM state register one clock cycle later. A MUX is used to drive the appropriate FSM output to the comparator. The select line of the MUX is driven by a function of the previous state and the inputs of the FSM, in this case a simple XOR between PS1 and PS0, delayed by one clock cycle. All faults in the next state logic are, thus, detected. Additionally, by postponing the comparison by one clock cycle, faults in the state register are also detected.

[1] For simplicity, we assume that FSM outputs are directly driven by the state register. SPaRe is readily extendible to include output logic.

3.2. SPaRe: Selective Partial Replication

The optimization objective of SPaRe is to minimize the output width of the prediction logic. Based on the observation that a subset of output bits per state transition is typically sufficient to detect all faults, SPaRe aims at identifying a minimal such set. The general version of SPaRe is depicted in figure 2. For every $(n + k)$-bit input combination, the prediction logic generates ℓ outputs that match a subset of ℓ out of the k FSM outputs. A Selection Logic is required to choose which FSM outputs to drive to the comparator for each $(n + k)$-bit input combination. Comparison is delayed by one clock cycle to also detect faults in the state register.

Success of SPaRe relies on efficient solutions to two key issues: identification of appropriate output values to be replicated by the prediction logic and cost-effective selection of circuit outputs to which they should be compared. Regarding the first issue, an ATPG tool capable of generating all test vectors and reporting both the good and faulty circuit outputs for every fault in the combinational next state logic is required. This information indicates the faults that can be detected at each output for each input vector and may be used to construct a matrix similar to the one shown in figure 3. SPaRe seeks a set of columns that covers all faults, such that the maximum number of output bits to be observed for any input vector is minimized. However, the exact selection of columns impacts directly the cost of the Selection Logic. More specifically, since the prediction logic only generates an ℓ-bit function, additional logic is necessary to select ℓ among the k circuit outputs to which the predicted ℓ bits will be compared. As shown in figure 2, this can be viewed as ℓ k-to-1 MUXes, each of which requires $\log k$ address bits. Therefore, if we allow any possible subset of size ℓ for every $(n+k)$-bit input combination, the Address Logic will generate $\ell \cdot \log k$ $(n+k)$-input functions. Compared to duplication,

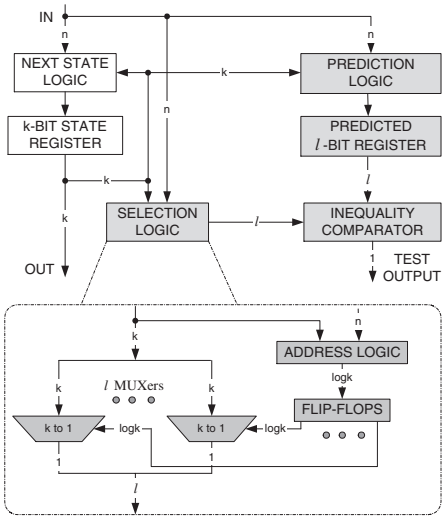

Figure 2. SPaRe: Selective Partial Replication

SPaRe implements $k-\ell$ fewer $(n+k)$-input functions for the Prediction Logic, at the cost of implementing $\ell \cdot \log k$ $(n+k)$-input functions and ℓ k-to-1 MUXes for the Selection logic. The cost of the Prediction Logic is linear in ℓ; the cost of the Selection Logic, however, increases almost linearly in ℓ up to $\ell = k/2$, at which point it starts decreasing, eventually becoming zero at $\ell = k$. Therefore, if $\ell > k/(\log k + 1)$, the total size of the Selection Logic and the Prediction Logic exceeds the cost of duplication.

Imposing such an upper bound on ℓ could significantly reduce the fault coverage of this scheme. Instead, we impose restrictions on the complexity of the Address Logic and by extension, on the acceptable solutions on the matrix of figure 3. SPaRe eliminates the Address Logic all together, therefore allowing that the $\log k$ select inputs of each multiplexer may only be driven directly by any $\log k$ out of the $(n+k)$ previous state and input bits. The form of acceptable solutions under this additional constraint, as well as a selection algorithm for identifying an appropriate set of columns that detects all faults are discussed in the following section.

3.3. Selection Algorithm

We focus on the next state logic of the FSM, which, given a previous state and an input generates the next state. The inputs to this component are $I_1 \dots I_k$ (the previous state) and $I_{k+1} \dots I_{n+k}$ (the FSM inputs). The outputs of this component are $O_1 \dots O_k$ (the next state). We denote the set of the 2^{n+k} possible previous state/input combinations by V.

Assume for the moment that we are given the matrix of figure 3, say A. We remind that SPaRe eliminates the ADDRESS LOGIC component of figure 2. For simplicity, we assume that 2 *specific* input bits, denoted by I_1 and I_2, drive *all* ℓ MUXes *and* also that ℓ is given. As a result, each MUX selects only among four of the FSM outputs; we remove this assumption promptly. Thus, the SELECTION LOGIC component of the diagram is fully specified. The SELECTION LOGIC splits the input vectors to 4 disjoint groups, each corresponding to a possible value for the pair $I_1 I_2 \in \{00, 01, 10, 11\}$; for all vectors in each group the *same* ℓ output bits are observed at the output of the SELECTION LOGIC. We denote the groups by G_1, G_2, G_3, G_4.

	VECTOR 0			...	VECTOR 2^{n+k}-1		
	OUT_0	...	OUT_{k-1}	...	OUT_0	...	OUT_{k-1}
$Fault_1$	1						
$Fault_2$					1		1
. . .	. . .	. . .	. . .	...	. . .	. . .	. . .
$Fault_m$			1	...	1		

Figure 3. Fault Detection Matrix

We now state the problem formally: given A, the groups G_1, G_2, G_3, G_4 and ℓ, pick ℓ output bits for each group so that the number of covered faults is maximized.

Prior to presenting an algorithm to solve the above problem, we revoke the simplifications we made earlier, starting with the assumption that ℓ is given. In practice, we seek the minimum ℓ for which we can detect i.e. 99% of the faults. Finding such an ℓ though is trivial; since $1 \leq \ell \leq k$, use binary search and solve the above problem $\log k$ times. We also assumed that the addressing bits (I_1 and I_2) were given; in practice we try all possible 2-bit addressing schemes ($\approx (n+k)^2/2$). If we were to use $c > 2$ bits to feed the MUXes, the number of possible addressing schemes increases; however, since we only allow up to $\log k$ addressing bits, it is always a small number. We note that in this case the number of groups would increase to 2^c instead of 4. Finally, we assumed that A is fully constructed; obviously, for large circuits, time/space constraints render this assumption infeasible. Thus, in large circuits, the following strategy is employed: for every fault, generate a large number (say r) of input vectors detecting it. Thus, assuming m faults in our circuit, at most mr vectors are generated. We subsequently identify the faults detected by each of these vectors, construct an $m \times mr$ matrix A' and solve the aforementioned problem in A' instead of A. Generally, A' admits less efficient solutions than A; as r increases the two solutions converge.

The size of the solution space for the above problem, assuming that ℓ and c are fixed, is $\binom{n+k}{c}\binom{k}{l}^{2^c}$. If ℓ and c are small constants, the size of the solution space is polynomial in both n and k. In practice, though, ℓ might be close to $k/2$, in which case the size of the solution space grows exponentially in k and it is impossible to explore it exhaustively. To understand its size, if $n = 2$, $k = 6$, $l = 3$ and $c = 2$ there are $4.5 \cdot 10^6$ possible solutions, while, if $c = 3$, there are more than $14 \cdot 10^{11}$ possibilities. Thus, we describe an algorithm to explore the space of possible solutions efficiently; given infinite time, our algorithm explores the whole state space. In practice, we explicitly limit its running time. We should also note that it is not necessary to drive all MUXes with the same input bits; indeed, better fault coverage might be achieved by using different bits. Thus, the state space is even larger, since the number of possible addressing schemes increases.

Our algorithm is simple: it randomly decides which ℓ output bits to generate for each group of input vectors; we denote by R_i the set of output bits that we generate for group G_i. Initially all the R_i's are empty. The algorithm essentially picks a group and decides which output bit to generate for this group; we decide which group to pick using biased sampling and favoring groups whose corresponding R_i contains fewer elements. Biased sampling is also used to decide which output bit to include in R_i. We assign a **score** to every output bit not already included in R_i: this score reflects the **significance** of this particular output bit for fault detection.

Intuitively, the **significance** of an output bit is a function of the number of faults it detects, and, in particular, faults that are not detected by a large number of vectors in V. As an example, we tend to favor an output bit that detects 2 faults that no other input vector can detect over an output bit that detects 5 faults, each detected by 10 other input vectors as well. Every time an output bit is selected to be included in R_i, we remove all faults covered by that bit for any input vector in G_i. The above process is repeated until all R_i contain exactly ℓ elements and the fault coverage is reported. If the result is unsatisfactory, we repeat the process until either a satisfactory result emerges or a fixed number T of iterations is exceeded; if the result is still unsatisfactory, we try a different addressing scheme. The **SPaRe** algorithm calls the **BasicSPaRe** algorithm with different G_1, G_2, G_3, G_4 until a target fault coverage is attained or the run time limit of the scheme is exceeded.

A brief note on x: while in our experiments a value of $x = 1$ returned acceptable solutions fast (typically, after trying at most 10 addressing schemes with $T = 100$), one could try different values of x to fine tune the algorithm. As an example, as x increases, our search becomes greedier: the output bit with the highest score is picked with very high probability. We prefer to present our algorithm using generic values for x; in practice, one could potentially use training data to learn the "best" value of x for the circuits at hand.

3.4. Analysis

SPaRe is non-intrusive and, by construction, guarantees a pre-specified fault coverage; in our case, 98.5%. Furthermore, since SPaRe predicts and compares the appropriate portion of the circuit output for every state transition, no false alarm is possible. SPaRe introduces latency in the detection of an activated fault, which will remain undetected until an appropriate state transition is performed. We stress, however, that SPaRe checks for faults for *every* state transition; since most stuck-at faults are detected by many state transitions, we may conjecture that the average latency of SPaRe is small. In Section 4 we see that this prediction is justified. We outline the expected hardware overhead of SPaRe. The following statement relates the hardware - assuming multilevel implementation using 2-input gates - required to implement a function of $n + k$ input bits and one output bit to the hardware required to implement a function of $n+k$ input bits and k output bits ($k \ll 2^{n+k}$).

Remark 1 *Almost all boolean functions* $f : \{0,1\}^{n+k} \rightarrow \{0,1\}^k$ *require at least* $k2^{n+k}/(n+k)$ *gates* if *the* k *output bits are uncorrelated.*

Proof (sketch): We observe that the number of functions $f : \{0,1\}^{n+k} \rightarrow \{0,1\}^k$ is $\left(2^{2^{n+k}}\right)^k = 2^{2^{(n+k)}k}$. Thus, Shannon's counting argument [8] proves our statement.

Assume for the moment that the k output bits are *uncorrelated*; then, the *minimum* hardware required for SPaRe is ℓ/k times the *minimum* hardware required for the original circuit. We can only examine how the *lower bound* of the size of SPaRe behaves; indeed, tight bounds for circuit sizes are notoriously hard to prove even under stringent assumptions. In practice, the output bits of the PREDICTION LOGIC and the original FSM are correlated, otherwise some states of our FSM would be unreachable. It is not clear though that as correlation increases the ratio of the size of SPaRe over the size of the original circuit increases; one expects the size of the SPaRe to decrease as correlation increases. In section 4 we observe that our predictions on the hardware overhead are quite accurate, even in the presence of correlation.

Algorithm BasicSPaRe

Input: $A, G_1, G_2, G_3, G_4, \ell$
Output: R_1, R_2, R_3, R_4, initially empty.

Preprocessing: Assign a score to each fault in A,

$$\mathbf{Score(F_j)} = \mathbf{nnz(A_{(j)})},\ \mathbf{j = 1 \ldots m}$$

$\mathbf{nnz(A_{(j)})}$ denotes the number of non-zero elements in the i-th row of A.

(a) Randomly pick one of the R_i, with probability

$$\Pr(\text{picking } R_i) = (\ell - |R_i|) / \sum_{i=1}^{4} (\ell - |R_i|)$$

Denote the one picked by R_i.

(b) Assign a score to each output bit $\mathbf{O_p} \notin \mathbf{R_i}, \mathbf{p} = \mathbf{1 \ldots k}$ ($x \in \mathcal{R}$, usually $x = 1$).

$$\mathbf{Score(O_p)} = \left(\sum_{\mathbf{F_j} \in \mathbf{S}} \mathbf{Score(F_j)}\right)^{\mathbf{x}}$$
$$S = \{\mathbf{F_j} : \mathbf{O_p} \text{ and any vector in } \mathbf{G_i} \text{ cover } \mathbf{F_j}\}$$

(c) Randomly pick one of the $\mathbf{O_p}$, with probability

$$\Pr(\text{picking } \mathbf{O_p}) = \frac{\mathbf{Score(O_p)}}{\sum_{\mathbf{p}:\mathbf{O_p} \notin \mathbf{R_i}} \mathbf{Score(O_p)}}$$

Denote the one picked by $\mathbf{O_p}$.

(d) $R_i = R_i \cup \{\mathbf{O_p}\}$

(e) Remove all faults (rows of A) covered by $\mathbf{O_p}$ and any vector in G_i.

(f) Repeat steps (a)-(e) until all the R_i's contain exactly ℓ elements and report the fault coverage.

(iter) Repeat steps (a)-(f) T times.

Algorithm SPaRe

Input: A, ℓ
Output: R_1, R_2, R_3, R_4 (initially empty).

(a) Create candidate G_1, G_2, G_3, G_4.

(b) BasicSPaRe($A, G_1, G_2, G_3, G_4, \ell$)

(c) Repeat (a)-(b) until the fault coverage is above target or the running time limit is exceeded.

FSM Type (States, Inputs)	Cost of Next State Logic	Predicted Output Bits	Cost of Prediction Logic	Hardware Overhead
(8, 1)	33408	2 / 3 (66 %)	28734	86.00 %
(8, 2)	70374	2 / 3 (66 %)	52846	75.09 %
(16, 1)	95275	2 / 4 (50 %)	52515	55.11 %
(16, 2)	186219	2 / 4 (50 %)	102294	54.93 %
(32, 1)	222411	2 / 5 (40 %)	100303	45.09 %
(32, 2)	423014	2 / 5 (40 %)	204086	48.24 %
(32, 3)	832571	2 / 5 (40 %)	369779	44.41 %
(64, 1)	504368	3 / 6 (50 %)	276888	54.89 %
(64, 2)	937744	3 / 6 (50 %)	522590	55.72 %
(64, 3)	1809757	3 / 6 (50 %)	1022400	56.49 %

Figure 4. Hardware Overhead Comparison

FSM Type	(8, 1)	(8, 2)	(16, 1)	(16, 2)	(32, 1)	(32, 2)	(32, 3)	(64, 1)	(64, 2)	(64, 3)
Testable Faults	53	123	143	286	362	649	1312	788	1433	2751
Detected by SPARE	53	122	141	283	358	640	1293	781	1418	2724
Coverage (%)	100.00	99.18	98.60	98.95	98.89	98.61	98.55	99.11	98.95	99.01

Figure 5. Fault Coverage on Original FSM Faults

FSM Type	(8, 1)	(8, 2)	(16, 1)	(16, 2)	(32, 1)	(32, 2)	(32, 3)	(64, 1)	(64, 2)	(64, 3)
Testable Faults	177	285	326	552	603	1074	1964	1340	2384	4496
Detected by SPARE	177	284	324	549	599	1065	1945	1333	2369	4469
Coverage (%)	100.00	99.64	99.38	99.45	99.33	99.16	99.03	99.47	99.37	99.39

Figure 6. Fault Coverage on All FSM Faults

4. Experimental Results

In this section, we compare SPaRe to duplication, in terms of hardware overhead, fault coverage, and fault detection latency. In order to preserve generality, we employ random FSMs with $K = 2^k$ states and n inputs. We experiment with ten different types of (K, n) FSMs, where K is the number of states and n is the number of inputs. The ten types are $(8, 1)$, $(8, 2)$, $(16, 1)$, $(16, 2)$, $(32, 1)$, $(32, 2)$, $(32, 3)$, $(64, 1)$, $(64, 2)$, and $(64, 3)$. Average results over five FSMs of each type are reported.

4.1. Hardware Overhead

In terms of incurred hardware overhead, SPaRe and duplication differ in the following aspects: duplication employs a replica of the combinational next state logic of the original FSM, while SPaRe employs a prediction logic which generates fewer output bits. As a result, SPaRe uses a narrower state register and a narrower comparator than duplication. However, a few additional MUXes are employed in SPaRe, balancing the cost savings of these modules. Essentially, in order to compare SPaRe to duplication, it is adequate to compare the cost of the next state logic of the FSM to the cost of the prediction logic of SPaRe.

In order to obtain these costs, the next state function of the FSMs generated through the above process is converted to *pla* format, synthesized using the *rugged* script of SIS [9], and mapped to a standard cell library comprising only 2-input gates. Since the proposed methodology is non-intrusive, no assumptions are made as to how the FSMs are encoded or optimized. The hardware cost of the circuit is reported by SIS through the *print_map_stats* command and the circuit is then converted to ISCAS89 [10] format. ATALANTA [11] is used to generate all vectors detecting each fault, and HOPE [12] is employed to provide both the good machine and the bad machine responses for every *(vector, fault)* pair, revealing the output bits at which each fault may be detected for every vector. This information is used to construct the matrix A necessary for SPaRe, through which the prediction logic functions are identified. These functions are subsequently converted to *pla* format, synthesized using the *rugged* script of SIS [9], and mapped to a standard cell library comprising only 2-input gates. The cost of the prediction logic is reported by SIS [9] through the *print_map_stats* command and the circuit is converted to ISCAS89 [10] format.

The results are summarized in the table of figure 4. The cost of the next state logic is reported, along with the number ℓ of prediction logic bits generated through the algorithm of section 3.3. The percentage in the parenthesis indicates the expected hardware overhead, based on the analysis of section 3.4. Finally, the cost of the prediction logic is reported. The rightmost column provides this cost as a percentage of the cost of the next state logic, indicating the hardware savings of SPaRe over duplication. As may be observed, the hardware overhead is, on average, 45% less than duplication. Furthermore, the average deviation between the expected overhead and the actual overhead is around 7%, implying that the ratio of predicted bits over next state logic bits is an accurate indication of incurred hardware overhead. We anticipate that this ratio will decrease further as the number of next state logic bits increases, thus resulting in even more savings.

4.2. Fault Coverage

In order to assess the effectiveness of the proposed method, we construct the FSM with SPaRe-based CFD and the FSM with duplication-based CED in ISCAS89 [10] format. The next state logic and the prediction logic are available from the hardware overhead experiment. Two copies of the next state logic, two state registers and a comparator are used for duplication. One copy of the next state logic, one copy of the prediction logic, a comparator and the MUXes for the selection logic are used for SPaRe.

Two experiments are performed employing these circuits. In the first experiment, we compare the number of faults in the *original* FSM detectable by SPaRe to those detectable by duplication. HITEC [13] is used to perform ATPG on the two constructed FSMs. In both ATPG runs only the faults in the original FSM are targeted and only the Test Output is made observable. The results are summarized in the table of figure 5. Duplication detects all testable faults in the original FSM, reported in the second row of the table. SPaRe, on the other hand, detects all faults that are covered in the solution provided by the algorithm of section 3.3. In our experiments, the threshold for algorithm termination was set to covering 98.5% of all faults. This is validated by ATPG, which yields an average fault coverage of 99% of all testable faults.

In the second experiment, we demonstrate the ability of SPaRe to also detect all testable faults in the hardware added for CFD. Two ATPG runs are performed using HITEC [13] on the FSM with SPaRe-based CFD, targeting *all* circuit faults. Both the test output and the original FSM outputs are made observable in the first ATPG run, while only the test output is made observable in the second ATPG run. The results are summarized in the table of figure 6. The number of faults missed by SPaRe in the tables of figures 5 and 6 is equal, indicating that all testable faults in the additional hardware are detected. On average, SPaRe-based CFD detects 99.4% of all testable faults.

4.3. Fault Detection Latency

The hardware savings achieved by SPaRe come at the cost of introducing fault detection latency. It is not possible to predict the exact latency of the method, since it depends on the values that appear at the FSM inputs during normal operation. Yet, an experimental indication of how much latency is introduced by SPaRe is necessary for its evaluation.

We measure fault detection latency based on fault simulation of randomly generated input sequences. More specifically, we use HOPE [12] to perform *two* fault simulations of the *same* sequence of randomly generated inputs, once observing both the test output and the FSM outputs, and a second time observing only the test output. The time step at which a fault is detected during the first fault simulation is the *Fault Activation* time, while the time step at which it is detected during the second fault simulation is the *Fault Detection* time. *Fault Detection Latency* is the time difference between Fault Activation and Fault Detection, therefore we can calculate the Fault Detection Latency for each fault, as well as the average Fault Detection Latency.

Worst-case results for each of the 10 different FSM types are summarized in the table of figure 7. We fault simulate a total of 5000 random patterns and snapshots of the results are shown after 10, 50, 100, 500, 1000, and finally all 5000 patters are applied. For each snapshot, we provide the number of faults *remaining* non-activated, the number of faults activated and *detected*, and the number of faults activated but *missed* (not yet detected) by SPaRe. We also provide the *maximum* and the *average* fault detection latency for the faults that are both activated and detected. Figure 8 presents a plot of faults activated and faults detected by SPaRe on the $(64, 3)$ FSM, as well as a plot of the average fault detection latency on the $(64, 1)$, $(64, 2)$, and $(64, 3)$ FSMs versus the number of applied random patterns.

While the maximum latency is significant, ranging up to 2714 clock cycles for the (64,3) circuit, the average latency is small, ranging up to only 28.35 clock cycles, which is 1.05% of the maximum latency. Additionally, most faults are detected quickly and the typical 90-10 rule applies for the average latency. More specifically, 90% of the faults are detected within 50% of the average latency, while the other 50% is contributed by the remaining 10% of the faults. For example, once 500 random vectors are applied to the (64,3) circuit, 96.69% of the faults are activated and 93.67% are detected. The average fault detection latency at this point is 11.66, which is 41.12% of the average latency when all faults are detected. Furthermore, the plot of Figure 8(a) shows that the number of faults activated but not yet detected by SPaRe is constantly small. Finally, as indicated in the plot of Figure 8(b), both the average and the maximum latency increase sub-linearly with the size of the circuit, guaranteeing scaling of SPaRe. Similar observations hold for all other circuits.

5. Conclusions

Cost-effective CFD requires careful examination of the trade-offs between the conflicting objectives of low hardware overhead, low fault detection latency, and high fault coverage. SPaRe explores the trade-off between fault detection latency and hardware overhead, under the additional constraint that the original circuit design may not be altered. Thus, a comparison-based approach is employed, where the next state logic of the original FSM is partially replicated into a prediction logic, selectively testing the circuit during normal operation. The problem of identifying cost-effective prediction logic functions is theoretically formulated and an algorithm for efficient selective partial replication is proposed. Experimental results demonstrate that SPaRe reduces the incurred hardware overhead by an average of 45% over duplication, while preserving the ability to detect more than 99% of the circuit permanent faults. Further reduction of this overhead is anticipated as the size of the circuit increases. While these savings come at the cost of introducing fault detection latency, the experimentally observed average latency is very low, ranging up to 28 clock cycles in the largest of our FSMs and scaling sub-linearly with the size of the circuit. In conclusion, when non-zero fault detection latency may be tolerated, SPaRe constitutes a powerful alternative to duplication.

References

[1] R. Sharma and K. K. Saluja, "An implementation and analysis of a concurrent built-in self-test technique," in *Fault Tolerant Computing Symposium*, 1988, pp. 164–169.

[2] N. K. Jha and S.-J. Wang, "Design and synthesis of self-checking VLSI circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 12, no. 6, pp. 878–887, 1993.

[3] N. A. Touba and E. J. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 16, no. 7, pp. 783–789, 1997.

[4] C. Zeng, N. Saxena, and E. J. McCluskey, "Finite state machine synthesis with concurrent error detection," in *International Test Conference*, 1999, pp. 672–679.

[5] A. Chatterjee and R. K. Roy, "Concurrent error detection in non-linear digital circuits with applications to adaptive filters," in *International Conference on Computer Design*, 1993, pp. 606–609.

[6] I. Bayraktaroglu and A. Orailoglu, "Low-cost on-line test for digital filters," in *VLSI Test Symposium*, 1999, pp. 446–451.

[7] Y. Makris, I. Bayraktaroglu, and A. Orailoglu, "Invariance-based on-line test for RTL controller-datapath circuits," in *VLSI Test Symposium*, 2000, pp. 459–464.

[8] C. E. Shannon, "The synthesis of two-terminal switching circuits," *Bell System Technical Journal*, vol. 28, pp. 59–98, 1949.

[9] E. M. Sentovich, K. J. Singh, L. Lavagno, C. Moon, R. Murgai, A. Saldahna, H. Savoj, P. R. Stephan, R. K. Brayton, and A. Sangiovanni-Vincentelli, "SIS: a system for sequential circuit synthesis," ERL MEMO. No. UCB/ERL M92/41, EECS UC Berkeley CA 94720, 1992.

[10] "ISCAS'89 benchmark circuits information," Available from `http://www.cbl.ncsu.edu`.

[11] "ATALANTA combinational test generation tool," Available from `http://www.ee.vt.edu/ha/cadtools`.

[12] H. K. Lee and D. S. Ha, "HOPE: An efficient parallel fault simulator for synchronous sequential circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, pp. 1048–1058, 1996.

[13] T. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," in *European Conference on Design Automation*, 1992, pp. 214–218.

FSM TYPE	TESTABLE FAULTS	STATISTIC	RANDOM 10	RANDOM 50	RANDOM 100	RANDOM 500	RANDOM 1000	RANDOM 5000
(8, 1)	53	REMAINING	8	0				
		DETECTED	44	53				
		MISSED	1	0				
		MAX LAT	3	26				
		AVG LAT	0.95	1.64				
(8, 2)	123	REMAINING	68	3	1	0		
		DETECTED	50	117	120	122		
		MISSED	5	2	2	1		
		MAX LAT	1	34	54	180		
		AVG LAT	0.92	2.11	3.09	3.54		
(16, 1)	143	REMAINING	56	5	3	0		
		DETECTED	72	131	136	140		
		MISSED	15	7	4	3		
		MAX LAT	8	36	58	68		
		AVG LAT	1.34	2.23	2.98	3.25		
(16, 2)	286	REMAINING	118	14	4	0		
		DETECTED	145	259	273	282		
		MISSED	23	13	9	4		
		MAX LAT	7	39	65	264		
		AVG LAT	1.18	2.25	3.21	6.73		
(32, 1)	362	REMAINING	151	39	6	1	0	0
		DETECTED	180	294	330	351	354	357
		MISSED	31	29	26	10	8	5
		MAX LAT	5	42	78	296	502	924
		AVG LAT	1.10	2.80	4.26	9.94	12.61	13.01
(32, 2)	649	REMAINING	360	105	42	1	0	0
		DETECTED	217	461	551	628	631	633
		MISSED	132	83	56	20	18	16
		MAX LAT	7	45	82	409	677	1012
		AVG LAT	1.17	3.82	6.64	17.79	19.66	21.32
(32, 3)	1312	REMAINING	913	387	187	25	5	0
		DETECTED	290	767	998	1217	1260	1281
		MISSED	109	158	127	70	47	31
		MAX LAT	8	47	94	478	825	1538
		AVG LAT	1.94	4.19	7.83	20.76	26.87	28.77
(64, 1)	788	REMAINING	451	89	54	3	1	0
		DETECTED	255	640	689	764	770	776
		MISSED	82	59	55	21	17	12
		MAX LAT	8	45	73	428	857	1093
		AVG LAT	1.34	3.39	4.49	10.15	12.83	14.47
(64, 2)	1433	REMAINING	999	365	63	6	2	0
		DETECTED	382	972	1184	1394	1403	1408
		MISSED	52	96	86	33	28	25
		MAX LAT	6	39	80	377	804	1798
		AVG LAT	1.10	2.22	3.97	11.31	14.20	17.55
(64, 3)	2751	REMAINING	1970	1081	640	91	25	0
		DETECTED	669	1525	1983	2577	2680	2720
		MISSED	112	145	128	83	46	31
		MAX LAT	8	46	94	466	947	2714
		AVG LAT	1.32	2.16	4.02	11.66	20.51	28.35

Figure 7. Fault Detection Latency of SPaRe-based CFD

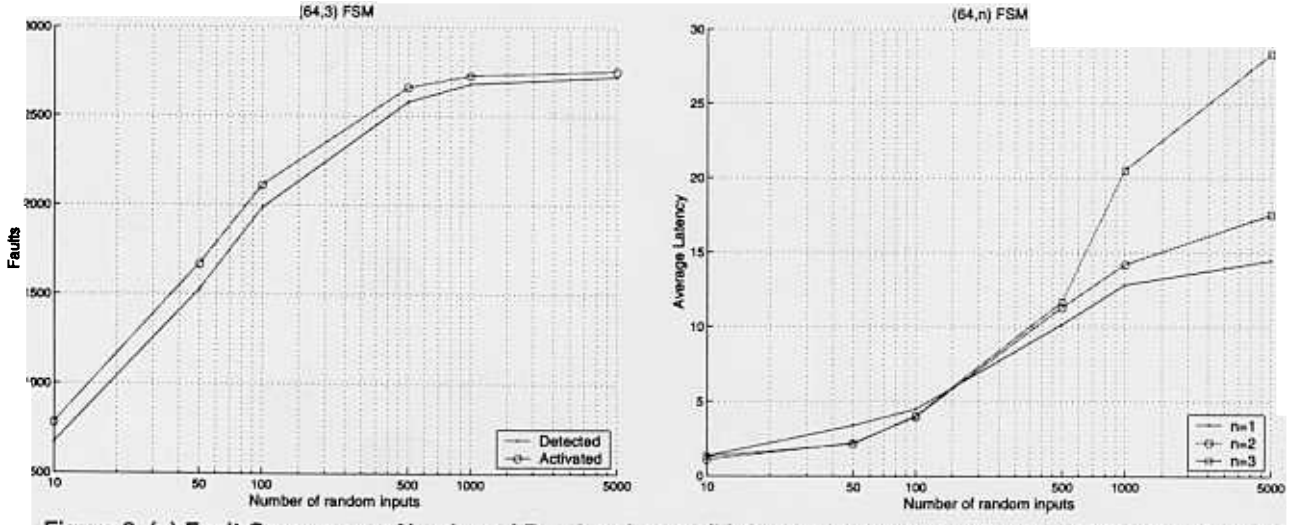

Figure 8. (a) Fault Coverage vs Number of Random Inputs (b) Average Latency vs Number of Random Inputs

Proceedings of the 16th International Conference on VLSI Design (VLSI'03)
1063-9667/03 $17.00 © 2003 IEEE