Using Implications for Online Error Detection

K. Nepal[†], N. Alves[∗], J. Dworak[∗], R. I. Bahar[∗]

[†]Electrical Engineering Department, Bucknell University, Lewisburg, PA 17837
[∗]Division of Engineering, Brown University, Providence, RI 02912

Abstract

In this paper, we investigate the use of logic implications for the online detection of intermittent faults and hard-to-detect manufacturing defects. We present techniques to efficiently identify the most powerful circuit implications that can be checked for violations so that the fraction of errors detected can be maximized while minimizing the additional hardware overhead. Importantly, our approach does not require re-synthesis of the targeted logic; the checker logic is added off the critical path and is run in parallel with the regular control logic. Trade-offs can be easily made between additional coverage of errors and additional area overhead. Our results show that significant error detection is possible—even with only a 10% area overhead.

1. Introduction

For years to come, the International Technology Roadmap for Semiconductors will continue to pursue aggressive size scaling. However as CMOS scales, circuit reliability degrades due to a host of issues—including defects, process variations, single event upsets, and operational and environmental influences (such as supply voltage fluctuations, cross-coupling, and thermal noise). Ensuring reliable computation requires mechanisms to detect and correct errors during normal circuit operation.

The strategies employed for detecting errors vary. For example, in the case of memory that may be corrupted due to high-energy particle strikes, appropriate parity or Hamming code check bits are often sufficient to detect and potentially even correct errors. However, detecting errors in memory elements is inherently easier than detecting errors in logic because verifying memory only requires that we check that the data we read is the same as the data that was stored. Thus, we know what the answer should be *a priori*. In contrast, in the case of logic, we must determine that the functional transformation of the inputs to the outputs has occurred correctly. This is obviously much more difficult because it is the very logic that is being checked that we are using to calculate the answer. We therefore don't know the correct results *a priori*, and thus, more complicated or highly expensive schemes often must be used.
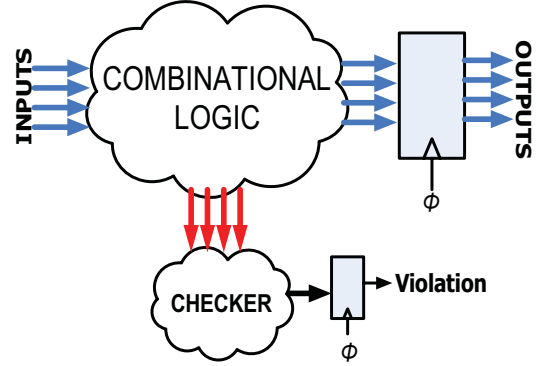

Figure 1. Circuit with implication checker.

Many previous approaches have introduced some sort of redundancy to provide for detection of errors in logic. This redundancy could include multiple executions of an input sequence at different times or in different copies of the hardware. Other techniques use encoding to predict some checkable property of the outputs. High-level functional assertions such as those identified during functional verification, may also be hardcoded into the design and used to signal the presence of an error (e.g., [1], [10]).

In this work, we present a novel approach to online error detection that *automatically* identifies *gate-level* invariant relationships (or assertions) that must be satisfied for the circuit to operating correctly. No knowledge of high-level behavioral constraints are required to identify these invariant relationships. Violations of these expected relationships can then be used to efficiently identify errors. Our approach is graphically represented in Figure 1. While the outputs of the combinational logic are being computed and then latched, the invariant relationships within the circuit are checked in parallel and latched. Any violations of these invariants will be detected at the output of the latches.

The invariant relationships we investigate in this work are logic value implications that exist between pairs of circuit sites. Consider the circuit shown in Figure 2. Due to logical constraints in the circuit, there are several implications that exist between various nodes in the circuit (see right side of the figure for a partial list). For example, it can be verified that whenever node $N4 = 1$, node $N24 = 0$ to retain correct logical
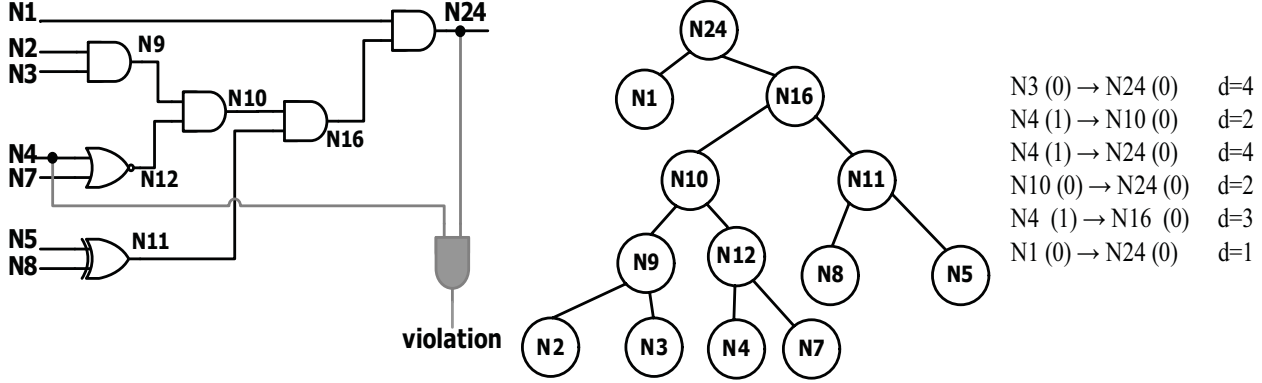
N1 N24
N2 N9
N3
N10
N16
N4
N7 N12
N5 N11
N8

violation

N24
N1 N16
N10 N11
N9 N12 N8 N5
N2 N3 N4 N7

N3 (0) → N24 (0)   d=4
N4 (1) → N10 (0)   d=2
N4 (1) → N24 (0)   d=4
N10 (0) → N24 (0)   d=2
N4 (1) → N16 (0)   d=3
N1 (0) → N24 (0)   d=1

Figure 2. A combinational circuit, its graphical representation, and list of sample implications.

consistency. If this relationship does not hold, an error must have occurred in the intervening logic between the two sites or at the second site. The logic shown in gray can be easily added to the circuit to check for violation of this implication. For instance, if node $N10$ becomes stuck-at-1 during online operation, the added checker logic would flag an error on any input vector where $N4 = 1$ and the value at node $N10$ propagates to $N24$ (i.e., is observable at $N24$).

While the circuit shown in Figure 2 is simple, such implications naturally exist in larger circuits as well. If valuable implications can be identified within the circuit and verified online using checker logic, then this becomes a very powerful tool for detecting errors during run time. However, including all possible implications in the checker logic is generally too costly in terms of area overhead, so some means of selecting the most valuable implications needs to be employed.

Our focus is on applications where perfect operation is desirable but not essential. As a result, our goal is not to eliminate the possibility of any error escaping detection, but to cost-effectively reduce the chance of that escape occurring. Our approach consists of two parts. First we present an efficient technique for identifying implications between sites that are at least two gates apart based on a combination of circuit simulation to identify potential implications followed by verification with a SAT solver. Next, we present a heuristic method for selecting an optimal subset of these implications to be included in checker logic. Our results show that using implication information can be a very effective technique for detecting errors at run time. We demonstrate that significant error detection can be achieved for several test circuits—even with area overheads of only 10%. Trade-offs can easily be made between additional overhead and additional error detection.

The rest of the paper is organized as follows. Section 2 discusses related work in online error detection and points out the unique features of our approach. In Section 3 we present our algorithms for identifying useful implications and then selecting a subset of these implications given a user specified area budget. We evaluate our proposed method in Section 4 on a set of MCNC and ISCAS benchmarks and discuss the significance of the results. Finally, in Section 6 we conclude the paper and discuss possible future extensions to our work.

2. Related Work

Online detection and correction of errors is a well-researched field. As previously noted, there is a significant difference between the detection of errors in memory (where we simply need to ensure that the value read from memory is the same as the value that was previously written) and the detection of errors in the functional logic of a circuit. In the former case, efficient protection from soft errors and single event upsets can be obtained through error detecting and correcting codes such as Hamming codes. Once identified, permanent errors can be also be avoided through the use of spare rows and columns [12, 7].

In the case of online detection of errors in the logic, alternative methods must be considered. In general, all of these methods involve the use of some type of redundancy in time or space. For example, in the case of soft or transient errors, redundancy in time may involve the execution of the code in multiple threads on a single piece of hardware so that the end results can be compared [27, 28, 34, 4]. More recently, Gomaa and Vijaykumar have presented a method involving instruction-level redundancy that duplicates instructions when resources are available [19].

Alternatively, redundancy in space can be achieved by simple duplication of the entire design or part of the design. The results of the duplicated logic can then be compared. For example, the IBM S/390 processor duplicated the instruction and execution units so that their results could be compared on every clock cycle [39]. In some cases, duplication with a complementary form of the hardware has been proposed [30]. If error correction is to be obtained, triple modular redundancy (TMR) can be used instead [38]. Obviously, fully duplicating or triplicating the circuit hardware is very expensive in terms of power and area. Thus, other methods (some of which do not provide full error

coverage) have been proposed. For example, selective triple modular redundancy was proposed for SRAM-based FPGAs in [31]. In this case, SEU-sensitive subcircuits were identified and only these subcircuits were protected with TMR. Similarly, the approach presented in [26] tried to decrease the area overhead required for the error detection circuitry by only duplicating parts of the circuit that were determined to be most SEU-susceptible.

Others have used a variety of coding techniques, such as parity prediction, to identify expected properties of the outputs. Two distinct approaches have been proposed for the design of circuits with parity bits. The first approach synthesizes the functional logic and then an appropriate low complexity parity-check code that ensures good coverage is implemented [25, 33, 20, 2]. The second approach is to select the parity-check code *a priori* and then synthesize the minimal functional logic with structural constraints to ensure high coverage [17, 14, 36]. Other works have focused on the use of unidirectional codes such as Berger [6] or Bose-Lin codes [9, 13]. Several different coding schemes were compared in [24]. While the use of these codes can be very effective for detecting errors in logic circuitry, the area overhead can be quite significant (sometimes more than doubling the size of the circuit). In addition, they may require that the original circuit be altered in order to generate the codes.

Other online error detection schemes include Built-In Concurrent Self Test (BICST) [32] and Reduced Observation Width Replication (ROWR) [16]. For BICST, hardware is added to predict the appropriate response to a set of pre-computed test vectors. The ROWR scheme is similar to BICST, but aims to reduce the hardware overhead by only checking the minimum required output values. While these methods will generally provide for *eventual* detection of all targeted faults within the test set, they do not guarantee the detection of all transient faults or guarantee immediate detection of a targeted fault on its first appearance.

We note that the assertions we are targeting in this work are different from previous works that use high-level assertions that were first generated during functional verification, and later integrated into hardware checkers to enhance post-fabrication fault detection (see, for example, [15], [1], [10]). While potentially very useful for detecting errors, a limitation of this approach is that identification of such assertions is dependent upon an understanding of the functional intent of the circuit made by the designer. In addition, the scope of these assertions for detecting errors is limited.

In this paper, we present a novel approach for the online detection of errors in logic circuits based on logic implications [11]. Logic implications have been used widely in logic synthesis for such purposes as area/delay/power optimization [22, 5, 29], peak current estimation [8], false noise analysis [18], and efficient ATPG [35]. Implication identification is also strongly related to structural analysis using observability don't care information. Previous works have proposed using this information to guide localized circuit restructuring
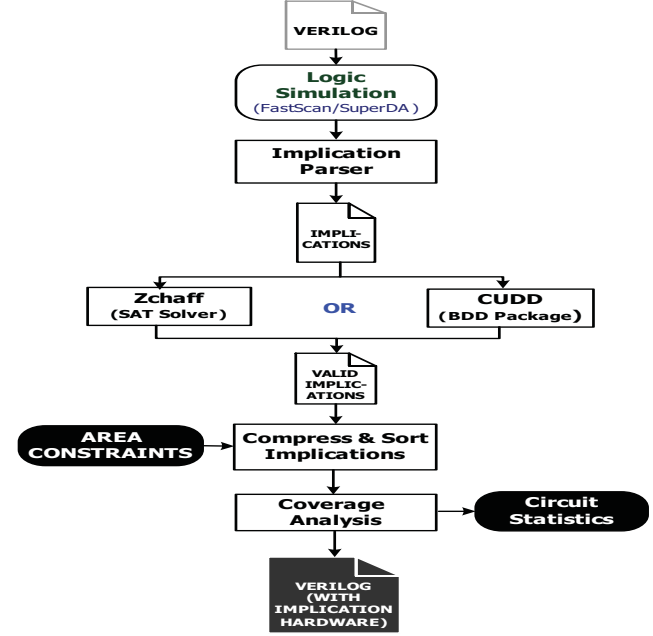

Figure 3. Implication generation/addition flow

such that errors will more likely be logically masked and thereby not sensitized at a primary output [3, 21]. While these are promising approaches for reducing the soft error rates in circuits, they require modification of the original circuit that could potentially adversely affect delay on critical paths.

In contrast, in this work we will incorporation checker logic that operates on the *original* synthesized version of the circuit—allowing the checker logic to be added off of the critical path and not requiring resynthesis of the circuit being protected. Our work is novel in that it will *automatically* identify invariant relationships (assertions) at the *gate level* through the identification of logic implications. These implications will be incorporated into checker logic that will flag an error when an implication is violated so that the system can take appropriate action (e.g., re-executing the failing input sequence).[1] Our scheme will also allow explicit trade-offs to be easily made between error coverage and acceptable hardware overhead.

3. Algorithm

As mentioned earlier, a typical circuit may contain thousands of valid implications. Finding all these implications can become a daunting and time consuming task. In addition, it is generally not cost-effective to

---

[1]Note: If the additional checker hardware itself contains a fault (static or transient), it is possible that errors may be erroneously reported by the hardware. This is a general limitation of parity checking schemes as well, such as [32, 16, 25, 2].

include all of them in the checker hardware because of the high area overhead required. Often, we would be no better off than simply duplicating the circuit and checking for differences in the outputs. To avoid this problem, our approach intelligently searches for implications and selects those that are "most useful" for detecting faults in the circuit. In the simplest case, most useful can be defined as those implications that detect the most faults under most input conditions.

The basic flow of our approach is shown in Figure 3 and includes the following steps:

1. Run logic simulation to quickly identify *potential* implications.

2. Check all potential implications for validity.

3. Eliminate implications subsumed by others.

4. Determine the quality of all valid implications.

5. Select a subset of implications such that the lowest probability of an undetected error is obtained given a user-specified hardware budget.

The first 3 steps are part of the "implication discovery" process, while the last 2 intelligently select a subset of these implications to include in the checker hardware. The remainder of this section describes these steps in more detail.

3.1. Discovering Implications

Implications can reside anywhere in the circuit. Past approaches have focused on starting at a specific node and using a recursive learning process to discover implications [22]. While this does have the potential advantage of targeting the search to a specific node, it may be quite time consuming because every node must be searched independently. Instead, we take a parallel approach and narrow our search for implications by running circuit simulation, using SuperDA [37], with random vectors as inputs. SuperDA implements event-driven parallel pattern simulation and is capable of simulating 32 input vectors at a time.

For each input vector applied to the circuit, we record the logic values for all nodes in the circuit. Once the simulation is complete, we can identify potential implications by checking to see if certain node pair values are absent in the simulation runs. For instance, if the implication $(a = 1) \implies (b = 1)$ exists between nodes a and b in a circuit, then for every input vector applied, there should never be an instance where $a = 1$ and $b = 0$. This comparison step turns out to be very fast because we can check 32 logic values (the size of an unsigned integer) for each node in parallel by doing bit vector compares. This step ignores simplistic implications across a single gate.

Note that these are only *potential* implications. Since we are running test vectors on only a sample of all possible 2^n input vectors, where n in the number of inputs to the circuit, we may be missing some

input vector that would show the potential implication to be invalid. Therefore, once we have generated a list of potential implications, we need some more formal method to verify that these implications are indeed valid. There are two approaches we can consider. The first one involves construction of the BDD for the nodes in question. The BDD approach can be attractive because generating the BDD will also generate other information that is useful for determining error coverage. However, this approach does not scale for larger circuits, where the number of inputs can be quite large. Instead, we consider using a SAT solver (such as ZChaff [23]) to check for the presence of an instance that would violate an implication. If no such instance exists, then the implication is deemed valid. In our case, using the ZChaff SAT solver to verify the implications required that we first convert our circuit description from Verilog format to conjunctive normal form (CNF). Each implication could then be verified separately by adding it into the CNF circuit description.

Using again the implication $(a = 1) \implies (b = 1)$ as an example, the SAT solver would verify if, for the given network, the condition $a = 1$ and $b = 0$ can be excited. If the SAT solver fails, then the implication is valid. If the SAT solver succeeds, then it returns an input vector that generates this condition. Since this implication check can be posed as a straightforward problem for the SAT solver, we can process all potential implication serially very quickly, even for large circuits, and in the end generate a list of all true implications for the circuit.

Once we have our preliminary list of validated implications, it is possible that some implications may be entirely contained within other implications in terms of their fault detection capabilities. In this case, it would not be useful in terms of improving fault coverage to include those implications that are subsumed by others in our checker logic. For example, consider again the circuit in Figure 2. Of the six implications shown in the figure, implications 2–5 are all in the path from node $N4$ to $N24$. Furthermore, there are no fanout branches off of this path to other portions of the circuit. Of these implications, the third one in the list $(N4 = 1 \implies N24 = 0)$ has the longest distance. This means that all nodes that have $N4$ as a parent node and $N24$ as a child node and that lead to the implication that $N24 = 0$ can be dropped from the implications list since they are already covered by a "higher-order" implication.

In our synthesis flow, we eliminate these subsumed implications by a combination of distance and node coverage analysis. During implication generation, we calculate the distances between the nodes that make up the implication pair by using a graph representation of the circuit. As an example, Figure 2 includes the graph for the combinational circuit described in Section 1. Applying this quick preprocessing step immediately after we verify the validity of the implication helps reduce the number of implications that need to be used for implication quality analysis.

```
Procedure IMPLICATION_QUALITY(implication, node, fault_type)
    fault_undetected = 0;
    foreach input in vector_set do
        if( (OBSERVABLE(node, fault_type, input) AND
            (!IMP_VIOLATION(node, fault_type, input) )
                fault_undetected++;
        end if
    end for
    return 100 * (1−fault_undetected / input_patterns);
```

Figure 4. Procedure IMPLICATION_QUALITY

3.2. Determining Implication Quality

Once we have a list of all implications in the circuit, it is possible to generate checker logic for each implication as a means of detecting errors in the circuit. However, including all these implications, in general, would be very inefficient. Typically, a circuit will contain thousands of implications, so generating checker logic for every implication could easily double or triple the size of the circuit. Ideally, the implications that can detect the most (or the most important) errors should be included in the checker logic. Finding these "important implications" requires a combination of structural and fault analysis on the circuit to determine the quality of each implication.

For a given set of input vectors, we compute whether a fault that is present at an internal node will propagate to the output (i.e., is observable). At the same time, we also check if that fault causes a violation of the implication being tested. The *quality* of that particular implication and fault pair is then computed as

$$implication\_quality = 100 \cdot \left(1 - \frac{fault\_undetected}{input\_patterns}\right) \quad (1)$$

The value *fault_undetected* is the number of input patterns where the fault is observable and the implication is not violated. The procedure for calculating this value is outlined in Figure 4.

Note that procedure IMPLICATION_QUALITY allows us to compute the quality of coverage for a particular fault on a node with a particular implication. To compute the implication quality for the entire circuit, we take the average implication quality over all the faults in the circuit.

Note that the procedure IMPLICATION_QUALITY is not specific to a particular fault type. However, in this paper we only consider stuck-at faults in our evaluation of implication quality. Although true stuck-at faults, which would be present on every clock cycle, do not accurately model the errors that we would be detecting online, analyzing stuck-at faults provides several advantages. Stuck-at faults are well-known, easy to model, and easier to compare to other work. Furthermore, many other errors will temporarily behave like stuck-at faults when they are present; they will temporarily cause incorrect values to occur in the hardware which may then propagate to a flip-flops or primary outputs. If we can detect such stuck-at faults effectively, it follows that we often will be able to detect some more esoteric errors that cause a logic value

```
Procedure COMPRESS_IMPLICATIONS(imp_set, best_imps, fault)
    foreach node in network do
        INJECT_FAULT(node, fault);
        best_quality = 0;
        foreach imp in imp_set do
            new_quality = IMPLICATION_QUALITY(imp, node, fault);
            if( (new_quality > best_quality) OR
                ( (new_quality == best_quality) AND
                  (DISTANCE(imp) > DISTANCE(best_imps[node])) ) )
                best_quality = new_quality;
                best_imps[node] = imp;
            end if
        end for
    end for
```

Figure 5. Procedure COMPRESS_IMPLICATIONS

to change only occasionally—e.g. for only particular input patterns or under other unknown or seemingly random conditions.

Thus, our goal is to try to cover all the stuck-at faults on all the nodes in the circuit using a set of valid implications. While it is possible that some faults will not be covered by any of the implications we discovered using the procedure outlined in Section 3.1, in general, we have found that there will be few faults in this category. On the other hand, it is possible that some of our implications may not be very effective at detecting faults in general, and these may be removed from our implication set without hurting our overall coverage significantly. To compute this reduced implication set, we use the procedure COMPRESS_IMPLICATIONS, as outlined in Figure 5, which calls the procedure IMPLICATION_QUALITY.

Procedure COMPRESS_IMPLICATIONS considers only one fault at a time while it iterates over each node in the network. For each node, it iterates through all implications and selects a single implication that has the highest quality for the particular fault on that node. This implication is saved in the array *best_imps* under the node index value. At the end of the procedure call it is likely that multiple nodes will share the same "best" implication, thereby allowing us to compress our implication list even further.

We ran experiments on a number of circuits from the ISCAS and MCNC benchmark suites to test the effectiveness of our implication compression procedure. An initial implication set was first generated and validated using the steps outlined in Section 3.1. We then ran COMPRESS_IMPLICATIONS on the implication set to reduce the number of implications in the set. The results of compression are show in Table 1. The total number of valid implications of distance greater than 2 is shown in column 2, and in column 3 we report the number remaining after the subsumed implications are eliminated. Column 4 shows the number of implications left after running procedure COMPRESS_IMPLICATIONS. The compression rate (comparing columns 3 and 4) is shown in column 5. The rate of compression depends on the specific benchmark; however, across all benchmarks, the number of implications can be compressed very efficiently leading to an average compression ratio of 84.5%.

Circuit	Orig.	After Subsume	After Compress	Compress Rate (%)
rd73	2082	998	154	84.6
Z5xp1	4296	2049	173	91.6
clip	2568	1126	160	85.8
Z9sym	2453	1203	134	88.9
b12	1070	461	83	82.0
misex2	2341	985	107	89.1
C432	1198	376	87	76.9
C499	6205	3833	112	97.1
C880	5054	1554	227	85.3
C1355	28716	13250	636	95.2
C1908	4023	1908	898	52.9
AVERAGE				84.5

Table 1. Percent savings after compression.

3.3. Selecting Implications

<mark>Note that even if we included all the valid implications in the checker logic, we most likely would not achieve 100% error detection;</mark> however, by completing the fault analysis described earlier, we can obtain an upper bound on the total fault coverage possible for a particular circuit using implication checkers. The next step is to determine the relative "effectiveness" of each of the remaining implications so that we may have a means of selecting a subset of these implications that satisfy a given constraint. For our purposes, we consider the following two conditions for selection:

- Maximizing the total number of faults detected given a specific hardware budget X, where X is a percentage of the total area of the original circuit.

- Maximizing the total number of "least-observable" faults detected given a specific hardware budget X.

The first approach does not favor one type of fault over another. Rather, it simply tries to minimize the total probability of an undetected error with the constraint that the hardware to implement the checker logic cannot be greater than $X\%$ of the area of the original circuit. This approach is appropriate if we are focusing mainly on detecting transient faults that would not have appeared during the normal chip testing process, and we are therefore trying to cover the broadest set of faults. To achieve this goal, we start by simply ranking all implications according to their average probability of an undetected error and then select the top N implications such that the checker logic does not exceed $X\%$ of the original circuit area.

The second approach is focused more on difficult-to-detect defects and process variations and may be appropriate if the goal is to detect online problems that were missed during test. An additional advantage of this approach is the fact that our inserted implication hardware can also serve as test points during manufacturing test, potentially allowing more defects in the internal logic to be detected during testing.

For this second approach, we start by first calculating the observability of a fault at a circuit node. We use this observability as a weight for the fault's probability of producing an undetected error. The circuit designer can adjust the weights based on what degree of importance is to be given to less observable faults. Now for each implication, we calculate the overall probability of an undetected error by taking the weighted average of each individual fault's probability of an undetected error. The implications are then sorted and the top N implications are selected for the checker logic such that the checker logic does not exceed a designer-specified $X\%$ of the original area.

4. Results

Once we have added the implication-based error checking hardware to our circuit, one of the following four scenarios is possible on each clock cycle when a fault occurs in the circuit logic:

- **Case 1**: Fault occurs at internal node and is visible at primary output(s) (or latches). Implication checker flags a violation. (True detection)

- **Case 2**: Fault occurs at internal node but is not visible at primary output(s) (or latches). Implication checker catches the fault and flags a violation. (False Positive)

- **Case 3**: Fault occurs at internal node but is not visible at primary output(s) (or latches). Implication checker does not detect violation. (Benign Miss)

- **Case 4**: Fault occurs at internal node and is visible at primary output(s) (or latches). Implication checker *does not* detect violation. (True Miss)

For the first case, our implication hardware is able to successfully detect a fault that would have resulted in an error at one or more primary outputs. In other words, the checker circuitry was able to detect an observable, or *important* fault. For the second and the third scenarios, a fault may have occurred somewhere in the circuit; however, the fault did not truly matter because it did not alter the correct values expected at the primary outputs (i.e., it was not observable since it wasn't sensitized by an appropriate input vector). The detection of such a fault by our checker hardware, as would be the case in scenario 2, may be wasteful in terms of forcing the re-execution of already-correct data by the circuit. However, in general, we expect the rate of transient errors to be relatively rare (on the order of 10^{-6} or less), so performance hits due to false positives are likely to be of little consequence in many applications. Furthermore, these detections, if saved, may provide information that could be used later on for tracking down intermittent faults. Of course, a permanent error will never allow the program to proceed, but this is also not highly problematic because a permanent error is likely to cause significant data corruption

in the future (if it has not already) and such errors cannot be corrected by simple re-execution. Case 3 is non-problematic because the circuit's outputs (and latches) all contain appropriate values and no error was flagged. Finally, case 4 is the worst possible scenario because a true error remains undetected by our checker. By optimally choosing which implications to include in our checker hardware, we aim to minimize the number of times faults fall into this category.

circuit	In	Out	Case 1	Case 2	Case 3	Case 4
rd73	7	3	12.3	30.2	55.3	2.1
Z5xp1	7	10	18.6	28.0	51.1	2.3
clip	9	5	12.1	29.0	55.1	3.9
Z9sym	9	1	3.7	36.9	56.2	3.2
b12	15	9	12.0	21.7	61.7	4.5
misex2	25	18	15.2	26.9	55.9	1.9
C432	36	7	12.5	25.7	59.7	2.0
C499	41	32	10.5	11.3	65.5	12.8
C880	60	26	18.9	13.3	58.1	9.8
C1355	41	32	5.4	25.9	64.4	4.3
C1908	33	25	20.6	26.3	50.6	2.5

Table 2. Distribution of the interaction between faults and input patterns.

Table 2 reports for the different benchmark circuits how often, on average, a fault will fall into each category when a set of random input patterns is applied. Data was generated considering all discovered implications. Also included in the table are the number of inputs and output signals for each circuit. The high percentage of faults falling under Cases 2 and 3 show that a majority of the errors that occur in the circuits are logically masked and are not observable at the outputs. That is, on average, 84% of the time a given fault doesn't affect the outputs. However, 26% of the time, although no change occurs at the output, our checker hardware will flag a violation. This is because, while the fault is being logically masked as it makes its way from its origin to the circuit output, the checker hardware does not have the capability to capture the logical masking effect and raises a violation flag.

Using the fault classification shown in Table 2, we can now compute how often an internal fault that is observable at one of the circuit's output is detected and flagged by the implication checker hardware (*i.e. Case1/[Case1+Case4]*). Results are shown in Figure 6.

Of all the benchmark circuits shown in Figure 6, we see that the implication hardware added to C499 detects only about 50% of the observable faults. C499 is a single-error-correcting circuit and an analysis of the benchmark reveals that 50% of the logic gates in the circuit are two-input XOR gates. The presence of XORs makes it harder to find meaningful logic implications that might enable us to cover faults at those nodes. Recall that for a two input AND gate with inputs A and B, a value of 1 at the output implies that input A must be at logic 1 and similarly a logic 0 at input B implies a 0 at the output. However, for an XOR with inputs A and B, a 0 or a 1 at the output does not allow us to infer the state of either A or B in isolation. Since we are using the logical implications between just
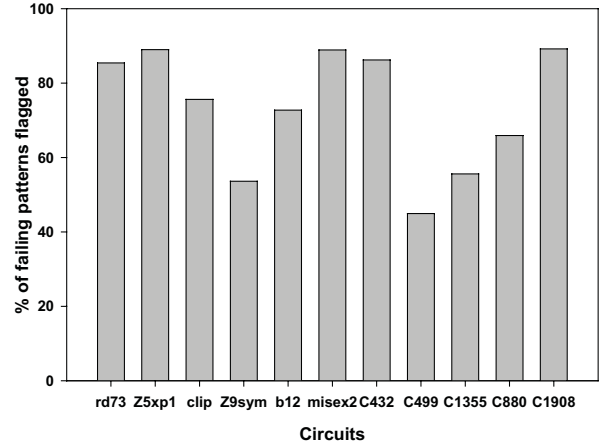
Figure 6. Detection of observable faults using implication checker circuitry.

two nodes in a circuit, nothing can be inferred for two input XOR gates that make up the majority of the circuit in C499 and thus leads to a lower error coverage compared to the remaining benchmarks. C1355 is functionally identical to C499 but all XOR gates have been expanded into NAND gates. This means more implications can be found for C1355 within the expanded XORs, and hence the detections at implication checker for C1355 is slightly better compared to C499. However, the detection is not as good as shown for some other benchmarks because the XOR structure still exists in abundance. We plan to address our limited ability to detect faults within circuits containing many XOR gates in future work. For instance, these types of faults may be detectable through the identification of more complex implications (i.e., those that consider relationships that exist when more than two circuit nodes/sites are considered).

5. Hardware Implementation Overhead

Up to this point, we have analyzed the error coverage that can be achieved using all the implications found after we apply the procedure COMPRESS_IMPLICATIONS (i.e., those from the 4th column in Table 1), with no mention of cost of implementing the checker. In this section we will discuss more explicitly the hardware implementation of the checker as well as the overhead involved.

The simplest checker implementation consists of a handful of gates for each implication that will indicate the violation of that particular implication. For example, if signal $A = 1$ implies signal $B = 0$, then the implication is violated if both A and B are equal to 1. A single AND gate suffices to flag this implication violation. Similarly, if $A = 1$ implies $B = 1$, the implication is violated if $A = 1$ and $B = 0$. This implication violation can be flagged with an AND gate
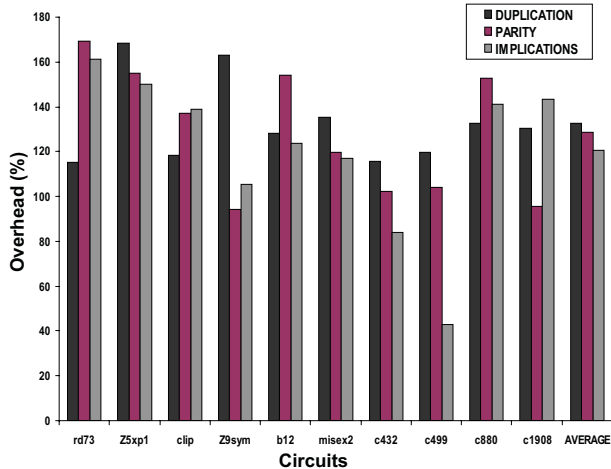
Figure 7. Area overhead for the 3 different error detection techniques.

and an inverter so that the signal A can be AND'ed with the inverted value of B. Once appropriate signals are generated for each implication being checked, these individual signals can be OR'ed together with a wired OR to create a single violation signal.[2]

To determine the true hardware overhead when all non-subsumed implications are included in the checker logic, checkers like the one described above were created for each benchmark circuit using a 180 nm TSMC library and Mentor Graphics Toolset. For comparison, two additional versions of each circuit were also implemented. In the first case, the circuit was duplicated, and checker logic consisted of an XOR of each output with its copy. The XOR's were OR'ed together to create a single error signal. In the second case, a simple parity implementation was also derived. The parity prediction circuitry was optimized for area, and the parity of the outputs was calculated and compared to the predicted parity to create single error single. We note that no additional logic or circuit modifications were used to eliminate the possibility of aliasing; thus, a parity implementation could theoretically be more expensive than that shown here. The results are shown in Figure 7.

The results show that, on average, logic duplication had the most overhead followed by parity code and then our implication checker. For any individual circuit, the number of implications available has a significant impact on the size of the overhead shown. For other error detection techniques employing codes such as Berger and Bose-Lin codes, Mitra showed that compared to simple duplication, Berger codes had an extra 41% area overhead and Bose-Lin codes had an extra

28% overhead [24].

A significant advantage of our method is the ease of trading off error coverage with hardware overhead by only including a subset of the available implications in our checker logic. With this in mind, we next used the procedure described in Section 3.3, to determine how error coverage may be affected given a user-specified limit on the hardware overhead of the checker circuitry. For these experiments, we approximated hardware overhead as total number of extra gates added compared to the original gate count of the circuit, where each implication check is assumed to be implemented using a single gate. Using such an approximation is useful because it allows implications to be chosen without explicitly generating the layout. Once a layout is actually generated, the exact overhead may vary somewhat due to routing and other issues. However, we have found that on average, the actual overhead is often relatively close. For the circuits studied, our actual average overhead was 12.6% when 10% overhead was targeted and 57.5% when 50% overhead was targeted.

Table 3 shows the results for the benchmark circuits. The column marked *no obs* refers to the case where the logical implications derived for these circuits were sorted and picked based on their coverage of all the faults with no emphasis placed on the observability of the faults. The columns *more obs* and *less obs* report the average probability of undetected error when the logical implications used in the checker circuit where chosen according to their coverage of the more observable and less observable faults respectively. For all benchmark circuits, it can be seen that ==checker hardware favoring implications that covered the less observable faults always had a higher probability of an observable fault being undetected at the checker compared to the other two cases.== An analysis of the coverage of individual nodes of these circuits shows that the less observable nodes can be covered by a wider range of implications. For instance, in C432, nodes that are observable only one-third of the time have an average probability of undetected error of 11.5% while those that are observable two-thirds of the time have a lower probability of being undetected of only 6%. ==This implies that choosing implications that detect the more observable faults first will yield a lower average probability of undetected error for the entire circuit since the less observable faults can be covered more easily by the remaining implications.== The results in Table 3 also show that even with a very low area overhead significant coverage can be achieved. This is very encouraging and provides a circuit designer an opportunity to make intelligent trade-offs between reliability and circuit area.

6. Conclusions

We have presented an online error detection technique based on the use of logic implication information. Since more implications exist in a circuit than can

---

[2]It should be noted that such an implementation is not fully testable because some internal faults cannot be excited when no errors are present in the circuit. This can be resolved through some additional logic and a test control signal that will enable the implication to be artificially violated during test.

Circuits	no obs					less obs					more obs				
	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%	10%	20%	30%	40%	50%
b12	12.3	9.7	8.5	7.8	7.4	12.5	11.2	10.2	9.0	8.4	12.0	9.8	8.7	7.9	7.5
misex2	12.8	9.5	7.4	6.1	5.2	14.6	13.4	12.7	10.3	8.5	12.8	10.0	7.7	6.7	5.2
rd73	7.2	6.3	5.8	5.0	4.4	8.4	6.4	5.7	5.0	4.5	7.3	6.0	5.7	5.0	4.5
Z5xp1	14.5	11.5	10.0	9.0	7.9	14.3	11.9	9.9	9.00	8.2	14.6	12.0	10.0	8.4	7.7
clip	11.0	10.2	8.9	7.8	6.7	11.0	10.2	9.3	8.2	7.2	11.6	9.7	8.4	7.2	6.6
Z9sym	5.6	5.2	4.8	4.5	4.3	5.6	5.2	4.8	4.5	4.3	5.7	5.3	4.8	4.5	4.3
C432	7.2	4.5	3.9	3.3	2.7	7.4	5.9	5.2	4.6	3.0	7.3	4.3	3.5	3.0	2.3
C499	16.9	13.7	13.7	13.6	13.5	16.9	14.0	13.7	13.6	13.5	16.9	13.7	13.7	13.6	13.5
C880	22.6	19.1	17.4	16.2	15.1	23.8	21.4	19.4	18.3	16.9	23.1	20.2	18.6	16.7	15.2
C1908	14.3	13.1	11.0	8.4	7.6	14.5	13.4	11.1	9.8	9.0	13.9	11.5	9.0	7.4	6.6

Table 3. Variation of the probability of undetected error given a fixed hardware overhead when implications are selected based on degree of node observability.

be efficiently represented in hardware, it is important to identify the most useful implications to check such that the fraction of errors detected can be maximized while minimizing the additional hardware overhead. A key advantage of our approach is that we automatically identify these implications without requiring any knowledge of high-level behavior constraints. In addition, our approach does not require re-synthesis of the targeted logic; the checker logic is added off the critical path and is run in parallel with the regular control logic. The amount of error coverage obtainable varies by circuit; however, for several circuits, we have shown that we can detect almost 90% of all errors that propagate to a primary output by using implication-based checker logic. Furthermore, with only a 10% area overhead, our results show that on average it is possible to reduce the probability of an error being both observable, and undetected, to about 11%.

For future work, we are currently exploring the effectiveness of using implication information to detect errors across multiple cycles. Detecting these implications will inherently increase the logical "distance" of our potentially useful implications. Moreover, these implications may be more effective for detecting faults that are physically located only a few levels upstream from the flip-flops or latches in the design. These faults are generally among the most difficult to detect with single-cycle implications. Implications across time cycles also have the potential to be very effective at detecting some types of delay faults. Specifically, a delay that causes an incorrect value to be latched at the flip-flops will create a logical discrepancy when considered across multiple clock cycles.

References

[1] Yael Abarbanel, Ilan Beer, Leonid Glushovsky, Sharon Keidar, and Yaron Wolfsthal. FoCs: automatic generation of simulation checkers from formal specifications. In *CAV '00: Proceedings of the 12th International Conference on Computer Aided Verification*, pages 538–542, 2000.

[2] S. Almukhaizim, P. Drineas, and Y. Makris. Cost-driven selection of parity trees. In *VLSI Test Symposium*, pages 319–324, 25-29 April 2004.

[3] Sobeeh Almukhaizim, Yiorgos Makris, Yu-Shen Yang, and Andreas Veneris. Seamless integration of SER and rewiring-based design space exploration. In *International Test Conference*, pages 1–9, Oct. 2006.

[4] T. M. Austin. Diva: A reliable substrate for deep-submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, November 1999.

[5] R.I. Bahar, M. Burns, G. Hachtel, E. Macii, H. Shin, and F. Somenzi. Symbolic computation of logic implications for technology-dependent low-power synthesis. In *Proceedings of the International Symposium On Low Power Electronics And Design*, pages 163–168, Monterey, CA, USA, August 1996.

[6] J. M. Berger. A note on an error detection code for asymmetric channels. *Information and Control*, 4:68–73, 1961.

[7] Dilip K. Bhavsar. An algorithm for row-column self-repair of RAMS and its implementation in the Alpha 21264. In *Proceedings of th 1999 International Test Conference*, pages 311–318, 1999.

[8] S. Bobba and I.N. Hajj. Estimation of maximum current envelope for power bus analysis and design. In *Proceedings of the International Symposium on Physical Design*, pages 141–146, Monterey, CA, USA, April 1998.

[9] B. Bose and D. J. Lin. Systematic unidirectional error-detecting codes. *IEEE Transactions on Computers*, C-34:1026–1032, 1985.

[10] Marc Boule, Jean-Samuel Chenard, and Zeljko Zilic. Assertion checkers in verification, silicon debug and in-field diagnosis. In *ISQED '07: Proceedings of the 8th International Symposium on Quality Electronic Design*, pages 613–620, 2007.

[11] F.M. Brown. *Boolean Reasoning*. Kluwer Academic Publishers, Hingham, MA, USA, 1990.

[12] M. Choi, N. Park, F. Lombardi, Y.B. Kim, and V. Piuri. Balanced redundancy utilization in embedded memory cores for dependable systems. In *Proceedings 17th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, pages 419–427, December 2002.

[13] D. Das and N. A. Touba. Synthesis of circuits with low-cost concurrent error detection based on Bose-Lin codes. In *VLSI Test Symposium*, pages 309–315, 1998.

[14] K. De, C. Natarajan, D. Nair, and P. Banerjee. RSYN: a system for automated synthesis of reliable multilevel circuits. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2:786–195, 1994.

[15] Rolf Drechsler. Synthesizing checkers for on-line verification of system-on-chip designs. In *ISCAS '03: Proceedings of the 2003 International Symposium on Circuits and Systems*, pages IV–748 – IV–751, May 25-28 2003.

[16] P. Drineas and Y. Makris. Concurrent fault detection in random combinational logic. In *Proceedings Fourth International Symposium on Quality Electronic Design ISQED*, pages 425–430, March 2003.

[17] S. Ghosh, N.A. Touba, and S. Basu. Synthesis of low power ced circuits based on parity codes. In *VLSI Test Symposium*, pages 315–320, 1-5 May 2005.

[18] A. Glebov, S. Gavrilov, D. Blaauw, and V. Zolotov. False-noise analysis using logic implications. *ACM Trans. Des. Autom. Electron. Syst.*, 7(3):474–498, 2002.

[19] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, June 2005.

[20] Michael Gössel. *Error Detection Circuits*. McGraw-Hill, London, NY, USA, 1993.

[21] Smita Krishnaswamy, Stephen M. Plaza, Igor L. Markov, and John P. Hayes. Enhancing design robustness with reliability-aware resynthesis and logic simulation. In *ICCAD*, pages 149–154, Nov. 2007.

[22] W. Kunz and P.R. Menon. Multi-level logic optimization by implication analysis. In *Proceedings of the International Conference on Computer-Aided Design*, pages 6–10, San Jose, CA, USA, November 1994.

[23] Y. Mahajan, Z. Fu, and S. Malik. Zchaff2004: An efficient sat solver. *Lecture Notes in Computer Science*, 3542:360–375, 2005.

[24] S. Mitra and E.J. McCluskey. Which concurrent error detection scheme to choose? In *Proceedings International Test Conference*, pages 985–994, October 2000.

[25] K. Mohanram, E. Sogomonyan, M. Gossel, and N. Touba. Synthesis of low-cost parity-based partially self-checking circuits, 2003.

[26] K. Mohanram and N.A. Touba. Cost-effective approach for reducing soft error failure rate in logic circuits. In *Proceedings International Test Conference*, pages 893–901, October 2003.

[27] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual IEEE/ACM international symposium on Microarchitecture*, pages 214–244, 2001.

[28] S. K. Reinhardt and S. S. Mukherjee. Transient-fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[29] Bernhard Rohfleisch, Alfred Kölbl, and Bernd Wurth. Reducing power dissipation after technology mapping by structural transformations. In *Design Automation Conference*, pages 789–794, New York, NY, USA, 1996. ACM.

[30] R. Sedmak and H. Liebergot. Fault tolerance of a general purpose computer implemented by very large scale integration. *IEEE Transactions on Computers*, C-29:492–500, 1980.

[31] R. Sedmak and H. Liebergot. Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs. *IEEE Transactions on Nuclear Science*, 51:2957–2969, 2005.

[32] R. Sharma and K.K. Saluja. An implementation and analysis of a concurrent built-in self-test technique. In *Digest of Papers Eighteenth International Symposium on Fault-Tolerant Computing FTCS-18*, pages 164–169, June 1988.

[33] E. S. Sogomonjan and Michael Gössel. Design of self-parity combinational circuits for self- testing and on-line detection. In *Proceedings of the IEEE International Workshop on Defect and Fault Tolerance in VLSI Systems*, pages 239–246, Washington, DC, USA, 1993. IEEE Computer Society.

[34] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault-tolerance. In *Proceedings of the Ninth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, November 2000.

[35] P. Tafertshofer, A. Ganz, and M. Henftling. A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists. In *Proceedings of the International Conference on Computer-Aided Design*, pages 648–655, San Jose, CA, USA, November 1997.

[36] N.A. Touba and E.J. McCluskey. Logic synthesis of multilevel circuits with concurrent error detection. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(7):783–789, Jul 1997.

[37] M. R. Trinka. Defect site prediction based upon statistical analysis of fault signatures. Master's thesis, Texas A&M University, August 2003.

[38] J. von Neumann. Probabilistic logics and synthesis of reliable organisms from unreliable components. In C. Shannon and J. McCarthy, editors, *Automata Studies*, pages 43–98. Princeton University Press, 1956.

[39] C. F. Webb and J. S. Liptay. A high frequency custom CMOS S/390 microprocessor. *IBM Journal of Research and Development*, 41:463–473, 1997.