# A Cost Effective Approach for Online Error Detection Using Invariant Relationships

Nuno Alves, *Student Member, IEEE,* Alison Buben, Kundan Nepal, *Member, IEEE,* Jennifer Dworak, *Member, IEEE,* and R. Iris Bahar, *Member, IEEE*

***Abstract*—This paper investigates the use of logic implication checkers for the online detection of errors. A logic implication, or *invariant relationship*, must hold for all valid input conditions; therefore, any violation of this implication will indicate an error due to an intermittent fault. Techniques are presented to efficiently identify the most useful logic implications to include in checker hardware such that the probability of error detection is maximized while minimizing the additional hardware and delay overhead. Results show that significant error detection is possible—even with only a 10% area overhead—while minimizing impact on delay and power.**

***Index Terms*—Fault detection, implications, invariance, online error detection, reliability.**

## I. Introduction

FOR YEARS TO COME, the International Technology Roadmap for Semiconductors will continue to pursue aggressive size scaling. However, as complementary metal-oxide-semiconductor scales, circuit reliability degrades due to a host of issues, including defects and single event upsets. Ensuring reliable computation requires mechanisms to detect and correct errors during normal circuit operation.

The strategies employed for detecting errors vary, e.g., 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 random 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 random 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 do not know the correct results *a priori*, and thus, more complicated or highly expensive schemes often must be used.

Many previous approaches have introduced some sort of redundancy to provide for detection of errors in random 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], [2]).

In this paper, we present a novel approach to online error detection that *automatically* identifies *gate-level* invariant relationships that must be satisfied for the circuit to be operating correctly. No knowledge of high-level behavioral constraints is 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 Fig. 1. While the outputs of the random combinational logic are being computed, the invariant relationships within the circuit are checked in parallel. Any violations of these invariant relationships will be detected and used to generate an error signal. Note that while this figure shows a combinational circuit, the method applies equally well to sequential circuits—both within and across multiple clock cycles.

The invariant relationships we investigate in this paper are logic value implications that exist between pairs of circuit sites. Consider the combinational circuit shown in Fig. 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 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).

Manuscript received August 18, 2009; revised December 4, 2009. Current version published April 21, 2010. This work was supported in part by the National Science Foundation, under Grants CCF-0915302 and CCF-0915884. This paper was recommended by Associate Editor, F. Lombardi.

N. Alves, J. Dworak, and R. I. Bahar are with the Division of Engineering, Brown University, Providence, RI 02912 USA (e-mail: nuno@brown.edu; Jennifer_Dworak@brown.edu; Iris_Bahar@brown.edu).

K. Nepal is with the Department of Electrical Engineering, Bucknell University College of Engineering, Lewisburg, PA 17837 USA (e-mail: kundan.nepal@bucknell.edu).

A. Buben is with the Department of Computer Science, Indiana University of Pennsylvania, Indiana, PA 15705 USA (e-mail: a.j.buben@iup.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2010.2043590

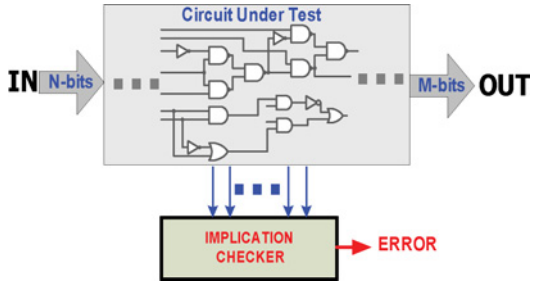


Fig. 1. Combinational circuit with implication checker hardware.

While the circuit shown in Fig. 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 in random logic during run time. However, including all possible implications in the checker logic is generally too costly in terms of area and delay overhead, so some means of selecting the most valuable implications needs to be employed.

In this paper, 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 an error escaping detection, but to cost-effectively reduce the chance of that escape occurring.

This article makes the following contributions.

1) We present an efficient method for identifying implications within a circuit using a combination of circuit simulation and satisfiability (SAT)-based techniques.
2) We show that implications may exist over multiple time cycles, and monitoring these implications in the checker logic can significantly improve error detection rates for some circuits.
3) We present a heuristic method for selecting the most useful of these implications to be included in the checker logic. In addition, we show how area and delay constraints can be easily folded into the implication selection process.
4) We show that our approach is very power efficient compared to other fault detection techniques.

The rest of the paper is organized as follows. Section II discusses related work in online error detection and points out the unique features of our approach. In Section III, we present our algorithms for identifying implications, both within a single-cycle and across multiple cycles, and then selecting a subset of these implications given a user specified area and delay budget. We evaluate our proposed method in Section IV on a set of benchmarks from the International Symposium on Circuits and Systems (ISCAS) and discuss the significance of the results. Finally, in Section V, we conclude this paper and discuss possible future extensions to our work.

## II. 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 also 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 [3], [4].

In the case of online detection of errors in the logic, alternative methods must be considered. In general, these methods involve the use of some type of redundancy in time, space, or information. 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 [5]–[8]. Another example of time redundancy involves duplicating instructions whenever processor resources are available [9].

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 [10]. In some cases, duplication with a complementary form of the hardware has been proposed [11]. If error correction is to be obtained, triple modular redundancy (TMR) can be used instead [12]. 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 static random access memory-based field programmable gate arrays (FPGAs) in [13]. In this case, single-event upset (SEU)-sensitive subcircuits were identified and only these subcircuits were protected with TMR. Similarly, the approach presented in [14] 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 [15]–[18]. 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 [19]–[21]. Other works have focused on the use of unidirectional codes such as Berger [22] or Bose–Lin codes [23], [24]. Several different coding schemes were compared in [25]. 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) [26] and reduced observation width

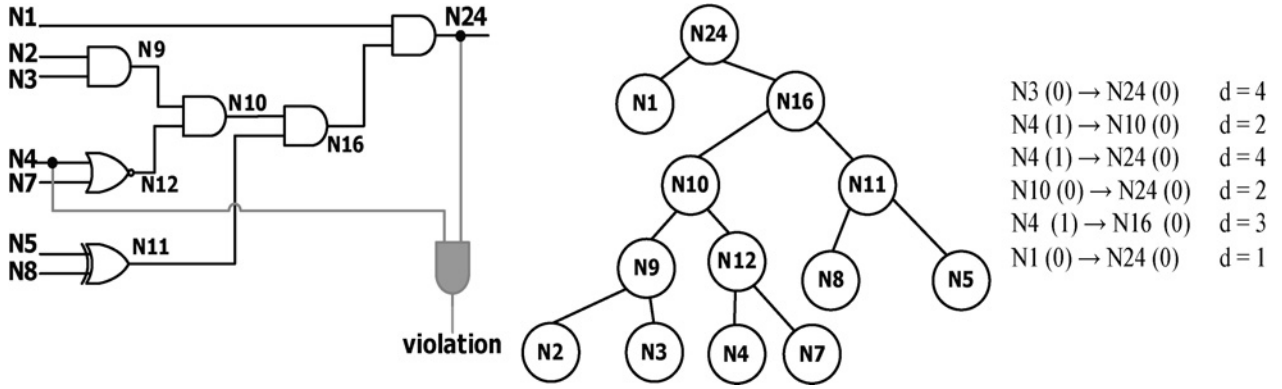


Fig. 2. Combinational circuit, its graphical representation, and list of sample implications.

replication (ROWR) [27]. For BICST, hardware is added to predict the appropriate response to a set of precomputed 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 paper are different from previous papers 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, [1], [2], [28], [29]). 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 [30]. Logic implications can be detected using recursive methods [31]–[33], and other heuristics [34], [35] and have been used widely in logic synthesis for such purposes as area/delay/power optimization [36]–[38], compaction of test vectors [39], peak current estimation [40], false noise analysis [41], and efficient automatic test pattern generation [42]. Logic implications have also been used as a means of guiding localized circuit restructuring to increase the chance of logically masking errors [43], [44]. While these masking techniques are promising for reducing the soft error rates in circuits, they require modification of the original circuit that may be unappealing to their designers.

In contrast, in this paper we incorporate checker logic that operates on the *original* synthesized version of the circuit—not requiring resynthesis of the circuit being protected. Our work is similar to [45] in that our goal is to make the addition of new hardware as unobtrusive as possible; however, unlike [45], we do not limit ourselves to relationships that exist among flip-flops within a single-cycle. In addition, [45] limits itself to just checking the outputs of a circuit in the control logic of a processor against a subset of the truth-table. What we are proposing is more general.

Our work also has some similarity to [46], where *checking functions* are identified and used to reduce latency for fault detection in online testing. More specifically, they look for a relationship where a site (or a Boolean combination of two sites) is functionally equivalent to another site. We do not restrict our checker logic to include only relationships that imply full equivalence. In addition, their work assumes a fault will persist across multiple time cycles, whereas we evaluate implications on a clock-cycle by clock-cycle basis, making our approach potentially better suited for transient faults.

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]

In practice, a circuit may contain thousands of single and multicycle logic implications, more than would be practical to include in the checker hardware. A major contribution of this paper is our development of an algorithmic flow for identifying the most useful of these logic implications to be included in the checker logic. In this way, our scheme will allow explicit trade-offs to be made between error coverage and acceptable hardware and performance overhead. The following section describes our algorithmic flow in more detail.

## III. Algorithm

Finding all implications in a circuit can become a daunting and time consuming task. In addition, due to the high-area overhead required, it is generally not cost-effective to implement *all* implications in checker hardware. 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 (both single and multicycle) and selects those that are most valuable for error detection. In the simplest case, "most valuable" can be defined as those implications that detect the most faults under most input conditions.

[1]Note that the checker logic itself may contain defects. The presence of any redundant defects may reduce the error coverage ultimately achieved. If full testability of the checker logic is desired, this generally can be easily accomplished with the addition of a test control signal and some additional gates in the implication checker.

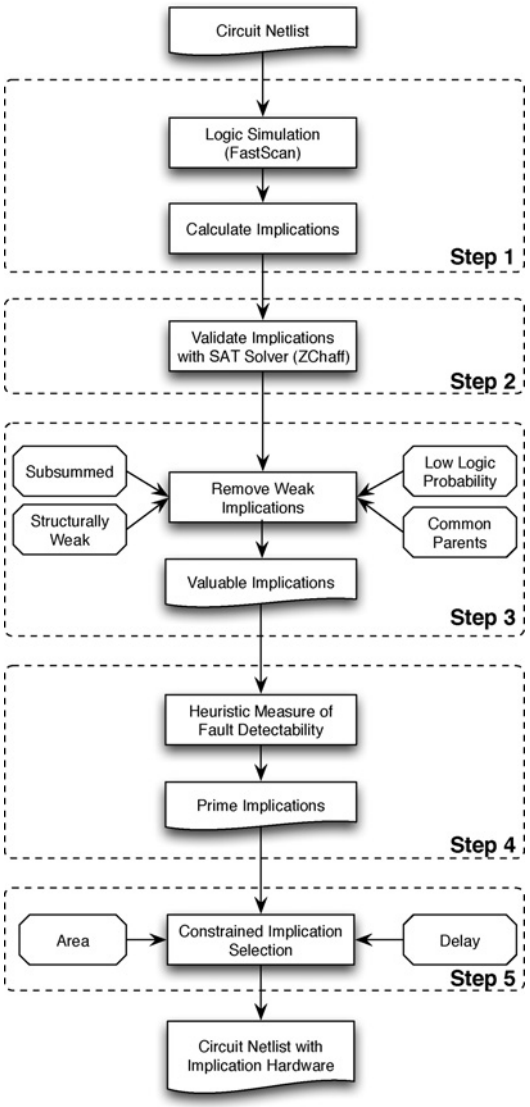


Fig. 3. Implication generation/addition flow.

The basic flow of our approach is shown in Fig. 3 and includes the following steps.

1) Run logic simulation to quickly identify *potential* implications.
2) Check all *potential* implications for validity.
3) Eliminate weak implications.
4) Perform logic simulation to assert which implications are the most effective.
5) Select a subset of implications such that the highest probability of a detected error is obtained given a user-specified hardware/power/delay budget.

The first two steps are part of the "implication discovery" process, while the last three intelligently select a subset of these implications to include in the checker hardware. The remainder of this section describes these steps in more detail. We use the following terms in our description.

1) *Good circuit simulation:* Logic simulation of a circuit devoid of faults.
2) *Implicant:* In the implication $P \Rightarrow F$, P is the implicant.
3) *Implicand:* In the implication $P \Rightarrow F$, F is the implicand.
4) *Valuable implications:* Implications that are likely to detect many faults (output of step 3 in Fig. 3).
5) *Prime implications:* A subset of the valuable implications obtained after running our heuristic algorithm.
6) *Topological distance:* By defining a circuit to be an undirected topological graph and a circuit site a vertex, then the distance between two sites is the minimum length of the paths connecting them.

### A. Implication Discovery—Single-Cycle

Our process of finding implications starts with a circuit netlist. With this netlist we run good circuit simulation over 32 000 vectors, using the Mentor Graphics FastScan tool (v8.2009.1). Since implications can reside anywhere in the circuit, for each input vector applied to the circuit, we record the logic values on all nodes in the circuit.

Once the simulation is complete, we load the logic simulation output into a customized C++ application which can identify potential implications by checking to see if certain node pair values are absent in the simulation runs. For instance, if the implication $(a = 1) \Rightarrow (b = 1)$ 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.

### B. Implication Discovery—Multicycle

Our approach to find implications is general and can be easily extended to include implications *across multiple time cycles* during our analysis. In this way, it may be possible to discover that the value of a circuit site at time $t$ may imply a value at another circuit site (or even the same circuit site) at time $t + n$, where $n \geq 1$. Checking for these additional implication violations across clock cycles has the potential to be a particularly powerful approach for error detection.

Consider the simple circuit shown in Fig. 4(a). There are no useful implications across more than a single gate that we can use for error checking in this circuit. However, we can search for implications across time cycles by creating a virtual copy of the circuit [as shown in Fig. 4(b)] and executing our implication search over this expanded circuit. In this case, we find that $B = 0$ in the first clock-cycle implies that $F = 0$ in the second clock-cycle.

From this example, which can be generalized to more complex circuits, we can see that detecting implications that exist across multiple time cycles will increase the number of implications that we can consider. More importantly, implications across time cycles will be more effective for detecting faults that are physically located only a few gates from the flip-flops and latches in the design. These faults are generally among the most difficult to detect with single-cycle implications. Thus, faults that were not adequately covered by implications in a single time-cycle may find themselves better covered across cycles with the additional topological distance.

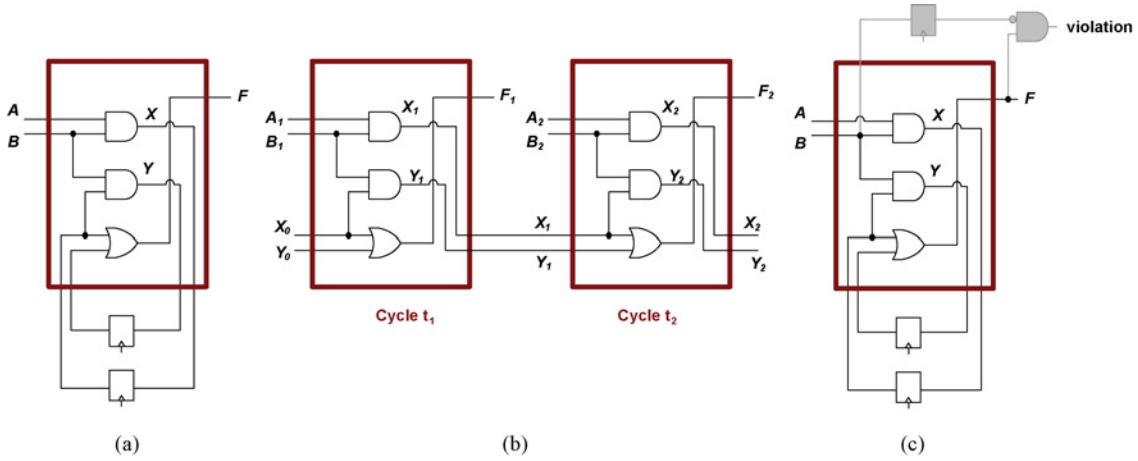


Fig. 4. (a) Simple sequential circuit. (b) Circuit expansion over two time frames. (c) Circuit with checker hardware (in gray) that raises a violation when implication $B_1 = 0 \Rightarrow F_2 = 0$ is violated.

Since our approach for finding implications across multiple cycles consists of creating virtual copies of the sequential circuits (essential converting them into combinational ones), the method outlined in Section III-A can be used for both sequential and combinational circuits.

### C. Implication Validation

Note that all the implications up to this point, are only *potential* implications. Since we are running test vectors on only a sample of all possible $2^{n+s}$ input vectors, where $n$ is the number of inputs to the circuit and $s$ is the number of flip-flops, 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. To accomplish this, we use the ZChaff [47] SAT solver 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 order to use ZChaff to verify the implications, it is required that we first convert our circuit description from Verilog format into conjunctive normal form (CNF). This procedure is done using the generalized CNF formulas for various gates described in [48]. Once the circuit is converted, each implication is separately added into the CNF circuit description and checked for validity in the SAT engine.

Using again the implication $(a = 1) \Rightarrow (b = 1)$ as an example, the SAT solver would verify if, for the given network, the opposite condition $a = 1$ and $b = 0$ can be excited. If the SAT solver fails, which implies that the condition is *FALSE* for all possible variable assignments 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 implications serially very quickly, even for large circuits, and in the end generate a list of all true implications for the circuit.

### D. Removing Weak Implications

As previously stated, the goal of our implication checker logic is to add some online error detection capabilities to a circuit. Once we have our preliminary list of validated implications, it is possible that some implications are not useful in terms of improving fault coverage. Those implications can then be safely discarded from the list of implication consideration. Through our studies we are able to group these ineffective implications into four types:

1) implications that are subsumed by other implications;
2) implications with low probability of being triggered;
3) structurally weak implications;
4) implications with common parents or offsprings.

1) *Subsumed Implications:* We discovered 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 Fig. 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 $\Rightarrow$ N24 = 0) has the longest topological 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. These subsumed implications can be quickly detected, and removed, by converting the circuit into a graph. As an example, Fig. 2 includes the graph for the combinational circuit described in Section I. Applying this quick preprocessing step immediately after we verify the validity of the implication reduces the number of viable implications by 40–70%.

2) *Low-Probability Implications:* There are certain implications that, despite their structural ability to cover many distinct faults, when added to the circuit have a negligible impact on error detection. This is normally a consequence of the circuit function which prevents the occurrence of

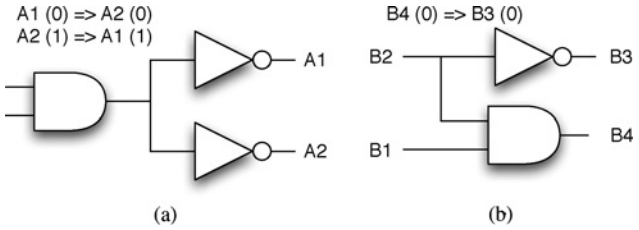


Fig. 5. (a) Two structurally weak implications. (b) Implication whose implicant and implicand share the common parent B2.

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

Fig. 6. Procedure COMPRESS_IMPLICATIONS.

the appropriate logic values that trigger an implication. In our experiments, we noticed that implications whose implicant node has less than a 5% chance of having the proper logic signal on good circuit simulation with 32 000 random vectors, tend to have an inconsequential impact on error detection. Such implications are prime candidates for removal.

While this threshold value could be optimized for each circuit, we chose to use a single value for all circuits since the sole purpose of this step is to quickly remove inefficient implications from further consideration. Empirically, 5% turned out to be a reasonable value.

3) *Structurally Weak Implications:* Throughout the circuit there can be several equivalent implication pairs. As demonstrated in Fig. 5(a), these equivalent implications are those for which the implicant and implicand are in both implications but in reversed order. As expected this occurrence is most frequent in buffer chains. In situations such as these, since both implications have an equivalent coverage, we can safely remove one of them.

4) *Common Parent Nodes:* Our experiments also showed us that implications where the implicant and implicand have a common parent (or a common offspring) tend to have a negligible effect on error detection and can be removed from consideration. This is mainly due to the fact that these implications tend to cover very few faults. Fig. 5(b) illustrates this type of ineffective implication.

By weeding out ineffective implications in the steps just described, we are able to dramatically reduce the number of implications. We named this reduced set of implications valuable implications. However, we may still be left with more implications than would be practical to use in the checker logic. Indeed, generating checker logic for every valuable implication at this point 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.

### *E. Determining Implication Quality: A Heuristic Measure of Fault Detectability*

We now describe our approach that quickly estimates which of the valuable implications can cover the most observable faults, and sorts them in order of importance for the purposes of fault detection.

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), and at the same time, we also check if that fault causes a violation of the implication being tested. We define *implication detection* as the number of times this implication under test is violated when the fault is observable. This *implication detection* metric is not specific to a particular fault type. However, in this paper we only consider stuck-at faults in our evaluation. 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 often used in 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 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, 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 Fig. 6, with 32 000 random vectors.

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 detectability for the particular fault on that node. This implication identifier, together with its number of detections, is saved in the array *best_imps* under the node index value. This procedure does not favor one type of fault over another; its sole purpose is to quickly process data and disregard redundant implications which cover faults better covered by other implications. Once all faults are processed, we sort the implications in descending order according to number of fault detections with which

they are associated. We call this sorted implication list prime implications. With this method, an implication closer to the top of the same list is more likely to be effective at detecting errors than implications located toward the end of the same list.

### F. Selecting Implications

Note that even if we included all the prime implications in the checker logic, we most likely would not achieve 100% error detection; 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 only these prime implications. The next step is to select an appropriate subset of these implications that satisfies a given hardware budget constraint. For our purposes, we consider the following two conditions for selection.

1) *Maximizing the Probability of Detection:* The first selection criterion 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 a user specified percentage of the original circuit area. To achieve this goal, we start by selecting the top N prime implication such that the checker logic does not exceed $X\%$ of the original circuit area.

2) *Improving the Critical Path Delay:* In some circumstances, implementing the implication subset that yields the highest probability of error detection also carries an undesirable increased delay due to the increased load capacitance on critical path nodes. Here we present an algorithm that aims to select an implication subset with a reduced impact on critical path delay while maintaining a reasonable probability of detection. We initially add the entire prime implication set to the circuit and by running static timing analysis we are able to determine which of those implications are on the critical path. With that result in hand we then create a new subset of prime implications which allows only a limited number of implications to include nodes on a critical path. In particular, the new subset is created such that (a) no more than 10 implications can be from nodes on the critical path, and (b) a specific node cannot be included in more than four implications. These values were chosen empirically after experimenting with a range of values.

## IV. Results

We ran experiments with a number of combinational and sequential circuits from the ISCAS'85 and ISCAS'89 benchmark suites to validate the effectiveness of our approach. Each set of implications was derived using the general algorithm described in Section III which combines the use of random vector simulation and the ZChaff SAT solver to identify and validate implications. This process is relatively fast and can quickly identify all gate level implications without the need for any high-level functional information. We first evaluate the types of implications we discover. Next, we look at the impact our compression procedure has on the number of implications we retain as "high quality." We also evaluate the fault detection capability of our implication-based checker hardware with and without imposing hardware and delay constraints. Finally, we compare the power dissipation of our approach with logic duplication and parity approaches.

TABLE I
Number of Implications in Each Class

| Circuit | # of | | | | # of Implications | | |
|---|---|---|---|---|---|---|---|
| | PI | PO | FF | Gates | First Cycle | Cross -Cycle | Second Cycle |
| s298 | 3 | 6 | 14 | 75 | 1687 | 3300 | 66 |
| s420 | 19 | 2 | 16 | 140 | 12 898 | 20 761 | 0 |
| s444 | 3 | 6 | 21 | 119 | 3054 | 8468 | 546 |
| s510 | 19 | 7 | 6 | 179 | 17 845 | 25 556 | 260 |
| s713 | 35 | 23 | 19 | 139 | 11 485 | 10 819 | 2789 |
| s953 | 16 | 23 | 29 | 311 | 13 197 | 13 065 | 34 |
| s1196 | 14 | 14 | 18 | 388 | 19 781 | 2599 | 164 |
| s1488 | 8 | 19 | 6 | 550 | 20 822 | 17 336 | 750 |

### A. Single Versus Multicycle Implications

Three different sets of implications were collected for each sequential circuit:

1) first cycle implications (implications at time $t$);
2) second cycle implications (implications at time $t + 1$);
3) cross-cycle implications (implications between time $t$ and $t + 1$).

We identify first cycle implications by simulating the sequential circuit for a single-cycle (with multiple vectors) where flip-flops are treated as pseudo-primary inputs and pseudo-primary outputs (as would be done for simulation of combinational test patterns of a full-scan design). We assume that all inputs (true primary circuit inputs (PIs) and flip-flops) are independent, and thus only implications that are valid for all possible $2^{n+s}$ input combinations will be considered valid, where $n$ is the number of PIs, and $s$ is the number of flip-flops.

Second cycle and cross-cycle implications are both identified through the simulation of two clock cycles using time-frame expansion as was shown in Fig. 4(b). Specifically, two copies of the circuit are created, and the feedback through the flip-flops is broken and replaced with direct connections between the two circuit copies.

Second cycle implications only involve circuit sites that are both contained within the second copy of the circuit. While first cycle implications must be valid for all possible combinations of the flip-flops and primary input values, in subsequent clock cycles, the flip-flop values may be constrained due to the fact that some states are unreachable during correct operation of the circuit after the circuit is initialized. This effectively means that implications can be considered valid for a smaller set of input vectors, thereby increasing the chance that an implication will exist between two sites in a subsequent clock-cycle. If even more clock cycles were analyzed, more unreachable states would likely be discovered, and more implications could be considered. Finally, cross-cycle implications are those in which a value at a circuit site in the first cycle implies a value at a circuit site in the second cycle.

In Table I, we report the number of ZChaff validated implications, discovered from each class (first cycle, between

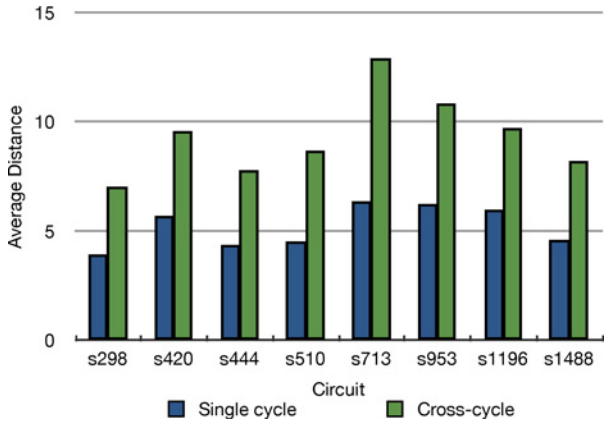


Fig. 7. Average implication distance for single and between-cycle implications.

TABLE II
IMPLICATION COUNT REDUCTION AFTER RUNNING OUR HEURISTIC MEASURE OF FAULT DETECTABILITY

| Circuit | Validated | Valuable | Prime | Compress Rate (%) |
|---|---|---|---|---|
| c432 | 1484 | 543 | 105 | 92.9 |
| c499 | 2496 | 968 | 81 | 96.7 |
| c880 | 3360 | 1297 | 284 | 91.5 |
| c1355 | 24 280 | 7409 | 615 | 97.4 |
| c1908 | 36 954 | 13 588 | 797 | 97.8 |
| c2670 | 75 578 | 31 643 | 612 | 99.1 |
| c3540 | 147 952 | 54 633 | 1632 | 98.9 |
| c5315 | 63 336 | 1588 | 1126 | 98.2 |
| c6288 | 13 754 | 6343 | 1832 | 86.7 |
| c7552 | 203 474 | 94 655 | 3074 | 98.4 |
| Average | | | | 95.8 |

cycles, or second cycle only). Note that by expanding the search beyond a single-cycle, many more implications can be discovered. We also investigated the effect that the implication class has on the topological distance between implication sites. By considering implications across multiple cycles, we expect to increase the distance of the potential implications we are considering. In Fig. 7, we show the average implication topological distance for the single and cross-cycle implications. Notice that, as expected, the cross-cycle implications tend to have a larger logical distance between the implication sites than implications contained within a single-cycle. Implications with larger distance have the potential to cover more errors and hence are likely to be more valuable when used for error detection.

### B. Evaluating Compression Rates

Including all of the implications in our checker logic would be prohibitively expensive. However, from the discussion in Section III, we do not expect all these implications to provide the same quality. Thus, the weak implications can be removed from our set without significantly hurting the overall fault detection rate using the procedures outlined in Section III-E.

The results of implication compression for combinational circuits are show in Table II. Column 2 shows the total number of ZChaff validated implications with topological distance greater than 2. In column 3, we report the number of valuable implications, which are the implications remaining after the weak implications are eliminated. Column 4 shows the number of implications left after subsequently running our heuristic measure of fault detectability. We call these implications prime implications. It is from these implications that the final implications to be included in the checker hardware will be chosen (subject to overhead constraints.) The compression rate (comparing columns 2 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 95.8%. This shows us that the vast majority of the naturally occurring implications have very little impact for error detection purposes.

TABLE III
NUMBER OF PRIME SEQUENTIAL IMPLICATIONS

| Circuit | First Cycle | Cross-Cycle | Second Cycle Only |
|---|---|---|---|
| s298 | 39 | 109 | 3 |
| s420 | 35 | 160 | 0 |
| s444 | 33 | 182 | 19 |
| s510 | 94 | 167 | 6 |
| s713 | 195 | 156 | 8 |
| s953 | 233 | 150 | 6 |
| s1196 | 449 | 44 | 2 |
| s1488 | 409 | 263 | 51 |

Similarly, Table III summarizes the results of the implication compression procedure on sequential circuits. Note that the third data column only includes those implications that were "newly" discovered in the second cycle and are not also present in a first cycle analysis. In general, we can get over an order of magnitude reduction in the number of implications in the checker logic by using this compression process. Also note that for all circuits, the cross-cycle implications remain an important component of the final implication list.

### C. Evaluating Fault Detection Rates

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.

1) *Case 1:* Fault occurs at internal node and is visible at primary output(s) (or latches). Implication checker flags a violation (True Detection).
2) *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).
3) *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).
4) *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 or latches. In other words, the checker circuitry was able to detect an observable, or *important* fault. For the second and the third scenarios, a

TABLE IV
DISTRIBUTION OF THE INTERACTION BETWEEN FAULTS AND INPUT PATTERNS FOR SEQUENTIAL CIRCUITS WITH *Prime Implications*

| Circuit | Case 1 | Case 2 | Case 3 | Case 4 |
|---|---|---|---|---|
| s298 | 21.6 | 17.5 | 50.7 | 10.1 |
| s420 | 15.9 | 11.6 | 61.9 | 10.6 |
| s444 | 22.0 | 21.8 | 50.8 | 5.4 |
| s510 | 28.1 | 11.3 | 54.7 | 5.9 |
| s713 | 20.6 | 9.4 | 57.7 | 12.3 |
| s953 | 24.8 | 8.6 | 61.5 | 5.2 |
| s1196 | 7 | 22.1 | 65.3 | 5.6 |
| s1488 | 9.4 | 7.2 | 78.5 | 5 |

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 was not sensitized by an appropriate input vector). The detection of such a fault by our checker hardware, as shown in case 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 nonproblematic 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.

Table IV reports the percentage of the time that an error will fall into each category for different benchmark circuits with all prime implications. While the circuits shown on the table are sequential, our experiments have shown us that combinational circuits yield a very similar distribution. The data presented is an average for all un-collapsed faults in each circuit when 32 000 random input patterns are applied. 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, 74% of the time a given fault does not affect the outputs.

Using the fault classification shown in Table IV, 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., case 1/[case 1 + case 4]). This metric is henceforth referred as the *Probability of Detection*. Results are shown in Fig. 8.

Our detection rates are relatively high for most circuits. However, we see that the implication hardware added to c499 detects only about 25% of the observable faults. This circuit implements a single-error-correction and an analysis of the

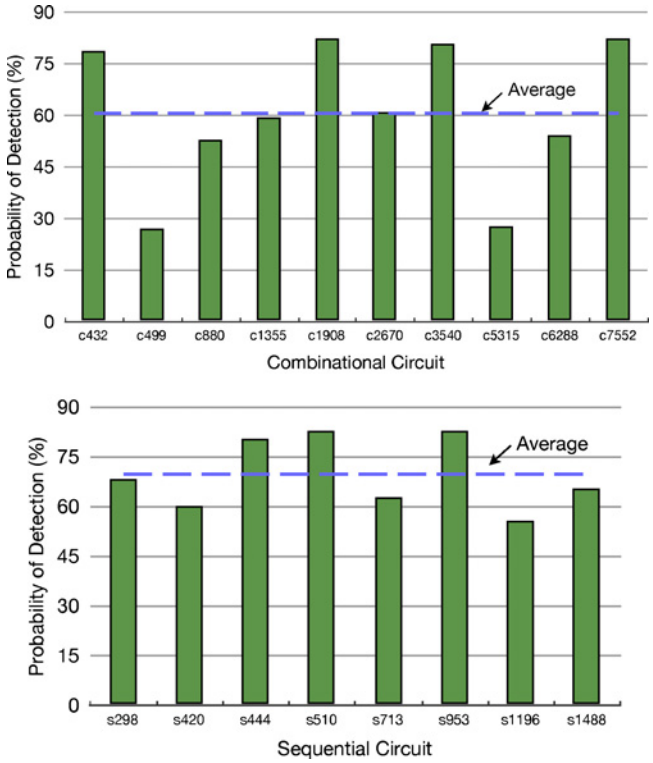


Fig. 8. *Probability of Detection* for combinational and sequential circuits with all *prime implications* and their respective average detection.

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 (and input B) must be at logic 1 and similarly a logic 0 at input B (or input A) 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 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 better compared to c499. In the case of c5315, we did not have a large number of high-quality prime implications to begin with. We suspect this is due to the large number of MUXes and XOR functions in the circuit. However, the detection is not as good as shown for some other benchmarks because the XOR structure still exists in abundance. By expanding our implication search procedure to identify more complex implications (i.e., those that consider relationships that exist when more than two circuit nodes are considered), we expect to be able to improve fault detection rates for circuits containing many XOR or MUX gates.

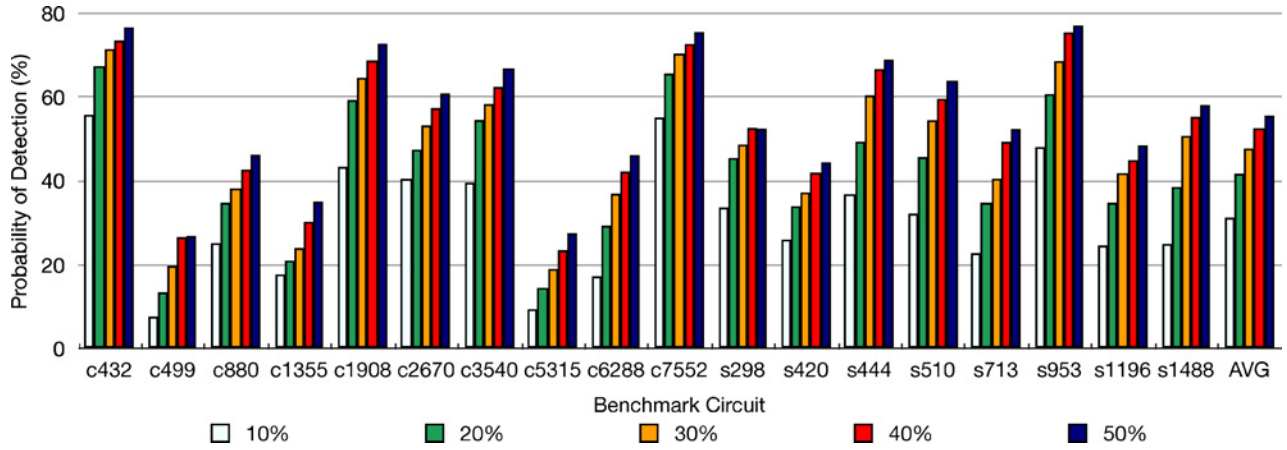


Fig. 9. Average error coverage achieved for different area overhead thresholds.

### D. Hardware Implementation Overhead

Up to this point, we have analyzed the error coverage that can be achieved using all the prime implications found after we apply our heuristic measure of fault detectability (i.e., those from the fourth column in Table II), 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.

If we choose to include a particular implication in the error checking hardware, in its simplest implementation, this translates into including a single gate in the checker hardware for each implication (for instance, the additional AND gate shown in gray in Fig. 2 raises a flag if N4 = 1 AND N24 = 1 occurs due to an error in the circuit and thus violates the implication). For the multicycle implication $B_1 = 0 \Rightarrow F_2 = 0$ from Fig. 4(a), we would need to save the value of $B$ in a latch and AND its complement with $F$ in the original sequential circuit, as shown in Fig. 4(c). This implication would be able to detect faults in both clock cycles. Ultimately, the results of these and other individual implication signals can be OR'ed into a single error signal (possibly using a wired OR) which can then trigger an appropriate error recovery mechanism.

To determine the true hardware overhead when all nonsubsumed implications are included in the checker logic, checkers like the one described above were created for each benchmark circuit using a Taiwan Semiconductor Manufacturing Company (TSMC) 180 nm library and Mentor Graphics Toolset.

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. For these experiments, we approximated hardware overhead as the 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. This allows implications to be chosen without explicitly generating the layout. However, once a layout is actually generated, the exact overhead may vary somewhat due to routing and other issues. On average, the actual overhead we get is often relatively close, with our actual average overhead being 12.6% when 10% overhead was targeted and 57.5% when 50% overhead was targeted. Using this method, even for small overheads, we get a good *Probability of Detection* as well as a natural decrease in power and delay.

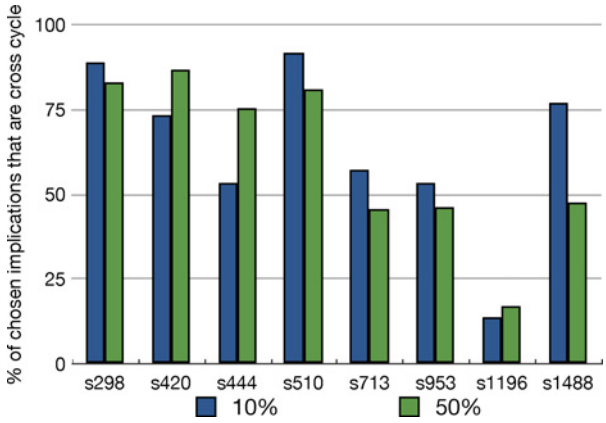


Fig. 10. Percentage of cross-cycle implications for two area overhead thresholds.

These favorable results are natural throughout our implication checker no matter the overhead. If we compare the whole set of sorted implications against 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 [25]. The ability to have a smaller area is an added advantage that those looking for some added stability which is cost effective. With this in mind, we next used the procedure described in Section III-F, to determine how one could maximize the probability of detecting an error, given a user-specified limit on the hardware overhead of the checker circuitry.

In the case of sequential circuits, because cross-cycle implications require an additional flip-flop to hold circuit site values from one clock-cycle to the next, they are more expensive than single-cycle implications. Analysis with the Mentor Graphics layout tool, ICStation, shows that in general the insertion of a cross-cycle implication is approximately twice as expensive in terms of area as a single-cycle implication when both standard

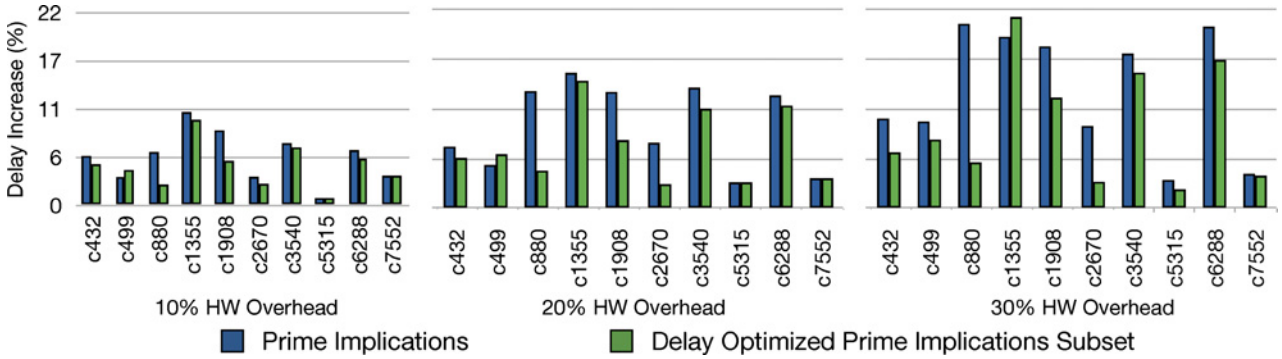


Fig. 11. Critical path delay when *prime implications* are selected with and without delay constraints. Results are relative to an *unmodified* circuit.

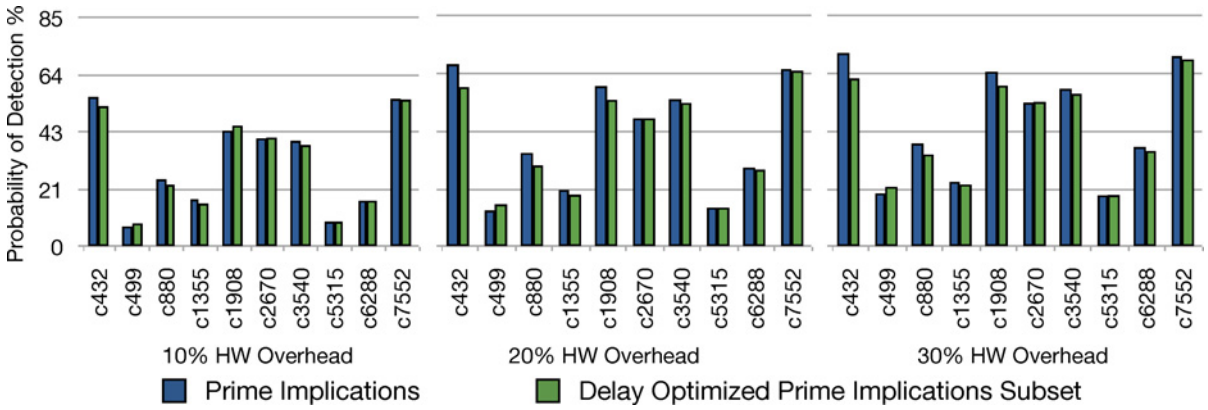


Fig. 12. *Probability of Detection* when *prime implications* are selected with and without delay constraints.

cell area and routing are considered. Similar area constrained experiments were run for the sequential circuits.

The results in Fig. 9 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. Note that, even when the checker is restricted to low-area overheads, the final error rate for the circuit will still generally be very small. In addition, we see that the benefits from increasing area overhead results in diminishing amounts of coverage; for many of the circuits, going beyond 20% area overhead does not provide a significant additional advantage.

Finally, Fig. 10 shows the percentage cross-cycle implications making up both the 10% and 50% overhead experiments. Even with the additional cost of cross-cycle implications, their superior error detection capability ensures that they play a significant role in maximizing error coverage.

### *E. Improving Critical Path Delay*

While implications provide an efficient means for online error detection, if an implication includes a node on a critical path, its inclusion in the checker logic can increase the circuit's delay since capacitive load on the node will increase. Therefore, we need to select a new subset of implications that will maintain a reasonable probability of error detection, while preventing drastic delay increases. The algorithm to select this new subset of implications, optimized for delay, was described in Section III-F. By applying this algorithm, we found that, on average, only 11% of the implications in the checker logic contained nodes on a critical path when using a 10% area overhead. For 20% and 30% area overhead, critical path implications accounted for only 9% and 8% of the checker logic implications, respectively.

Fig. 11 shows the increased delay of the circuit with and without delay constraints added to the implication selection process, compared against an unmodified circuit without any checker logic. For the different hardware overheads it can be clearly seen that there is a consistent reduced delay when the new implication subset is used on all circuits but c499. We believe that the delay increase on this particular circuit is related to its inherent structure. By limiting implications on the original critical path, our algorithm selected implications that created a new critical path. However, this particular case is not worrisome; on an absolute scale, circuit c499 has the second smallest added delay. Regardless of this outlier, on average, by choosing this new delay-aware implication subset, we were able to reduce the delay by 2.6%.

Since this new implication subset was selected by pruning the set of implications that was optimized for error detection, a reduction on the *Probability of Detection* was expected. The results in Fig. 12 show that the average reduction in the *Probability of Detection* was only 1.7%. It is noteworthy that several circuits, such as c6288 and c2670, experienced very little, or no, reduction in their *Probability of Detection*. We believe this is a direct consequence of our heuristic approach that selects prime implications (see Section III-E). Note that we are generating a strict order for our prime implications based on our heuristic measure of fault detection quality.

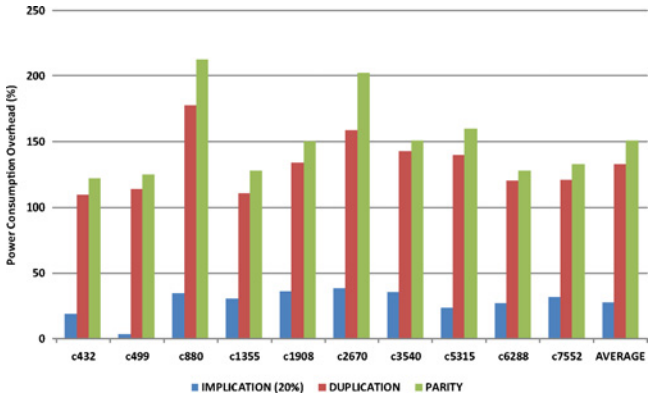


Fig. 13. Comparing the power overhead of adding 20% implication hardware, logic duplication and parity with respect to the original unmodified circuit.

In reality, there may not be a significant difference in fault detection ability among implications that are ordered relatively close in this sorted list.

In conclusion, Fig. 12 shows us that it is possible to apply a selection criterion that will reduce the critical path delay while keeping a similar *Probability of Detection*. For example, circuit c2670 experienced a reduced delay impact of 3.9% with an equivalent *Probability of Detection*.

### *F. Power Requirements*

The addition of any new circuitry will cause an increase in both dynamic and leakage power. In this section, we compare the power overhead of our implication based approach to that of logic duplication and simple parity implementations. We use the circuits from Figs. 11 and 12 with 20% hardware overhead and delay optimized for comparison. A 20% overhead was chosen because, in many circuits, this appears to provide a good tradeoff between fault detection and area overhead. Specifically, a significant increase in fault detection often occurs between 10% and 20% while the increase from 20% to 30% is much smaller. In the case of logic duplication, the original circuit was copied, and the checker circuitry consisted of an XOR of each output with its copy. The XORs were ORed together to create a single error signal. For the circuit with parity protection, the parity prediction circuitry was optimized for area, and the parity of the outputs was calculated and compared to the predicted parity to create a single error signal. 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. Using this scheme, we found the probability of error detection to be 73%, on average, for these circuits.

The verilog netlist of all circuits were imported into the Mentor Graphics toolset and a SPICE netlist was extracted using Mentor's Design Architect and ICStation tools. The netlist was simulated using the TSMC 180 nm model and the Mentor Graphics MACH-PA power analysis tool. A total of 1000 random patterns were generated for each input and the average power consumption measured over a period of 100 ms using a supply voltage of 1.8-V. Fig. 13 shows a summary of the power overhead of adding additional hardware. Results show that the addition of 20% implication hardware results in an average power dissipation increase of 28%. The power dissipation overhead was the lowest for c499 (about 3.5%) and was around 30% for the larger benchmark circuits. The average power dissipation overhead for logic duplication and parity were 133% and 151% respectively. The higher power overheads due to logic duplication and parity is seen mostly in the four arithmetic logic unit benchmarks (c880, c2670, c3540, and c5315).

From these results, we can conclude that very high-error detection rates can come at a high price. If such high overheads in power cannot be afforded for a given design, then our approach offers an easy way of trading off some error coverage for significant reductions in power dissipation.

## V. Conclusion and Future Work

We have presented an online error detection technique for both combinational and sequential circuits based on the use of logic implication information. These implications may exist within a single clock-cycle or across multiple clock cycles, and the violation of an implication indicates the presence of an error. A key advantage of our approach is that we automatically identify these implications without requiring any knowledge of the high-level circuit behavior. It is also particularly amenable to enabling fine-grained tradeoffs between fault detection capability and area overhead. Finally, it does not require resynthesis of the logic being protected; the checker logic is run in parallel with the regular control logic.

Because many more implications exist than can be reasonably included in the checker hardware, we have presented heuristic techniques for identifying the most valuable implications that achieve the greatest fault detection. The number of implications included in the checker logic can be easily limited to satisfy user-defined maximum area overhead constraints. For most circuits, we found that significant increases in fault detection capabilities occurred until an area overhead of 20% was reached. Adding additional implications provided additional error detection but at a slower rate.

We have also studied the impact of the implication hardware on power and delay. We have presented a methodology for algorithmically selecting an effective subset of the prime implications that minimizes the added delay. Although the reduction in added delay varies from circuit to circuit, for some circuits, such as c1908, dramatic reductions in the additional delay were achieved with very little impact on fault detection.

In the future, we would like to investigate the potential added benefits of identifying implications that exist across three or more cycles. In addition, we would like to identify and use more complex implications (i.e., those that consider relationships that exist when more than two circuit nodes/sites are considered). These implications may be particularly good for detecting faults within circuits containing many XOR gates. Finally, we intend to extend this paper to explicitly target delay faults and additional power reduction. For example, multicycle implications, in addition to giving us powerful implications for the detection of otherwise hard-to-detect errors near latch boundaries, also have the potential to be very effective for

detecting some types of delay faults. 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. Appropriate cross-cycle implications that include this delay path will be able to detect this delay-induced error without any complicated timing or clock-gating needed for capturing the checker results.

## REFERENCES


[1] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: Automatic generation of simulation checkers from formal specifications," in *Proc. Comput.-Aided Verificat. (CAV)*, 2000, pp. 538–542.

[2] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proc. Int. Soc. Quality Electron. Design (ISQED)*, 2007, pp. 613–620.

[3] M. Choi, N. Park, F. Lombardi, Y. Kim, and V. Piuri, "Balanced redundancy utilization in embedded memory cores for dependable systems," in *Proc. Int. Symp. Defect Fault Tolerance Very-Large-Scale Integrat. Syst.*, Dec. 2002, pp. 419–427.

[4] D. K. Bhavsar, "An algorithm for row-column self-repair of RAMS and its implementation in the Alpha 21264," in *Proc. Int. Test Conf. (ITC)*, 1999, pp. 311–318.

[5] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proc. Int. Symp. Microarchitecture*, 2001, pp. 214–244.

[6] S. K. Reinhardt and S. S. Mukherjee, "Transient-fault detection via simultaneous multithreading," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, Jun. 2000, pp. 25–36.

[7] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream processors: Improving both performance and fault-tolerance," in *Proc. Architectural Support Programming Languages Operating Syst. (ASPLOS)*, Nov. 2000, pp. 257–268.

[8] T. M. Austin, "Diva: A reliable substrate for deep-submicron microarchitecture design," in *Proc. Int. Symp. Microarchitecture*, Nov. 1999, pp. 196–207.

[9] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic transient-fault detection," in *Proc. Int. Symp. Comput. Architecture*, Jun. 2005, pp. 172–183.

[10] C. F. Webb and J. S. Liptay, "A high-frequency custom CMOS S/390 microprocessor," *IBM J. Res. Develop.*, vol. 41, nos. 4–5, pp. 463–473, 1997.

[11] R. Sedmak and H. Liebergot, "Fault tolerance of a general purpose computer implemented by very large scale integration," *IEEE Trans. Comput.*, vol. 29, no. 6, pp. 492–500, Jun. 1980.

[12] J. von Neumann, "Probabilistic logics and synthesis of reliable organisms from unreliable components," in *Automata Studies*. Princeton, NJ: Princeton Univ. Press, 1956, pp. 43–98.

[13] P. K. Samudrala, J. Ramos, and S. Katkoori, "Selective triple modular redundancy (STMR) based single-event upset (SEU) tolerant synthesis for FPGAs," *IEEE Trans. Nucl. Sci.*, vol. 51, no. 5, pp. 2957–2969, Oct., 2004.

[14] K. Mohanram and N. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *Proc. Int. Test Conf. (ITC)*, Oct. 2003, pp. 893–901.

[15] K. Mohanram, E. Sogomonyan, M. Gossel, and N. Touba, "Synthesis of low-cost parity-based partially self-checking circuits," in *Proc. On-Line Testing Symp.*, 2003, pp. 35–40.

[16] E. S. Sogomonjan and M. Gössel, "Design of self-parity combinational circuits for self-testing and on-line detection," in *Proc. IEEE Int. Workshop Defect Fault Tolerance Very-Large-Scale Integrat. Syst.*, 1993, pp. 239–246.

[17] M. Gössel and S. Graf, *Error Detection Circuits*. New York: McGraw-Hill, 1993.

[18] S. Almukhaizim, P. Drineas, and Y. Makris, "Cost-driven selection of parity trees," in *Proc. Very-Large-Scale Integrat. Test Symp (VTS)*, Apr. 25–29, 2004, pp. 319–324.

[19] S. Ghosh, N. Touba, and S. Basu, "Synthesis of low power CED circuits based on parity codes," in *Proc. Very-Large-Scale Integrat. Test Symp (VTS)*, 2005, pp. 315–320.

[20] K. De, C. Natarajan, D. Nair, and P. Banerjee, "RSYN: A system for automated synthesis of reliable multilevel circuits," *IEEE Trans. on Very-Large-Scale Integrat. Syst.*, vol. 2, no. 2, pp. 786–195, Jun. 1994.

[21] N. Touba and E. McCluskey, "Logic synthesis of multilevel circuits with concurrent error detection," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 16, no. 7, pp. 783–789, Jul. 1997.

[22] J. M. Berger, "A note on an error detection code for asymmetric channels," *Informat. Control*, vol. 4, pp. 68–73, Mar. 1961.

[23] B. Bose and D. J. Lin, "Systematic unidirectional error-detecting codes," *IEEE Trans. Comput.*, vol. 34, no. 11, pp. 1026–1032, Nov. 1985.

[24] D. Das and N. A. Touba, "Synthesis of circuits with low-cost concurrent error detection based on Bose–Lin codes," in *Proc. Very-Large-Scale Integrat. Test Symp (VTS)*, 1998, pp. 309–315.

[25] S. Mitra and E. McCluskey, "Which concurrent error detection scheme to choose?" in *Proc. Int. Test Conf. (ITC)*, Oct. 2000, pp. 985–994.

[26] R. Sharma and K. Saluja, "An implementation and analysis of a concurrent built-in self-test technique," in *Proc. Int. Symp. Fault-Tolerant Comput.*, Jun. 1988, pp. 164–169.

[27] P. Drineas and Y. Makris, "Concurrent fault detection in random combinational logic," in *Proc. Int. Soc. Quality Electron. Design (ISQED)*, Mar. 2003, pp. 425–430.

[28] R. Drechsler, "Synthesizing checkers for on-line verification of system-on-chip designs," in *Proc. Int. Symp. Circuits Syst. (ISCAS)*, May 25–28, 2003, pp. 748–751.

[29] S. Das, R. Mohanty, P. Dasgupta, and P. Chakrabarti, "Synthesis of system verilog assertions," in *Proc. Design, Automat. Test Eur.*, vol. 2. Mar. 2006, pp. 1–6.

[30] F. M. Brown, *Boolean Reasoning: The Logic of Boolean Equations*. Boston, MA: Kluwer, 1990.

[31] L. Entrena, E. Olías, J. Uceda, and J. Espejo, "Timing optimization by an improved redundancy addition and removal technique," in *Proc. Eur. Design Automat. Conf. (EURO-DAC)*, 1996, pp. 342–347.

[32] W. Kunz and D. K. Pradhan, "Recursive learning: A new implication technique for efficient solutions to CAD problems: Test, verification, and optimization," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 13, no. 9, pp. 1143–1158, Sep. 1994.

[33] J. Zhao, E. M. Rudnick, and J. H. Patel, "Static logic implication with application to redundancy identification," in *Proc. IEEE Very-Large-Scale Integrat. Test Symp. (VTS)*, 1997, pp. 288–293.

[34] K. Gulrajani and M. S. Hsiao, "Multinode static logic implications for redundancy identification," in *Proc. Design, Automat. Test Eur. (DATE)*, 2000, pp. 729–735.

[35] M. A. Iyer and M. Abramovici, "Fire: A fault-independent combinational redundancy identification algorithm," *IEEE Trans. Very-Large-Scale Integrat. Systems*, vol. 4, no. 2, pp. 295–301, Jun. 1996.

[36] R. Bahar, M. Burns, G. Hachtel, E. Macii, H. Shin, and F. Somenzi, "Symbolic computation of logic implications for technology-dependent low-power synthesis," in *Proc. Int. Symp. Low Power Electron. Design (ISLPED)*, Aug. 1996, pp. 163–168.

[37] W. Kunz and P. Menon, "Multilevel logic optimization by implication analysis," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 1994, pp. 6–10.

[38] B. Rohfleisch, A. Kölbl, and B. Wurth, "Reducing power dissipation after technology mapping by structural transformations," in *Proc. Design Automat. Conf. (DAC)*, 1996, pp. 789–794.

[39] N. Alves, K. Nepal, J. Dworak, and R. I. Bahar, "Compacting test vector sets via strategic use of implications," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 2009, pp. 791–796.

[40] S. Bobba and I. Hajj, "Estimation of maximum current envelope for power bus analysis and design," in *Proc. Int. Symp. Phys. Design (ISPD)*, Apr. 1998, pp. 141–146.

[41] A. Glebov, S. Gavrilov, D. Blaauw, and V. Zolotov, "False-noise analysis using logic implications," *ACM Trans. Design Automat. Electron. Syst.*, vol. 7, no. 3, pp. 474–498, Jul. 2002.

[42] P. Tafertshofer, A. Ganz, and M. Henftling, "A SAT-based implication engine for efficient ATPG, equivalence checking, and optimization of netlists," in *Proc. Int. Conf. Comput.-Aided Design (ICCAD)*, Nov. 1997, pp. 648–655.

[43] S. Almukhaizim and Y. Makris, "Soft error mitigation through selective addition of functionally redundant wires," *IEEE Trans. Reliability*, vol. 57, no. 1, pp. 23–31, Mar. 2008.

[44] S. Krishnaswamy, S. M. Plaza, I. L. Markov, and J. P. Hayes, "Signature-based SER analysis and design of logic circuits," *IEEE Trans. Comput.-Aided Design Integr. Circuits*, vol. 28, no. 1, pp. 74–86, Jan. 2009.

[45] R. Vemu, A. Jas, J. Abraham, S. Patil, and R. Galivanche, "A low-cost concurrent error detection technique for processor control logic," in *Proc. Design, Automat. Test Eur. (DATE)*, Mar. 2008, pp. 897–902.

[46] I. Pomeranz and S. M. Reddy, "Reducing fault latency in concurrent on-line testing by using checking functions over internal lines," in *Proc. Int.*

*Symp. Defect Fault Tolerance (DFT) Very-Large-Scale Integrat. Syst.*, 2004, pp. 183–190.

[47] Y. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient SAT solver," in *Lecture Notes in Computer Science*. Berlin, Germany: Springer, vol. 3542. 2005, pp. 360–375.

[48] S. Manich and J. Figueras, "Maximizing the weighted switching activity in combinational cmos circuits under the variable delay model," in *Proc. Eur. Design Test Conf.*, Mar. 1997, pp. 597–602.

**Nuno Alves** (S'08) received the B.S. in computer systems engineering from Boston University, Boston, MA, in 2002 and a Sc.M. in engineering from Brown University, Providence, in 2008. He is currently working toward the Ph.D. degree in electrical engineering at the Division of Engineering, Brown University, Providence, under the supervision of Dr. R. I. Bahar.

His current research interests are centered on reliable computing for emerging very-large-scale integration technologies, with special focus in concurrent error detection methods. His work is being sponsored by the Portuguese government via the program from Fundação para a Ciência e a Tecnologia.

**Alison Buben** is currently an undergraduate student with the Department of Computer Science, Indiana University of Pennsylvania, Indiana, where she is pursuing the B.S. degree in computer science.

In 2009, she was selected to participate in the CDC/CRA-W Distributed Research Experiences for Undergraduates program under the mentorship of Dr. R. I. Bahar. She is expecting to graduate in May 2011.

**Kundan Nepal** (S'04–M'07) received the B.S. degree in electrical engineering from Trinity College, Hartford, CN, in 2002, the M.S. degree in electrical engineering from the University of Southern California, Los Angeles, CA, in 2003, and the Ph.D. degree in electrical and computer engineering from Brown University, Providence, RI, in 2007.

Since 2007, he has been an Assistant Professor with the Department of Electrical Engineering, Bucknell University College of Engineering, Lewisburg, PA. His current research interests include defect/fault tolerant circuits and systems; nanometer digital very-large-scale integration system design and reconfigurable computing.

**Jennifer Dworak** (S'95–M'05) received the B.S. (summa cum laude) and M.S. degrees in electrical engineering from Texas A&M University, College Station, in 1998 and 2000, respectively. She received the Ph.D. degree in electrical engineering from Texas A&M University in 2004, under the supervision of Prof. M. R. Mercer.

Since 2005, she has been an Assistant Professor with the Division of Engineering, Brown University, Providence, RI. Her research interests include digital circuit testing and automatic test pattern generation, defective part level modeling, online error detection, design verification, and trust in integrated circuits.

Dr. Dworak received an National Science Foundation Graduate Fellowship and was a Best Paper Award recipient for a paper appearing in the 1999 Very-Large-Scale Integration Test Symposium. She was also the winner of the Best Student Presentation Award at the 2002 International Test Synthesis Workshop and a winner of the 2004 Test Technology Technical Council Naveena Nagi Award. She was general chair of the 2009 International Test Synthesis Workshop and is Co-Program Chair for the 2010 North Atlantic Test Workshop.

**R. Iris Bahar** (S'93–M'96) received the B.S. and M.S. degrees in computer engineering from the University of Illinois at Urbana-Champaign, Champaign, in 1986 and 1987, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Colorado, Boulder, in 1995.

From 1987 to 1992, she was with Digital Equipment Corporation, Hudson, MA, working on microprocessor hardware design. Since 1996, she has been with the Division of Engineering, Brown University, Providence, RI, where she is currently an Associate Professor. Her current research interests include computer architecture; computer-aided design for synthesis, verification, and low-power applications; and design, test, and reliability issues for nanoscale systems.

Dr. Bahar is currently an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS and the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS.