Enhancing Online Error Detection through Area-Efficient Multi-Site Implications

N. Alves*, Y. Shi*, J. Dworak††, R. I. Bahar*, K. Nepal†
*School of Engineering, Brown University, Providence, RI 02906
††Department of Computer Science and Engineering, Southern Methodist University, Dallas, TX 75205
†Electrical Engineering Department, Bucknell University, Lewisburg, PA 17837

***Abstract*—We present a new method to identify multi-site implications that can significantly increase the fault coverage of error-detecting hardware without increasing the area overhead. This method intelligently divides the input space about the functions of internal circuit sites and finds new valuable implications that can share gates in checker logic.**

I. Introduction

Over the last several decades, aggressive scaling of integrated circuits has lead to higher packing densities, increased circuit performance, and lower production costs. Unfortunately, these reductions in feature size also mean that circuits and microprocessors are becoming more susceptible to errors arising from multiple sources, including excessive process variations, defects, wearout, and operational and environmental influences.

Online error detection aims to monitor circuit behavior at run time and detect deviations from its normal operating behavior while the device is in operation. The ideal online error detection scheme would detect all errors without requiring significant circuit modifications and would not negatively impact performance, power, or area overhead. Unfortunately, no scheme is ideal, and multiple tradeoffs need to be taken into consideration when choosing a particular implementation.

In the past, we have proposed a method for using the logic implications that occur naturally in circuits to provide varying degrees of coverage for online errors and reported its error-coverage/power/delay benefits when compared against other approaches, such as parity and logic duplication [1]. These logic implications can provide valuable reductions in the error rate while allowing the hardware overhead of the checker logic to be easily constrained within a desired hardware budget.

However, while the simple two-site implications we have previously studied can provide very good coverage of some errors, other errors may not be detectable by any *simple implication*. Expanding the implication set to include implications that involve additional circuit sites has the potential to cover some of these errors; however, the number of multi-site implications we could possibly consider is enormous, and this makes the identification and selection of such implications nontrivial. Intelligent methods are necessary to quickly identify new implications that have a reasonable chance of providing good error coverage.

In this paper, we propose a novel method for identifying such multi-site implications. In this method we divide the input space according to the functions realized by particular internal circuit sites and identify new *residual implications* that are valid in each portion of the space. The gating hardware that determines when a particular implication is valid can then be shared across multiple implications, significantly reducing the cost of the multi-site approach. We will show that these new *residual implications* can significantly increase error coverage for a constant hardware budget when combined with both the *simple implications* we described in [1] and a form of *checking functions* similar to that proposed in [2]. Furthermore, we will show that these new *residual implications* are especially useful when protecting circuitry that has been optimized for area and/or delay.

II. Background

For over half a century, a wealth of techniques have been proposed and analyzed for the detection and correction of errors at run time. Some of these methods include coding techniques such as parity, Berger, and Bose Lin codes (e.g. [3]), executing portions of the code in redundant threads or in temporarily unused functional units (e.g. [4], [5]), and duplicating or tripling at least some of the circuit logic (e.g. [6]–[9]). Other online error detection schemes include Built-In Concurrent Self Test (BICST) [10] and Reduced Observation Width Replication (ROWR) [11] where prediction hardware is added to guess the appropriate response to a set of pre-computed test vectors. In addition, high-level functional assertions, such as those identified during functional verification, may also be hardcoded into the design to signal errors (e.g., [12]).

Other researchers have considered relationships that occur between flip-flops or between circuit sites at the gate level. For example, the authors of [13] investigated the protection of the control logic of a microprocessor though the use of relationships between functions of the flip-flops in a design. The authors of [2] proposed the use of checking functions, which identify circuit sites or functions of circuit sites that should always be equal to (or complements of) each other.

In the past, we have also investigated the use of relationships between sites at the gate level (i.e., logic implications) to identify internal circuit errors [1]. Logic implications arise naturally in digital circuits, and in their simplest form, they can be expressed in the following format:

$$\mathit{site}A = x \rightarrow \mathit{site}B = y, \text{ where } \{x, y\} \in \{0, 1\}$$

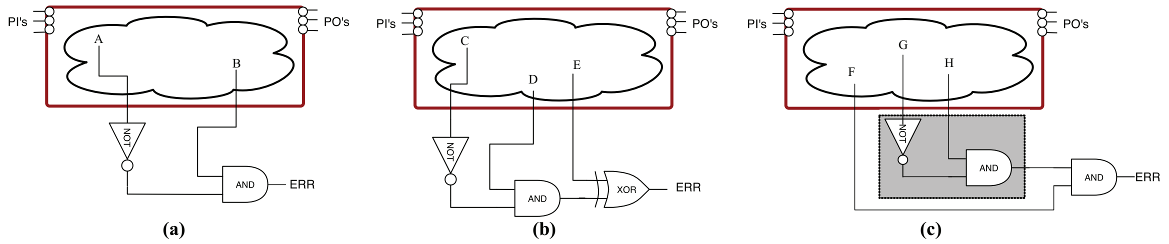

Fig. 1: Examples of error-detecting implications and checking functions implemented in parallel to the main logic circuitry. (a) A simple implication: $\textit{site}(A) = 0 \to \textit{site}(B) = 0$, (b) a checking function: $\overline{\textit{site}(C)} \cdot \textit{site}(D) \equiv \textit{site}(E)$, and (c) a residual implication: $\textit{site}(F) = 1 \to (\textit{site}(G) = 0 \to \textit{site}(H) = 0)$.

For example, in Figure 1(a), we show the simple implication $siteA = 0 \to siteB = 0$. In this case, checker logic can be inserted to flag an error should *siteA* be set to logic zero and *siteB* be set to a logic one.

However, while such implications have proven to be useful for online error detection, they also have several limitations. Specifically, each implication is generally only able to capture errors of a single polarity at a given site. Furthermore, it is often difficult to detect errors close to the primary outputs because we essentially "run out of logic" in which the implications can occur. Finally, it is also possible for no simple two-site implication to exist that can detect a particular error—especially when a circuit has been optimized to minimize area and/or delay.

Alternatively, the checking functions proposed in [2] may be used to detect errors with either polarity. However, they are dependent on the appearance of equalities between sites or functional combinations of sites naturally occurring in circuits. Unfortunately, many of these equality relationships are likely to be removed during logic minimization and thus checking functions may be less effective or prevalent in optimized circuits.

To address these limitations of previous methods, in this paper, we introduce a new method that efficiently identifies valuable multi-site logic implications. We will demonstrate that these new residual implications are capable of detecting many errors not covered by the approaches of [1], [2]. Furthermore, this new methodology is especially necessary when protecting circuits optimized for reduced area or delay.

III. Methodology

To detect errors not covered by simple two-site implications, additional relationships may be harnessed if more complex combinations of circuit sites are considered during the implication discovery process. In its most general form, a multi-site implication will encapsulate a relationship where some Boolean function of the values at a particular set of circuit sites *implies* the value of some Boolean function of other circuit sites. Thus, we have:

$$f(X_1, X_2, ...X_n) \to g(Y_1, Y_2, ...Y_m) \tag{1}$$

where $X_1, ...X_n$ and $Y_1, ...Y_m$ are sites in the circuit and f and g Boolean logic functions.

Of course, the number of such implications grows much too quickly to be considered exhaustively even for very small circuits. Furthermore, the need to realize the functions f and g with logic gates means that multi-site implications are inherently more expensive in terms of hardware overhead than two-site implications. As a result, f and g should be kept as small as possible. In this paper, we choose to replace the function g with the value of a single site. Thus, all multi-site implications considered are of the form:

$$f(X_1, X_2, ...X_n) \to Y_1 \tag{2}$$

However, we still must determine an effective way of finding good choices for the function f that will increase coverage with low hardware overhead.

A. Residual implications

Intuitively, one of the problems we face when discovering logic implications is the fact that an implication must be valid for all possible input combinations. circuit, we cannot use it. However, faults are only detectable for a subset of all possible patterns. If we divide the input space, it is often possible to find additional implications that are valid in the subspace that were not valid in the space as a whole.

Shannon's Expansion Theorem provides a means of decomposing a logic function into two pieces about a particular input variable. Thus, the logic function $f(x_1, x_2, ...x_i, ...x_n)$ can be written in the form:

$$f = x_i' f_{x_i'} + x_i f_{x_i} \tag{3}$$

Here $f_{x_i'}$ is the x_i' *residue* of f and is the function f when x_i is equal to 0. Similarly, f_{x_i} is the x_i residue and is the function f when x_i is equal to 1.

We apply this concept to the search for logic implications by choosing a site P in the circuit about which to divide the input space. (Note that in our implementation P does not need to be an input itself.) Specifically, we divide the input space into the part where P is equal to 0 and the part where P is equal to 1. Then, we find implications that are valid when P is equal to 0 as well as implications that are valid when P is equal to 1.

Because these implications are only valid in part of the input space, the error signal must be gated by the signal P in the checker logic. Specifically, an implication that is only valid when P is equal to 1 must be AND'ed with P and an implication that is only valid when P is equal to 0 must be AND'ed with P'. In the simplest form, this essentially produces a three-site implication consisting of the two sites that form a simple implication as well as the site P that determines when the implication should be guaranteed to be valid.

For example, Figure 1(c) demonstrates how residual implications can be realized within the checker logic. The dashed box contains the simple implication ($\textit{site}(G) = 0 \rightarrow \textit{site}(H) = 0$), which is only valid when the residual element $\textit{site}(F) = 1$.

Generating multi-site implications in this way has several advantages. First, multiple implications that are all valid under the same conditions can share a single AND gate with P or P', thus minimizing the additional overhead required. Furthermore, if we can find appropriate choices of the site P, the amount of effort required to find and evaluate possible multi-site implications is much less than in the general case.

B. Our approach

Our approach to extend online error detection using logic implications can be summarized in the following steps:

1) Perform good circuit simulation on a gate level netlist of the circuit. Find all potential 2-site logic implications, validate them for all input combinations with a SAT solver, and remove all "weak" implications from further consideration via a compression algorithm. This step follows the flow presented in [1].
2) Find checking functions that are present in the circuit, using an approach similar to step 1.
3) Perform fault simulation on the two sets of logic relationships found in steps 1 and 2 and sort the elements into a single ordered list according to error coverage capability.
4) Determine optimal splitting points, calculate a set of residual implications, and calculate their error detection capability.
5) Select an appropriate subset of the identified implications and/or checking functions subject to the desired hardware budget.

While the execution time is heavily dependent on circuit-size and functionality, for the selected benchmarks, we were able to compute the outcome of each step in a few minutes. The remainder of this section describes these steps in detail.

1) Finding simple logic implications

Our implication discovery process follows the algorithm presented in [1] and begins with good circuit simulation of the netlist using a subset of input vectors. By performing pairwise compares between all circuit sites, simple two-site logic implications are extracted and then subsequently validated using a SAT solver. This list of valid implications is then compressed using the algorithms discussed in [1], and from this compressed list implications can be evaluated and ultimately selected to form part of the checker logic hardware.

2) Finding checking functions

In our implementation, checking functions like those described in [2], [14], are discovered by:

1) Comparing every pair of sites and determining if they always have equal (or complemented) logic values.
2) Determining, at each site g, all sites that are within a 10 site radius and finding all *equality functions* between every two internal sites, i and j in that radius. These equality functions are of the form $g = h_i^* \; OP \; h_j^*$, where OP $\in$ {AND, OR, XOR} and where h^* is the function realized at a site in either its complemented or uncomplemented state.

Figure 1(b) shows the checker logic implementation of a checking function where a simple relationship between two sites, C and D, equals a third site E, for all input vectors. We discover our possible checking functions from logic simulation values, and then proceed to validate these checking functions with a SAT solver.

3) Evaluating implications and checking functions

Once we have generated the two sets of logic relationships from steps 1) and 2), we need to create a single ordered list that ranks all elements within the sets by their ability to detect errors. In essence, we want an implication or checking function to be closer to the top of the ordered list when it is better at detecting errors than the implications or checking functions below it. Information regarding faults left uncovered by both simple implications and checking functions will subsequently be used to identify our new residual implications.

The procedure begins by including, one at a time, all implications and checker functions found in steps 1 and 2 in the checker logic for the circuit under test. Fault simulation using the stuck-at-fault model and 32k random input vectors is performed. Only stuck-at faults were used in our experiments as they are well-known, easy to model, and often used in related work. Fortunately, other error sources may cause behavior similar to stuck-at faults on the cycle when they are present. By detecting stuck-at faults effectively, we may be able to fortuitously detect these other types of errors. From this, we determine which checker logic elements are able to detect which faults for each vector, and how many times each fault causes an error at the circuit outputs. Once all faults are processed, we sort the checker logic elements in descending order according to number of fault detections with which they are associated and dynamically removed duplicated fault detections during the sorting. This procedure does not explicitly favor one fault over another; its sole purpose is to quickly process data and disregard redundant checker logic elements with lower fault coverage capabilities than other checker logic elements. These ideas can be implemented with the optimization algorithm described in [1].

4) Finding residual implications

The search for residual implications entails selecting a particular *splitting site* (P), fixing its logic state, and recomputing all implications using the same method as described in step 1). The newly generated implications that are not in the list of simple implications, are called residual implications.

ISCAS85					ITC99				
Circuit	PI	PO	Gates	Faults	Circuit	PI	PO	Gates	Faults
c432	36	7	160	864	b04C	77	74	698	3022
c499	41	32	202	998	b05C	35	70	855	4490
c880	60	26	383	1760	b06C	11	15	58	214
c1355	41	32	546	2710	b07C	50	57	429	1868
c1908	33	25	880	3816	b08C	30	25	178	768
c2670	233	140	1269	5340	b09C	29	29	174	702
c3540	50	22	1669	7080	b10C	28	23	191	878
c5315	178	123	2406	10630	b11C	38	37	653	3242
c6288	32	32	2406	12576	b12C	126	127	1084	4924
c7552	207	108	3512	15104	b13C	63	63	363	1426

TABLE I: Structural benchmark characteristics

Selecting a good set of splitting sites is of utmost importance, as they will dictate the quality of the subsequently generated residual implications. This selection is done with the information collected in step 3 after performing fault simulation with the simple 2-site implications. We select the splitting sites to be those sites whose associated faults are propagated the most and whose errors are not often detected by any simple implications. The vast majority of the splitting sites are located at the circuit inputs, or at sites very close to one.

5) *Selecting and determining checker logic performance*

To evaluate the quality of a particular checker logic set, we define a *probability of detection* metric. This probability quantifies how well a particular set of implications can detect online errors. It is calculated by initially performing stuck-at-fault simulation, with 32k random input vectors, and then counting how often an internal fault that is observable at at least one of the circuit's outputs is detected and flagged by the implication checker hardware. More formally, the probability of detection (P_{det}) is defined as,

$$P_{det} = \sum_{i=1}^{\#\text{faults}} \sum_{j=1}^{\#\text{vectors}} \left(\frac{E_{i,j}}{F_{i,j}} \right) \tag{4}$$

where, $F_{i,j}$=1 when fault i propagates to an output with vector j and 0 otherwise. Similarly, $E_{i,j}$=1 if the checker logic is violated when $F_{i,j}$=1 and 0 otherwise. This metric does not take into account faults occurring inside the checker logic.

IV. Results

In this section we report the properties of the various checker logic implementations and their ability to detect faults. To determine the effectiveness of each implementation, we studied 20 circuits from two distinct benchmark suites, ISCAS85 [15] and ITC99 [16]. For the ITC99 circuits, we use the optimized gate-level combinational benchmarks available online [17]. Table I reports key structural properties for the selected benchmarks, such as the number of circuit inputs (PI), circuit outputs (PO), number of gates, and the number of stuck-at-faults.

A. *Comparing checking functions with simple implications*

Before assessing the effectiveness of our new residual implications, we first found the error coverage achievable using our implementations of both simple implications and checking functions. Through the procedures outlined in section III-B, we computed all simple implications and checking functions

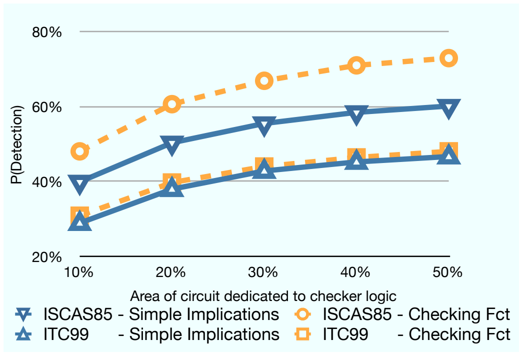

Fig. 2: Comparison in the probability of detecting an error when the checker logic is composed of either all simple implications or all checking functions.

for each benchmark. We then selected an appropriate subset of all simple implications or checking functions that would maximize coverage for a particular hardware budget. For these experiments, we approximated hardware overhead as the total number of extra standard cells added, compared to the original gate count of the circuit, where each simple implication is assumed to be implemented using a single standard cell and each checking function is implemented using two standard cells. In our experiments we allowed the checker logic to occupy an area equivalent to approximately 10%-50% of the circuit layout.

Once the checker logic elements were chosen, for each subset we used Eqn. 4 to calculate the probability that an error in a given clock cycle would be detected by the checker logic. This probability of detection is averaged across the ISCAS85 and ITC99 circuits, and the results are shown in Figure 2. Checking functions, represented with a dashed line, provide a higher probability of detection for the unoptimized circuits of the ISCAS85 suite. This is not an entirely surprising outcome because checking functions tend to exist in a circuit due to logic redundancies [2]. Since the combinational ITC99 benchmarks we selected have been optimized by their creators [17], it is likely that they contain fewer redundancies. This results in the checker logic created with a set of simple implications and that created with checking functions having roughly equivalent performance.

B. *Residual implications and splitting nodes*

We then began to investigate the effectiveness of residual implications. First, we investigated the impact of increasing the number of splitting nodes during the implication discovery process. Each additional splitting node provides a new way of dividing the input space into two pieces, and thus allows new residual implications to appear. Figure 3 shows the improvement in the probability of detection as we consider the inclusion of residual implications, generated by additional splitting sites, into the optimized set of simple implications. The plot, obtained after generating an optimized set of simple+residual implications for the ISCAS85 circuits, reveals an upwards trend in the probability of detection as the number of splitting sites increases. However, we found that increasing the number

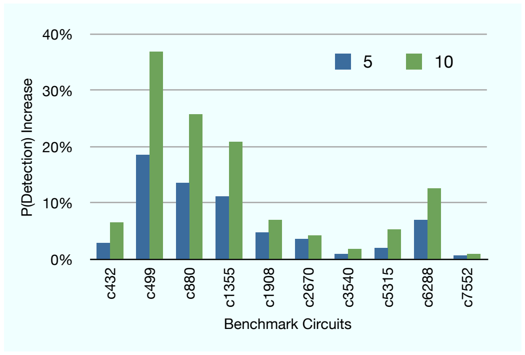

Fig. 3: Improvement in the probability of detecting an error as we allow 5 and 10 splitting sites. A 50% hardware budget for the checker logic was used for these experiments.

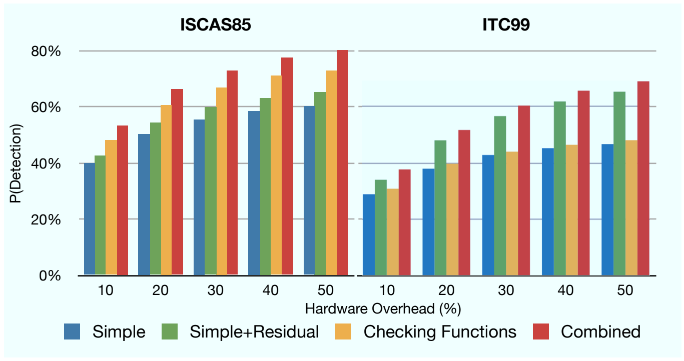

Fig. 4: Probability of detection for the various subsets of implications.

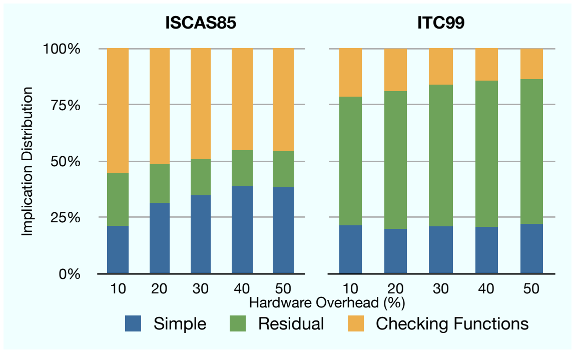

Fig. 5: Distribution of the types of relationships making up the checker logic when all three types are combined.

of splitting sites beyond 10 using our splitting site selection criteria did not necessarily increase the overall coverage in these circuits enough to justify the additional analysis cost. Thus, for the remaining experiments, we allowed 10 splitting sites to be the basis of our simple+residual implication set.

C. *Combining all checker logic elements*

We now consider four different scenarios for the checker logic implementation:

- simple two-site implications alone
- simple two-site implications and residual implications alone
- checking functions alone
- all three combined

In each case, a subset of the possible implications/checking functions are chosen to be included in the checker logic from the sorted list described in Section III-B. Figure 4 shows the propability of detection for the different types of checker hardware. The probability of detection values for each area overhead have been averaged across the ISCAS85 and ITC99 circuits separately. From this, we can observe that:

- While the optimized set of simple+residual implications is slightly inferior to the checking functions when applied to the ISCAS85 benchmarks, it is significantly better than either simple implications or checking functions when applied to the ITC99 benchmarks. This is encouraging because it indicates that residual implications can provide significant added value when applied to optimized circuitry with fewer redundancies.
- Because implications and checking functions have different fault-targeting properties, the combined checker logic is able to detect more errors than any individual checker logic implementation. By allowing only 10% of the hardware to be dedicated to the checker logic, we are able to detect errors on average almost 40% of the time for the ITC99 circuits and roughly 53% of the time for the ISCAS85 circuits.

Figure 5 shows the distribution for the type of implications in the optimized set of combined checker logic. Since checking functions were highly effective when applied to the ISCAS85 benchmarks, it is unsurprising that around half of the elements in the combined checker logic came from that group. The results are quite different for the circuits from the ITC99 benchmark suite, where residual implications became the major providers of error detection.

D. *Fault coverage overlap for different implication sets*

The combined checker logic showed us that we can get additional fault coverage by using checking functions together with implications. We also determined the fault coverage breakdown for different checker logic scenarios. We performed fault simulation on each ITC99 benchmark circuit, and for each propagated fault we determined which checker logic set would detect it. Using VennMaster [18], Figure 6 shows the relative number of faults covered by implications and checking functions. As expected, the faults covered by the simple+residual implications are not identical to the faults covered by checking functions. Figure 6 also highlights the fault coverage advantage of combining implications and checking functions in the checker logic, as it is able to cover most of the faults detected by the simple+residual implications and also most of the faults detected by the checking functions.

E. *Residual implications on synthesized circuits*

As reported above, we noticed that residual implications appear to be especially useful for circuits that have gone through some optimization procedure. To further investigate this hypothesis we re-synthesized ISCAS85 benchmarks, using

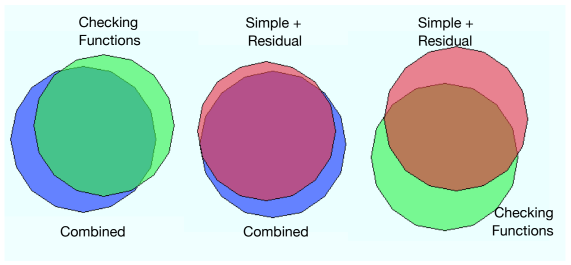

Fig. 6: Relative number of faults covered for all ITC99 benchmarks when 10% of hardware area is allocated to checker logic. Overlapping area indicates faults that are covered by more than one technique.

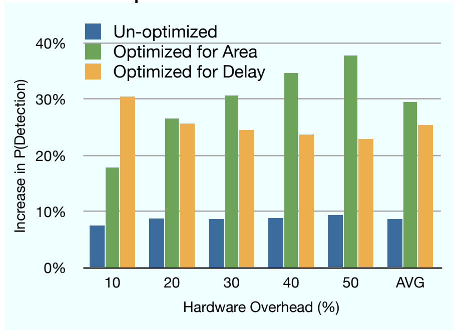

Fig. 7: Average improvement in probability of detection for ISCAS85 circuits using the simple+residual vs. the simple set of implications for different circuit optimizations.

Leonardo Spectrum, and optimized the circuits for area and delay. We then determined the relative impact on fault coverage that could be obtained when residual implications are added to the simple implication set for different overheads.

When we perform any logic re-optimization on a circuit, the internal structure will likely change, even though the logic functionality will remain the same. This implies that any previously calculated invariant relationship will not necessarily hold in a re-optimized circuit. For each re-optimized circuit, we re-computed all simple 2-site implications and residual implications, ensuring that the number of checker logic elements remain proportional to the new circuit area. Simulation results comparing optimized and unoptimized circuits are shown in Figure 7. In particular, we show the average improvement in probability of detection when we use the simple+residual implication set as opposed to the simple implications alone. As one can see, the residual implications for the area/delay optimized circuits become much more important contributors to the obtained error coverage when compared to non-optimized circuits.

V. Conclusions

In this paper we have introduced residual implications, a type of multi-site implication, that may be effectively used to detect faults not covered by simple implications or checking functions. We have presented an algorithm for finding and choosing a set of residual implications to be incorporated into the checker logic based on its ability to detect errors. While the exact amount of error detection provided from residual implications varies, our analysis has shown that significant improvement in error detection is possible, especially in the case of the ITC99 benchmarks studied.

We also compared the behavior of residual implications against simple 2-site implications in circuits optimized for area or delay. When inserted into checker logic for these optimized circuits, the residual implications provided on average a more than 25% increase in the ability to detect errors when compared to using only simple 2-site implications alone.

Finally, we showed how residual implications can be combined with other concurrent online error detection methods, such as checking functions. With the implementation of hybrid checker logic we were able to significantly improve the online error detection capability. By allowing only a 10% area overhead for the hybrid checker logic, we were able to detect more than 50% of the errors in the unoptimized ISCAS85 benchmarks, and almost 40% of the errors in the ITC99 benchmark circuits. With a 40% area overhead, error detection was almost 80% and 65% respectively.

References

[1] N. Alves, A. Buben, K. Nepal, J. Dworak, and R. I. Bahar, "A cost effective approach for online error detection using invariant relationships," *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, vol. 29, no. 5, pp. 788–801, 2010.

[2] I. Pomeranz and S. M. Reddy, "Reducing fault latency in concurrent on-line testing by using checking functions over internal lines," in *DFT*, 2004, pp. 183–190.

[3] S. Mitra and E. McCluskey, "Which concurrent error detection scheme to choose?" in *ITC*, Oct. 2000, pp. 985–994.

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

[5] M. A. Gomaa and T. N. Vijaykumar, "Opportunistic transient-fault detection," in *ISCA*, June 2005.

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

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

[8] K. Mohanram and N. Touba, "Cost-effective approach for reducing soft error failure rate in logic circuits," in *ITC*, Oct. 2003, pp. 893–901.

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

[10] R. Sharma and K. Saluja, "An implementation and analysis of a concurrent built-in self-test technique," in *FTCS*, June 1988, pp. 164–169.

[11] P. Drineas and Y. Makris, "Concurrent fault detection in random combinational logic," in *ISQED*, Mar. 2003, pp. 425–430.

[12] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, and Y. Wolfsthal, "FoCs: automatic generation of simulation checkers from formal specifications," in *CAV*, 2000, pp. 538–542.

[13] R. Vemu, A. Jas, J. Abraham, S. Patil, and R. Galivanche, "A low-cost concurrent error detection technique for processor control logic," in *DATE*, March 2008, pp. 897–902.

[14] I. Pomeranz and S. M. Reddy, "On finding functionally identical and functionally opposite lines in combinational logic circuits." *VLSI Design Conf*, pp. 254–259, 1996.

[15] F. Brglez and H. Fujiwara, "A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran," in *ISCAS*, 1985.

[16] F. Corno, M. S. Reorda, and G. Squillero, "Rt-level ITC'99 benchmarks and first ATPG results," *IEEE Des. Test*, vol. 17, no. 3, pp. 44–53, 2000.

[17] Reorda. (2010, September) ITC99 benchmarks. [Online]. Available: http://www.cad.polito.it/downloads/tools/itc99.html

[18] S. Chow and P. Rodgers, "Constructing area-proportional venn and euler diagrams with three circles," in *Euler Diagrams Workshop 2005*, August 2005. [Online]. Available: http://www.cs.kent.ac.uk/pubs/2005/2354