VLSI DESIGN
1998, Vol. 7, No. 4, pp. 347–352
Reprints available directly from the publisher
Photocopying permitted by license only

Testability-Driven Layout of Combinational Circuits

C. P. RAVIKUMAR* and NIKHIL SHARMA

Department of Electrical Engineering, Indian Institute of Technology, New Delhi 110016

(Received 27 June 1994; In final form 1 August 1995)

The layout of a circuit can influence the probability of occurrence of faults. In this paper, we develop algorithms that can take advantage of this fact to reduce the chances of hard-to-detect (HTD) faults from occurring. We primarily focus on line bridge faults in this paper. We define a bridge fault f as an HTD fault if an automatic test pattern generator fails to generate a test vector for f in a reasonable amount of CPU-time. It is common practice to drop such HTD faults from consideration during test generation. The chip fault coverage achieved by a test set is poor if the fault set consists of many HTD faults. We can combat this problem by avoiding altogether, or by reducing the probability of, the occurrence of HTD faults. In this paper, we consider hard-to-detect bridging faults and show how module placement rules can be derived to reduce the probability of these faults. A genetic placement algorithm that optimizes area while respecting these rules is presented. The placement algorithm has been implemented for standard-cell layout style on a SUN/SPARC and tested against several sample circuits.

Keywords: Bridge faults, placement algorithm, genetic algorithm

1 INTRODUCTION

The classical stuck-at fault model is a *logical* fault model and does not directly relate to *physical* manufacturing defects [8]. Thus, a physical defect that does not manifest itself as a stuck-at fault will remain undetected even if 100% stuck-at fault coverage is attained. Similarly, if testing reveals that a certain subset of stuck-at faults occur frequently, no preventive action can be taken to correct the manufacturing processes. In the recent past, other fault models such as bridge-faults and stuck-open faults have been introduced to alleviate this problem [4, 5, 7, 6, 8]. Jacomet introduced fault models which are influenced by the layout of the circuit [4, 5]. In [6], Koeppe uses local transformations of the transistor-level layout to avoid the occurrence of certain stuck-open faults. In the same vein, Levitt and Abraham derive physical design rules which improve stuck-open fault testability [7]. In this paper, we consider bridge fault testability and its relation to circuit layout. A bridge fault $x@y$ occurs between two lines (signals) x and y if there is an electrical short between the two lines [1]. The effect of a bridge fault is to give rise to a wired-AND or a wired-OR

*Corresponding author.

between lines x and y. Bridge faults can be further classified into *Feedback Bridge Faults* and *Non-feedback Bridge Faults*. A non-feedback bridge fault occurs when two lines which do not depend on one another are shorted; e.g a short between two primary inputs. On the other hand, when a bridge fault occurs between the output line and an input of a combinational circuit, the resulting feedback effect may give rise to sequential behavior. Such bridge faults cannot be satisfactorily modelled as stuck-at faults; recently IDDQ testing has been employed to test for the occurrence of bridge faults. Chakravarty and Thadikaran have introduced IDDQ test generation and fault simulation algorithms for both feedback and non-feedback bridge faults [2]. However, in their work bridge faults between all pairs of lines are considered for fault simulation. If there are n signal lines in the circuit, this would mean a fault set of size $n \cdot (n-1)/2 -$ a very large number for VLSI circuits. Knowledge about the *layout* of the circuit can be used to reduce the size of the fault set. Thus if two lines x and y are placed sufficiently apart in the layout, then the bridge fault $x@y$ can be dropped from the fault set. An *analysis* of the layout is required for this purpose. In such an approach, the presumption is that layout will be performed first, and test generation and fault simulation will follow the layout phase. In this paper, we consider the *synthesis* of layout based on the results of a bridge fault simulation. Thus we present an approach where all the $n \cdot (n-1)/2$ bridge faults will be considered and through efficient IDDQ test generation and fault simulation algorithms, we can identify a small set of hard-to-detect bridge faults. Chakravarty and Thadikaran have shown that such a fault simulation can be carried out in a reasonable amount of CPU time even for large combinational circuits [2]. Our approach is to use layout rules which will reduce the chances of HTD faults during manufacture. Table I summarizes the relative merits and demerits of the *layout synthesis for testability* approach and the *post layout fault simulation analysis* approach.

1.1 Confidence of Testing

Let H be the set of hard-to-detect (HTD) faults deived from the a fault simulation of an IDDQ test set T which considers all $n \cdot (n-1)/2$ bridge faults possible in a circuit with n lines. The fault-coverage FC of the test set T is given by

$$FC = \frac{|F_c|}{|F|} \times 100\% \tag{1}$$

where F_c is the set of faults covered by T and F is the fault set. If the layout of the circuit can be influenced by the knowledge of H to avoid a subset $H_c \in H$, then the fault coverage of a test set T improves to

$$FC' = \frac{|F_c|}{|F - H_c|} \times 100\% \tag{2}$$

In this paper, we use a bridge fault simulator to derive the set of HTD faults (See Section 2). Layout techniques can be used to reduce the probability of occurrence of a bridge fault $x@y$; in this paper, we only consider module *placement* rules which help us in this direction (Section 3). In Section 4, we discuss a genetic placement algorithm which minimizes the chip area while conforming to these placement rules. Results and conclusions are presented in Section 5.

2 HARD-TO-DETECT BRIDGE FAULTS

We used the procedure shown in Figure 1 to identify the HTD bridge faults. The procedure begins with a large number of random test vectors ($N = 1000$) and keeps doubling the number of test vectors until no new faults are detected by the larger test set. The fault simulator used in our implementation was provided by Chakravarty and Thadikaran [2]. The procedure converged in a small number of iterations for the examples which we considered.

TABLE I A Comparison of the "Post Simulation Layout Synthesis" Approach and "Post-Layout Analysis" Approach

	Synthesis Approach	Analysis Approach
Fault Simulation Effort	Higher; must consider all likely bridge faults	Smaller; need to consider bridge faults of nearby nets
Layout Analysis Effort	None	Higher; need to identify neighborhood lines
Layout Effort	Higher due to extra layout constraints	Lower; no extra constraints
Layout Area	Can be less compact due to layout constraints	More area-efficient
Confidence in	Higher, due to low chance of occurrence of HTD faults	Lower, since many adjacent faults may be HTD.

```
procedure HTD(F)
{ Identify the set of hard-to-detect faults in F}
begin
        repeat
                T := GenRandTest(N); {Initial Set of N random tests}
                Fc:= FaultSimulate(T); { Fc is the set of faults covered by T }
                F:=F − Fc; { Drop the covered faults }
                N:= 2 × N; { Double the number of faults }
        while (Fc > 0);
        H :=F;
end
```

FIGURE 1 Identifying Hard-to-detect faults.

```
procedure TestCost(P, H)
{ P is the placement of the circuit and H is set of HTD bridge faults}
begin
        Cost := 0;
        for each x@y ∈ H do
                if Overlap(x, y) then
                  Cost:= Cost + ω;
                else
        Cost := Cost + 1 / dist(B(x), B(y));
        return (Cost);
end
```

FIGURE 2 Evaluation of Testability Cost of a Layout

3 TESTABILITY-DRIVEN PLACEMENT

We define a pair of nets x, y as a critical pair if $x@y$ is an *HTD* bridge fault. A testability-driven layout algorithm must ensure that the nets x and y are placed sufficiently far apart so that fault $x@y$ is unlikely to occur. We define the bounding box of a net x as the smallest rectangle which includes all the pins of net x. Let $BB(x)$ and $BB(y)$ refer to the bounding boxes of the nets x and y in the circuit layout. If $x@y \in H$ and an overlap does not exist between $BB(x)$ and $BB(y)$, the bridge fault has low chance of actually occurring. This probability decreases rapidly as the distance between the rectangle increases. We shall define the distance between $BB(x)$ and $BB(y)$ as the smallest of the Euclidean distances of the form $dist(P, Q)$, where P and Q are corner points of $BB(x)$ and $BB(y)$ respectively. A cost function can be formulated to evaluate the testability property of a given layout for a given set of *HTD* bridge faults. Let $Overlap(x, y)$ be defined as shown in Equation 3 for two nets x and y. The procedure *TestCost* shown in Figure 2 computes the testability cost. In the next section, we shall use a genetic algorithm which minimizes the testability cost of the layout. Computationally, the testability evaluation procedure is fast and requires $O(|H|)$ time. A faster (but less accurate) testability estimation procedure can be devised by omitting the *else* part in the procedure of Figure 2.

$$\begin{aligned} Overlap(x, y) &= 1 \quad \text{if } BB(x) \cap BB(y) \\ &= \phi, \quad 0 \text{ otherwise} \end{aligned} \tag{3}$$

4 GENETIC ALGORITHM

Genetic algorithms are inspired by the biological process of evolution and natural selection; they have been used to solve many optimization problems [3, 9]. A genetic placement algorithm called *GENIE* was presented in [3].

A standard-cell placement program based on the genetic algorithm, called *GASP*, is described in

[9]. The genetic algorithm maintains a population of solutions, which are treated as individuals in a society. A crossover operator is applied to a pair of solutions (parents) to obtain a third solution (offspring). The offspring shares properties of both the parents. In addition, sometimes a randomized operator known as *mutation* is applied to the offspring to introduce new characteristics that are not present in either of the parents. A number of offspring are generated and the *survival of the fittest* rule is applied to maintain the size of the population at the original level. This completes a single generation of the genetic algorithm; at the end of each generation, the average cost of the population improves. The genetic algorithm runs through several generations until convergence is achieved; convergence may be defined as the condition when the best individual in the population has not improved in terms of cost function over a predefined number of generations.

We used the genetic algorithm for solving the testability-driven placement problem. The motivations for using the genetic algorithm are

- the algorithm can handle a number of objective functions such as testability cost, wiring length, and wiring congestion;
- the algorithm can lead us to global optimal solutions since it explores a much larger search space than other competing algorithms [9].

```
procedure GeneticPlacement(H, NL)
{ H is the set of HTD bridge faults and NL is the circuit netlist.}
begin
    for i := 1 to P do
        Population(i) := InitialPlacement(NL);
    Γ := 0 ;
    BC := BestCost (Population);
    repeat
        for i := 1 to O do begin
            M := SelectParent(Population);
            F := SelectParent(Population);
            Offspring(i) := Crossover(M,F);
            f = Flip(μ)
            if (f = 1) then Offspring(i) := Mutate(Offspring(i));
        end
        Population := Fittest(Population, Offspring, P);
        if BestCost(Population) = BC then Γ := Γ + 1;
        else begin
            Γ := 0;
            BC := BestCost(Population);
        end
    until Γ > Γmax;
end
```

FIGURE 3 Genetic Algorithm for Testability-Driven Placement.

The overall algorithm is shown in Figure 3. The procedures used within the genetic algorithm are summarized in Table II. A solution to the placement problem is represented in the form of 2-dimensional array $\mathcal{P}$, where $\mathcal{P}(i, j)$ contains the number of modules placed in the i-th row and j-th column. The placements which constitute the initial population were generated using a constructive placement algorithm similar to the one used in [3]. The crossover operator used in our implementation works by copying a patch of placement from parent M and the remaining placement from parent F. Figure 4 shows the details of the crossover operator. If the patch (set of locations) selected in the crossover operator corresponds to the bounding box of a net x which is part of a critical pair, the resulting offspring is less likely to have overlaps. The objective of the mutation operator is to reduce the bounding rectangle of a randomly selected net, which is achieved by a constructive procedure. When selecting a net n for mutation, priority is given to a net which belongs to a critical pair; this improves the testability of the resulting layout by reducing the chances of overlap of $BB(n)$ with bounding boxes of other nets.

The fitness measure used in our placement algorithm is a weighted sum of three cost functions which represent the total wirelength, wiring congestion, and the bridge-fault testability. The bridge-fault testability cost is estimated using the procedure shown in Figure 1. The estimators of wire length and congestion are well known in literature; we refer the reader to [3, 9].

5 RESULTS AND CONCLUSIONS

We have implemented the testability-driven placement algorithm on a Sun- SPARC workstation in the *C* programming language. Standard-cell layout

TABLE II Procedures used in the Genetic Algorithm

Procedure	Description
Initial Placement (*NL*)	Using netlist *NL*, generates a constructive placement of the circuit
BestCost (Population)	Returns the cost of the best solution in Population
Select Parent (Population)	Selects an individual randomly from the population, giving higher priority to parents with better fitness function
Crossover (*M*, *F*)	Returns an offspring after crossing the parent olutions *M* and *F*
Flip (μ)	Implements a Bernoulli-distributed random variable with parameter μ
Mutate (*O*)	Applies a small random perturbation to the offspring *O*
Fittest (Population, Offspring, *P*)	Selects the best *P* of the set Population $\bigcup$ Offspring

```
procedure Crossover(M, F)
{ M and F are placements which must be crossed to generate placement O }
begin
        O := F; {Copy placement F into O }
        Randomly select a set L of c locations;
        Let p_m be the set of modules in M occupying the locations in L;
        Let p_f be the set of modules in F occupying the locations in L;
        Let r_m be the modules in M not contained in p_m;
        Let r_f be the modules in F not contained in p_f;
        Copy the placement of modules in p_m into the offspring placement;
        { The remaining part of the procedure ensures
        that the placement O is valid.}
        s := p_m ∩ r_f;
        t := p_f ∩ r_m;
        repeat
                Randomly select two modules x, y, x ∈ s, y ∈ t;
                Swap the locations of modules x, y in O;
                s = s - {x}; t := t - {y};
        until s = φ and t = φ;
        return (O);
end
```

FIGURE 4 Crossover Operator in Genetic Algorithm.

style is a assumed in our implementation. We also implemented a parallel version of the genetic placement algorithm on a 32-node Meiko transputer; parallelization was achieved by decomposing the parameter space. We ran multiple copies of the genetic algorithm on different nodes of the transputer, each of them using a different set of genetic parameters. The important genetic parameters are the convergence parameter Γ_{max}, the population size *P* and the mutation probability μ. We experimented with several combinational circuits and the results are tabulated in Table III. The Table shows the percentage of the testability constraints satisfied by the final placement and the increase in the wiring cost to achieve testability. As explained in the previous section, a testability constraint requires two nets x and y to be placed far apart, where x, y belong to a critical pair. The increase in wiring cost is measured against the wiring cost when the testability requirement is turned off by setting the weight of the testability cost to zero in the composite cost function.

5.1 Conclusions

In this paper, we have presented a genetic algorithm for testability-driven placement. The fault model considered in this paper includes bridge faults which relate closely to the physical

TABLE III Results of Testability-Driven Placement Algorithm

Circuit Name	Number of Gates + *I/O Pins*	Number of Nets	Number of HTD Faults	% Constraints *Satisfied*	% Increase in Wiring Cost
full adder	18	17	12	85.70	12.96
parity checker	24	21	14	66.87	10.26
c17	13	6	5	80.00	13.33
c432	205	189	1049	94.28	4.91

design of the circuit. We found that at the cost of a marginal increase in the total wiring cost, the confidence level of a bridge-fault test can be improved significantly. Our work can be extended by considering the testability constraints in other stages in physical design such as global routing and channel routing. A global router implements each net in the form of a Steiner tree; one can consider global routing as a constrained optimization problem, where the Steiner trees corresponding to nets of a critical pair are to be constructed in a non-overlapping fashion. Despite testability-driven placement and global routing, there may be violations of testability constraints; a channel router can handle these leftover constraints by placing the nets of a critical pair in two different layers.

Acknowledgements

We thank Prof. Sreejit Chakravarty and Mr. Paul Thadikaran for providing us the bridge fault-simulator. We also thank the two anonymous referees for helpful comments.

References

[1] Abromovici, M. A., Breuer, M. A. and Friedman, A. D. (1990). *Digital Systems Testing and Testable Design*, Computer Science Press.

[2] Chakravarty, S. and Thadikaran, P. J. (1993). Simulation and generation of iddq tests for bridging faults in combinational circuits. In *Proceedings of the International Conference on VLSI Design*, Bombay, India.

[3] Cohoon, J. P. and Paris, W. D., Genetic placement. *IEEE Transactions on Computer-Aided Design*. **CAD-6**(6), November 1987.

[4] Jacomet, M. (1989). FANTESTIC: Towards a powerful fault analysis and test pattern generator for integrated circuits. In *Proceedings of the International Test Conference*, pages 633–642.

[5] Jacomet, M. (1990). Stuck-at-faults vs. layout dependent faults. In *Proceedings of the IEEE Workshop on Defect and Fault Tolerance in VLSI Systems*, Grenoble, France.

[6] Koeppe, S. (1987). Optimal layout to avoid CMOS stuck-open faults. In *Proceedings of the ACM/IEEE 24th Design Automation Conference*.

[7] Levitt, M. E. and Abraham, J. A., Physical design of testable VLSI – techniques and experiments. *IEEE Journal of Solid State Circuits*, **25**(2), April 1990.

[8] Maly, W. (1987). Realistic modelling for vlsi testing. In *Proceedings of the ACM/IEEE 24th Design Automation Conference*.

[9] Shahookar, K. and Mazumder, P., Placement techniques. *ACM Computing Surveys*, **23**(2), June 1991.

Authors' Biographies

C. P. Ravikumar received the Bachelor of Engineering degree in Electronics from Bangalore University (1983), the Master of Engineering degree in Computer Science from the Indian Institute of Science (1987) and Ph.D., in Computer Engineering from the University of Southern California (1991). Since September 1991, he is an Assistant Professor in the Department of Electrical Engineering at the Indian Institute of Technology, New Delhi. He was a research assistant in the University of Southern California during 1987 to 1991 in the Department of Electrical Engineering Systems. He is currently visiting the University of Southern California where he is conducting research in the area of system-level synthesis in the Department of Electrical Engineering Systems. His research interests are in the areas of synthesis, testing and testability, low power design, and high performance computing.

Nikhil Sharma received the Bachelor of Technology degree in Electrical Engineering from the Indian Institute of Technology, Delhi in 1994. He obtained a Master's degree in Computer Engineering from University of Wisconsin-Madison. He is currently an engineer in Synopsys Inc., Milipitas, CA. His interests include synthesis, simulation and testing of digital systems.