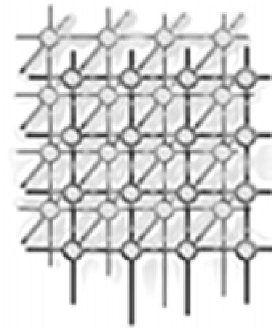
CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **12**:311–326



Parallel VLSI test in a shared-memory multiprocessor

C. Gil[1,*,†], J. Ortega[2] and M. G. Montoya[1]

[1]*Dept. de Arquitectura de Computadores y Electrónica, Universidad de Almería, La Cañada de San Urbano s/n, 04120 Almería, Spain*
[2]*Dept. de Arquitectura y Tecnología de Computadores, Universidad de Granada, Campus de Fuentenueva s/n, 18071 Granada, Spain*

SUMMARY

This paper presents three parallel procedures implemented in a shared-memory multiprocessor to generate the patterns that allow the testing of digital circuits. The implementation of these procedures in a multiprocessor uses the system memory better than in a distributed-memory multicomputer, since it is not necessary to store the circuit structure in the local memory of each processor, besides other common structures. The parallel test generation procedures are based on a new sequential algorithm which mixes both the Boolean difference and digital spectral techniques. It is thus different from other methods proposed that deal with the parallelization of test generation algorithms that carry out an implicit enumeration of the input pattern space. The first procedure distributes the set of faults using a backtracking procedure starting from a primary output and allocating a similar number of lines to each processor. The second procedure distributes the set of faults among the processors taking into account the distance from each line to its nearest primary output; it then applies the algorithm to generate the test pattern with some modifications. The third procedure uses a circuit partitioning procedure which allows similar sized parts of the circuit to be assigned to each processor while communications between processors are minimized. The experimental results obtained when the procedures are applied to the usual benchmark circuits (the ISCAS set) show figures for speedup better than in a multicomputer, although fewer processors are used. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: VLSI test; Boolean; digital spectral

1. INTRODUCTION

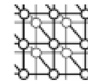
The generation of a test for a fault involves searching among the set of possible input vectors of the circuit until a test pattern is found. As the size and complexity of VLSI devices increases, the generation of test patterns is becoming more and more difficult.

*Correspondence to: C. Gil, Dept. de Arquitectura de Computadores y Electrónica, Universidad de Almería, La Cañada de San Urbano s/n, 04120 Almería, Spain.
†E-mail: cgil@peke.ualm.es

Contract/grant sponsor: CICYT, Spain; contract/grant number: TIC 97-1149, TIC 099-0361

Copyright © 2000 John Wiley & Sons, Ltd.

Received 15 July 1999
Revised 25 November 1999



The problem of automatic test pattern generation (ATPG) for combinational circuits has been extensively discussed during the last few years. The most frequently used fault model is the single stuck-at fault that fixes the faulty line to the logical value 0 (stuck-at-0 fault) or 1 (stuck-at-1 fault). Since ATPG is an NP-complete [1] problem with exponential complexity to circuit size, the application of parallel processing techniques to accelerate the process of generating test vectors is an active area of research.

The parallel test pattern generators reported up to now are based on algorithms such as PODEM [2] and its derivatives, which perform an implicit enumeration of the space of input patterns, guided by local information about the circuit. These are usually termed path-oriented techniques. When the algorithm makes an incorrect assignment it backtracks to the point where the last assignment was made and tries an alternative.

The strategies that can be used to parallelize a test algorithm [3–12] can be classified as: (a) *fault partitioning*, where the set of faults is divided among the processors that generate the patterns for each fault in their corresponding fault list; (b) *search-space partitioning*, where the space of assignments to the inputs, the space of assignments to the lines, or both, is divided among the processors which work together in the search for a pattern for each fault; and (c) *circuit partitioning*, where the circuit is distributed among the processors that apply the corresponding test algorithm to each subcircuit.

The problem with fault partitioning when using an implicit enumeration algorithm is that the amount of space to search varies greatly depending on whether or not the fault is hard to detect [1]. Thus it is necessary to use efficient dynamic schemes to balance the sets of faults assigned to the processors. Such a load balancing procedure consumes computer resources and could imply negative effects on the scalability of the parallel procedure.

With respect to search-space partitioning, perhaps PODEM [4] or PODEM-like algorithms are the algorithms that have been most frequently parallelized using this technique [3,6,8]. PODEM explores the solution space structured as a binary tree, so that the determination of disjoint subspaces to be evaluated by different processors is relatively easy. Nevertheless, as in the fault partitioning case, here too it is necessary to implement a load balancing method which divides the search space into parts that imply similar amounts of work. As it is difficult to determine the magnitude of the search space before the execution of the procedure, the load should be dynamically balanced, thus causing some overhead. Moreover, the speedup obtained for a given fault depends on the size of the tree being searched by the serial PODEM. If a fault is easy to detect, the parallel algorithm cannot provide much speedup because the part of the tree to be searched is small and cannot support much concurrency; only if the fault is hard to detect (HTD) is it possible to obtain acceptable speedups.

An important drawback to strategies (a) and (b) when they are implemented in multicomputers is that every processor must have fast access to the whole structure of the circuit. This implies that the circuit needs to be stored in the local memories of the processors used to speed up the test pattern generation process, thus requiring considerable memory resources. This is an important limitation for highly complex circuits, because the memory requirements can be excessive. Moreover, loading the circuit into the multicomputer would require a lot of time, thus limiting the speedup achieved by the parallel program.

Multiprocessors are now becoming commercially available and most modern programming environments support parallel programming. The way the processes in the parallel program are scheduled to the processors of the multiprocessor system significantly affects the performance. The shared-address-space (s-a-s) programming style is naturally suited to shared-memory multiprocessors.

Copyright © 2000 John Wiley & Sons, Ltd.

A parallel program on a shared-memory multiprocessor shares data by storing it in globally accessible memory. In our problem, s-a-s is adequate to store the structure of the circuit instead of storing it in each local memory as in a distributed-memory multicomputer.

Serial procedures other than implicit enumeration techniques include those based on the Boolean difference or a variation of it. These are based on the obtention of the test equation for a given fault. This equation is satisfied by all input patterns that can detect the corresponding fault. Moreover, in these kinds of methods the difference between hard-to-detect faults and easy-to-detect faults disappears because both faults usually require similar computing times. Although the first implementations of these methods were quite slow, several Boolean methods have recently been reported [13,14] that overcome this drawback. Nevertheless, to the best of our knowledge, the parallelization of these algorithms has not been considered yet.

Here we describe three different parallel alternatives based on a procedure in which a satisfiability algorithm is applied to the test equation. The algorithm [13] operates in the domain of the Reed–Muller spectral coefficients [6,15,16], not only quickly obtaining the equation that the patterns for a given fault must verify, but also formulating it such that it allows us to determine one of its solutions (i.e. a test pattern) in an easy way.

The paper is organized into five sections. Section 2 gives a description of the serial algorithm used to generate test patterns. Section 3 describes the parallel test procedures. Finally, Sections 4 and 5, respectively, give the experimental results obtained for some circuits of the ISCAS set and the conclusions.

2. THE TEST GENERATION ALGORITHM

This section presents a brief description of the algorithm to determine the test patterns for the stuck-at faults of a circuit providing the basis for the parallel procedures presented here. A more detailed description with the corresponding proofs can be found in [13]. It combines the Boolean difference and the properties of the Reed–Muller spectrum to obtain the test equation for each node and to determine one of its solutions, i.e. a test pattern. The part of the procedure which requires most computing resources is the obtention of the test equation. Once the test equation is obtained, the determination of one of its solutions is immediate.

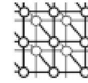
Given a circuit with n inputs, $y = (y_0, y_1, \ldots, y_{n-1})$ and w lines, $i = 0, \ldots, w-1$, the function implemented by the line i is termed as $f_i(y) = f_i(y_0, \ldots, y_{n-1})$. The function synthesized by a given output of the circuit is noted as $\mathcal{F}(y) = \mathcal{F}(y_0, \ldots, y_{n-1})$. For any internal line i, it is also possible to describe $\mathcal{F}(y)$ as a function of both the inputs, y, and f_i, $\mathcal{F}(y) = \mathcal{F}_i(f_i, y) \equiv \mathcal{F}_i(f_i, y_0, \ldots, y_{n-1})$ [14]. If the line i presents a stuck-at-α fault, then $f_i(y) = \alpha$ for all values of $y = (y_0, \ldots, y_{n-1})$, and the set of input patterns that will allow us to detect this fault is the one verifying

$$f_i(y_0, \ldots, y_{n-1}) = \overline{\alpha} \tag{1}$$

and

$$\mathcal{F}(\alpha, y_0, \ldots, y_{n-1}) \oplus \mathcal{F}_i(\overline{\alpha}, y_0, \ldots, y_{n-1}) = 1 \tag{2}$$

Copyright © 2000 John Wiley & Sons, Ltd.



The term $\mathcal{F}(\alpha, y_0, \ldots, y_{n-1}) \oplus \mathcal{F}_i(\overline{\alpha}, y_0, \ldots, y_{n-1})$ is called the Boolean difference of $\mathcal{F}$ with respect to f_i, indicated as $\partial\mathcal{F}/\partial\mathcal{F}_i$, and represents the set of inputs for which the value of $\mathcal{F}$ depends on the value of f_i. Thus, by considering that (1) is equivalent to $f_i(y_0, \ldots, y_{n-1}) \oplus \alpha = 1$, the Boolean equations (1) and (2) are equivalent to the following Boolean equation:

$$(f_i(y_0, \ldots, y_{n-1}) \oplus \alpha)(\partial\mathcal{F}/\partial f_i) = 1 \tag{3}$$

The set of solutions of (3) is the set of test patterns that allows the detection of the stuck-at-α fault in line i due to a change in the output line that synthesizes $\mathcal{F}(y)$. The problem now is to devise an efficient procedure to determine (3) expressed in an easy to manage way, and a method to find at least one of its solutions. Depending on the form of (3), the problem of obtaining one of its solutions may be quite hard to resolve. For example, when (3) is expressed as a CNF (conjunctive normal form) [14], the problem could be solved in polynomial time if it is a 2CNF problem, or it might in general require exponential time if it is a 3CNF problem, i.e. an NP-complete problem.

In the test generation method presented here, (1) and (2) are written in terms of the Reed–Muller spectrum. The digital spectral techniques enable us to code information about the structure of the circuit through the spectral coefficients corresponding to the function synthesized by the circuit. Thus, the Walsh and the Reed–Muller spectrum [15,17] have frequently been used in the field of digital logic, in the synthesis of functions, and in the signature definition for built-in self-testing.

Any function can be univocally expressed, in terms of the Reed–Muller coefficients, as a module-2 sum (EXOR) of products (AND):

$$\mathcal{F}(y) = \bigoplus_{j=0}^{2^n-1} \gamma_j (y_0^{j_0} y_1^{j_1} \ldots y_{n-1}^{j_{n-1}}) \tag{4}$$

where $j = j_0 + j_1 2^1 + \cdots + j_{n-1}2^{n-1}$, $y_j^0 = 1$, $y_j^1 = y_j$ and $\gamma_j \in [0, 1]$. The coefficients γ_j, $j = 1, \ldots, 2^n - 1$, are called Reed–Muller coefficients. For example, in the module-2 expression of the function of three variables, $\mathcal{F}(y_0, y_1, y_2) = y_0 \oplus y_0 y_2 \oplus y_1 y_2$, the coefficients γ_j with $j = 1$, $j = 5$ and $j = 6$ are equal to 1, and the remaining coefficients are 0. Thus, the function $\mathcal{F}(y)$ is determined by the set of coefficients $\mathcal{F}(y) = \{1, 5, 6\}$, which are equal to 1, or expressed in their binary representation $\mathcal{F}(y) = \{(100)(101)(011)\}$. Considering (4), it is possible to write the function $\mathcal{F}(y) = \mathcal{F}_i(f_i, y_0, \ldots, y_{n-1})$, where i is the line affected by the fault to be detected, as

$$\mathcal{F}_i(f_i, y) = \bigoplus_{j=0}^{2^n-1} \gamma_j (y_0^{j_0} y_1^{j_1} \ldots y_{n-1}^{j_{n-1}} f_i^{j_n}) \tag{5}$$

where $j = j_0 + j_1 2^1 + \cdots + j_{n-1}2^{n-1} + j_n 2^n$, $y_j^0 = 1$, $y_j^1 = y_j$, $f_i^0 = 1$, $f_i^1 = f_i$ and $\gamma_j \in [0, 1]$. Thus, extracting f_i as common factor, it is possible to obtain that

$$\mathcal{F}(y) = \mathcal{F}_i(f_i, y_0, \ldots, y_{n-1}) = h_i(y) \oplus f_i(y) g_i(y) \tag{6}$$

Thus, the Boolean equation (3) corresponding to the test patterns for a stuck-at-α fault in the line i is given by

$$(F_i(y) \oplus \alpha) g_i(y) = 1 \tag{7}$$

In this way, the procedure we propose to determine a test pattern for a stuck-at fault in the line i implies obtaining the f_i and g_i functions, to generate (7) and to calculate one of its solutions. The f_i

Copyright © 2000 John Wiley & Sons, Ltd.

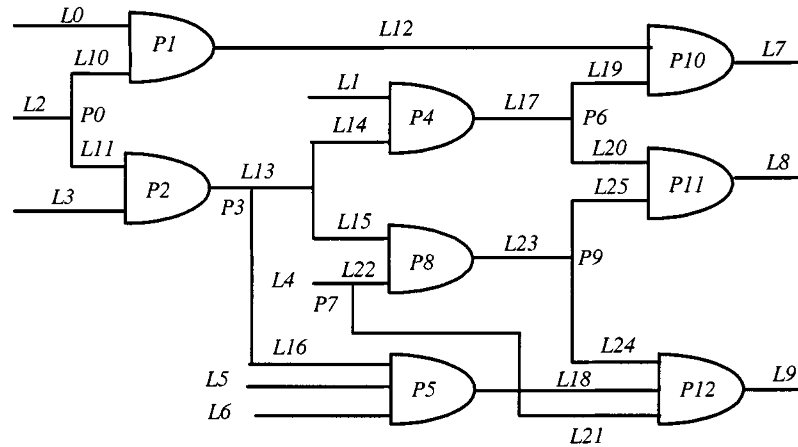


Figure 1. An example circuit.

functions can be calculated for all the lines of the circuit beginning from the inputs and proceeding along the internal lines until a given output is reached. In a similar way, the g_i functions can be obtained beginning from the outputs and finishing at the inputs of the circuit. The operations used in these processes depend on the particular logic gate [13].

All the operations involved in the determination of the f_i and g_i functions are carried out by using the Reed–Muller coefficients that describe each function, thus allowing us to compute them quickly and to obtain the equation expressed in such a way that it is easy to determine one solution for each equation.

As test equation (7) is expressed in terms of the Reed–Muller spectral coefficients, it is possible to find one of its solutions by using the following procedure [13]: the indices of the Reed–Muller coefficients that appear in the description of the left side of (7) are ordered from lower to higher values. If the lowest index is $j = (j_0, \ldots, j_{n-1})$, then a solution for (7) is $(y_0, \ldots, y_{n-1}) = (j_0, \ldots, j_{n-1})$.

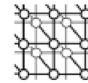
3. DESCRIPTION OF THE PARALLEL ALGORITHMS

Based on the previously described test generation algorithm, three different alternatives for its parallelization in a shared-memory multiprocessor are described as follows.

3.1. Parallelizing from fan-out cones

Among the different possibilities for parallelizing the sequential algorithm, the first idea consists of exploiting the independence between the processes that compute the g functions for each primary output of the circuit. Every partition of the circuit is called a fan-out cone and is built by backtracking from the target primary output along every possible path to a primary input. If the number of processors

Copyright © 2000 John Wiley & Sons, Ltd.



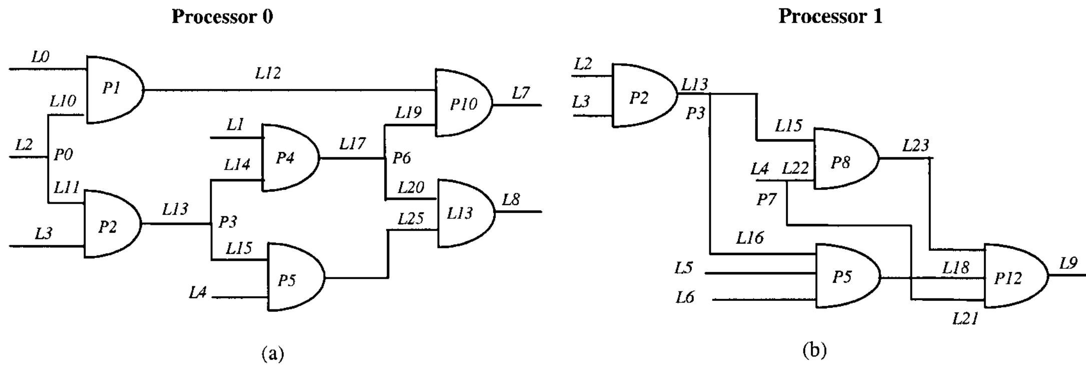


Figure 2. Structure of circuit of Figure 1 when two processors are used: (a) processor 0; (b) processor 1.

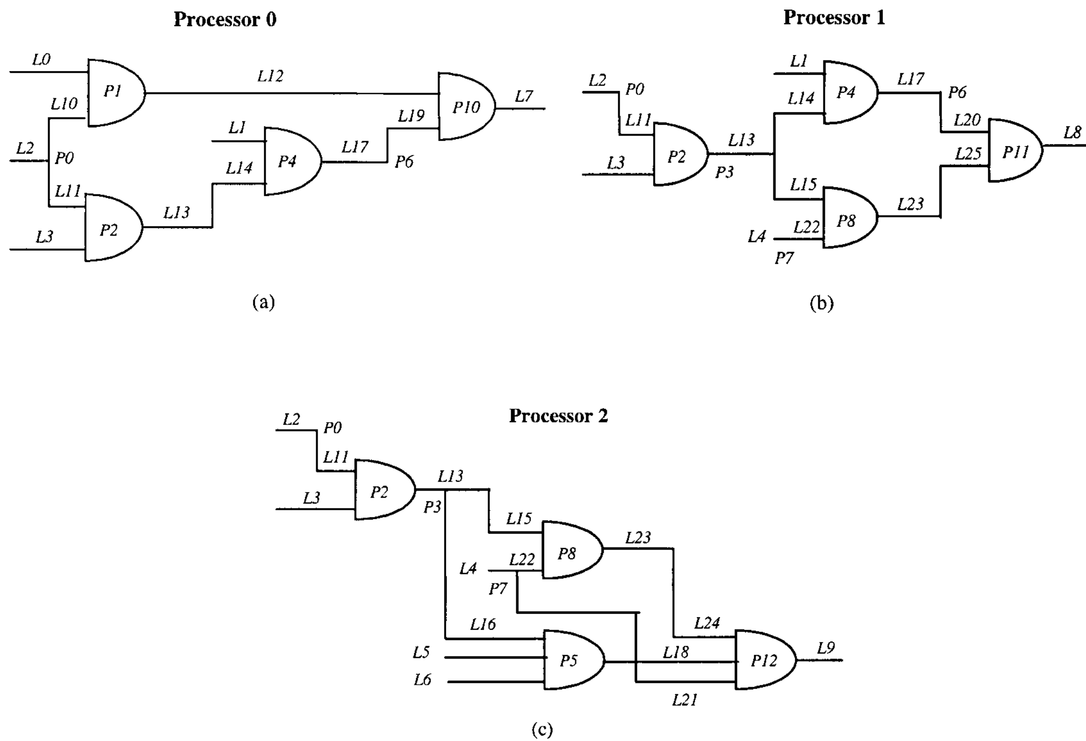
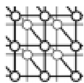


Figure 3. Structure of circuit from Figure 1 assigned to each processor: (a) processor 0; (b) processor 1; and (c) processor 2.

Copyright © 2000 John Wiley & Sons, Ltd.



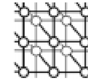

```
Procedure_1
Begin
  #pragma one processor
     read-circuit
  #pragma parallel local (node) shared (circuit, test_vectors)
  begin
     #pragma concurrent call
     #pragma pfor iterate (node←0, npr;1)
     for (node←0; node<npr; node++)
     begin
          gates_partition
          for each (line) do
               compute_f(line)
          for each (primary_output) do
               for each (line) do
                    compute_g(line,primary_output)
          compute_test_vectors
          #pragma critical
               write test_vectors
     end_for
  end_pragma_parallel
End_procedure1
```

Figure 4. Procedural description of procedure based on fan-out cones.

in the system is equal to the number of primary outputs, then the set of lines on the path from each specific primary output to the primary inputs is assigned to each processor. If there are fewer primary outputs than processors, the path from selected primary outputs is divided into different parts and each part is assigned to a different processor with no primary output assigned to it. The nodes where the division is made are the fan-out points; these are selected to ensure that no processor has to wait for any other processor to compute the g functions. If the number of primary outputs is greater than the number of processors, then a modulo operation is carried out, and a similar number of outputs (and their corresponding paths to the primary inputs) are assigned to each processor. Figure 1 shows an example circuit and Figures 2 and 3 show two possible partitions of the circuit and the set of nodes assigned to each processor when two and three processors are used, respectively.

As may be seen from Figures 2 and 3, some gates and lines are sometimes assigned to more than one processor, thus being processed two or more times. The main advantage of this is to avoid identifying non-redundant faults as redundant ones: a fault in a specific line may be redundant for a given primary output but may not be so for another output. The main disadvantage is that a different test pattern may be generated for the same fault in different processors. Figure 4 shows the procedural description of the parallel algorithm.

Copyright © 2000 John Wiley & Sons, Ltd.



Instead of creating processes at the very beginning of the parallel program run, it starts with a single process, called the master process, which reads the circuit and generates the structure to store it. When the master process needs to perform a task in parallel, it creates a predetermined number of other processes that work in parallel to perform that task. These processes are called slave processes and perform the following tasks. The procedure `gates_partition` makes a static division of the target circuit which is executed concurrently in each processor. The procedures `compute_f` and `compute_g` perform the computation of the f and g functions, respectively, as in the sequential algorithm. The procedure `compute_test_vectors` obtains the set of test patterns for the set of faults assigned to each processor.

When the task is complete, i.e. the test patterns are generated, the slave processes terminate and control returns to the master process. When separate processes access shared variables, i.e. the circuit description and the test vectors, it is important that the operations do not conflict with each other.

3.2. Parallelizing the *g* functions

In the test generation procedure presented in Section 2, it is relatively easy to predict the amount of work associated with the computation of the f_i and g_i functions. Given a line i in the circuit, the amount of work to determine the f_i and g_i functions mainly depends on the level of the circuit where this line is located. The nearer the line i is to the outputs of the circuit, the easier it is to compute the g_i function, and vice versa with the f_i function. Thus, the lines of the circuit can be quickly divided into subsets implying similar amounts of work, and then distributed among the processors.

The main objective of this parallel procedure consists of creating a partition of the circuit, so that each processor has a different set of lines for the computation of the g functions. The process of computing the g functions at each line is the same as the sequential procedure performs in the reconvergent fan-out nodes [13]. Taking into account that the decomposition of the function synthesized by the circuit in the line i is

$$\mathcal{F} = f_i g_i \oplus h_i, \quad i = 1, \ldots, w \tag{8}$$

for a stuck-at-0 fault in this line, equation (8) becomes

$$\mathcal{F}|_{f_i=0} = h_i, \quad i = 1, \ldots, w \tag{9}$$

where $\mathcal{F}|_{f_i=0}$ is the circuit output function when line i is stuck at the logic value 0. The expression that is generated when the line i has a stuck-at-1 fault is

$$\mathcal{F}|_{f_i=1} = g_i \oplus h_i, \quad i = 1, \ldots, w \tag{10}$$

where $\mathcal{F}|_{f_i=1}$ is the circuit output function when line i is stuck to the logic value 1. If the exclusive-or operation is performed between equations (9) and (10), we have

$$g_i = \mathcal{F}|_{f_i=0} \oplus \mathcal{F}|_{f_i=1}, \quad i = 1, \ldots, w \tag{11}$$

Then, applying (11) to each line of the circuit the coefficients of g are obtained. Since the procedure can be applied independently to each line, and assuming that a sufficiently good partition is provided to balance the workload, this procedure gives good speedup figures.

Figure 5 describes this parallel procedure. In the figure, the procedure `Distribute_lines` enables the workload to be distributed among the processors; this is executed concurrently in all the

Copyright © 2000 John Wiley & Sons, Ltd.

```
Procedure_2
Begin
     #pragma one processor
          read_circuit
     #pragma parallel local (node) shared (circuit, test_vectors)
     begin
      #pragma concurrent call
      #pragma pfor iterate (node←0; npr; 1)
       for (node←0, node<npr, node++)
       begin
          Distribute_lines
          Compute_All_f
          Compute_g
          Determine_Test_Pattern
       end_for
     #pragma critical
          write test_vectors
  end_pragma_parallel
End_procedure_2
```

Figure 5. Procedural description of the parallel g functions.

processors. Each processor determines the f_i functions for all the lines of the circuit by using the `Compute_All_f` procedure. In this way, the processors do some redundant work, and this does imply a reduction in the speedup that can be achieved. Nevertheless, the time needed to compute the f_i functions is several orders of magnitude less than that required to compute the g_i functions. For example, in our implementation, the time required to compute the f_is for the c1908 circuit of the ISCAS set [21] is about 1 s, while 850 s are required to calculate the g_is. As the determination of the f_is represents a small part of the work to be done, it is better to repeat the obtention of the f_i functions in every processor and thus avoid the communication and synchronization overheads arising from the access to the f_i functions by the processors when they need to use them to compute the g_i functions. The experimental results provided in Section 4 corroborate this assumption.

If we consider the example circuit of Figure 1, then the set of lines assigned to each processor for g computing is shown in Table I.

3.3. The circuit partitioning algorithm

This algorithm is based on a partition of the circuit under test and on the application of the previously described test generation procedure to each of the partitions of the circuit. The procedure starts with a partition algorithm; at the end of this, each processor knows which subcircuit has to work. The partition procedure we have implemented applies a hybrid heuristic which combines the well known simulated

Copyright © 2000 John Wiley & Sons, Ltd.

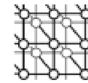


Table I. Lines of circuit of Figure 1 allocated to each processor.

Processor	Line index
0	L0, L3, L6, L9, L7, L12, L15, L18, L21, L24
1	L1, L4, L7, L10, L8, L13, L16, L19, L22, L25
2	L2, L5, L8, L11, L14, L17, L20, L23

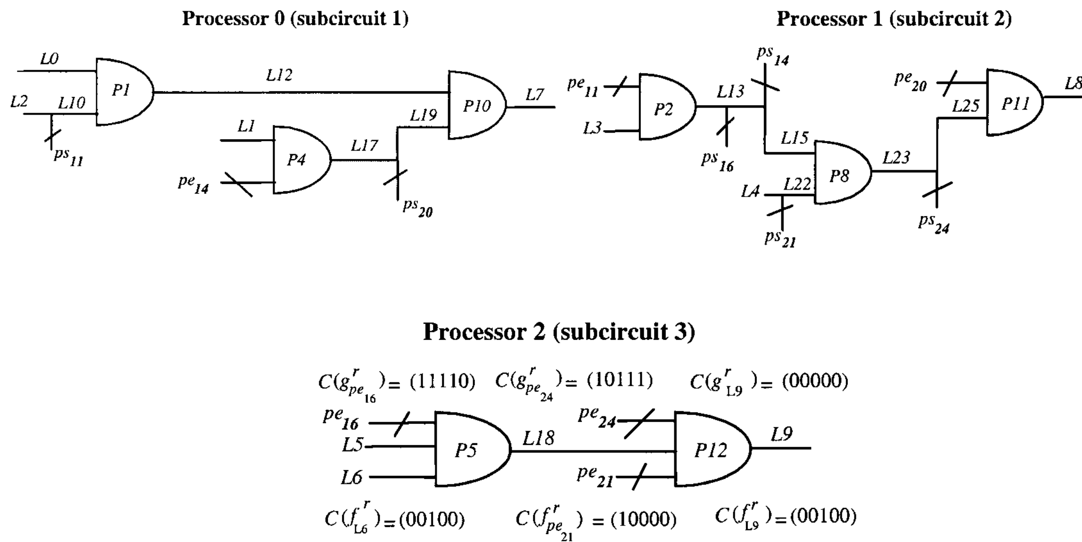


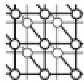
Figure 6. Example of partitioning: the circuit in Figure 1 is divided into three subcircuits.

annealing [18] and tabu search [19] techniques to optimize a multicriteria objective function, intended to optimize synchronization and maximize concurrency. The details can be found in [20], where the procedure is also compared with other alternatives.

As remarked above, once the circuit has been partitioned, each processor takes the set of logic gates and lines that define the subcircuit assigned to it. These lines can be classified into several types: primary inputs, pseudo-inputs (noted as *pe* in Figure 6), internal lines, primary outputs, and pseudo-outputs (noted as *ps* in Figure 6). The pseudo-inputs and the pseudo-outputs, labelled with the symbol '/' in Figure 6, are the lines that have been cut to define the partitioning of the circuit under test. In this way, the set of inputs to each partition contains the primary inputs and the pseudo-inputs, and the set of outputs contains the primary outputs and the pseudo-outputs.

Once each processor has its corresponding subcircuit, it is possible for all the processors to apply the test generation algorithm concurrently in order to determine the f and g functions at the nodes in

Copyright © 2000 John Wiley & Sons, Ltd.



each subcircuit. These functions are termed *relative functions*, and are noted with the superindex r, and f^r and g^r. The processors also intercommunicate to determine functions f and g (also called *absolute functions*) for the whole circuit at the nodes of their partition. In Figure 6, the relative functions f^r and g^r for some lines of subcircuit 3 are also shown.

The relative functions in different subcircuits are computed concurrently by the processors, so if subcircuits have similar sizes and complexities, the time required to obtain these relative functions is also similar. Once the relative functions f^r of the subcircuit have been computed, the next step to be completed by the processor is to determine the absolute functions that correspond to the whole circuit. To do this, it is necessary to synchronize the processors that have the pseudo-inputs and pseudo-outputs corresponding to a given cut line. This synchronization implies that the processor that has to read the spectral coefficients from the memory has to wait for the processor having the corresponding pseudo-output to write it into the memory.

Considering the process of computing the absolute functions from the relative ones, the processors might have to wait for one another until the coefficients of the absolute functions in their pseudo-inputs have been determined by the processor where the corresponding subcircuit is assigned. Nevertheless, as the computation of the absolute functions f in the outputs and pseudo-outputs can start immediately after computing the functions f in the inputs and pseudo-inputs, it is possible to minimize this waiting time. Moreover, the effect of this waiting time can be reduced by partitioning the circuit into subcircuits with the lowest possible number of precedence dependencies. This is the reason for minimizing the cuts and starting the partitioning process from an initial solution which is built from the primary inputs (primary outputs) to the outputs (inputs) [20].

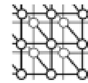
The g^r corresponding to different subcircuits can be computed concurrently with the f^r functions, and are obtained in terms of the primary outputs and pseudo-outputs of each subcircuit.

When the relative functions g^r have been obtained, the next step is to obtain the absolute functions g corresponding to the whole circuit. To do this, it is necessary to intercommunicate the processors having a pseudo-input and a pseudo-output that correspond to the same cut. In this case the coefficients are sent from the processor having the pseudo-input to that having the pseudo-output.

As in the computation of the absolute functions f, the waiting time of the processors is minimized as the computation of the g functions in an input or pseudo-input can start when the g functions of the outputs and pseudo-outputs of the circuit are available; it is also possible to reduce the waiting time by dividing the circuit starting from the outputs and minimizing the number of cuts. When the absolute functions are obtained, the test equation for each node of the circuit can be built and solved in the same way as in the serial procedure [13]. This can be done concurrently because each processor independently completes the computations corresponding to the nodes allocated to it.

Figure 7 shows the procedural description of the algorithm. The procedure `Divide_circuit` makes a static division of the circuit under test using a hybrid heuristic that mixes simulated annealing and tabu search techniques. The procedure `Computing_relative_functions` determines functions f^r and g^r corresponding to the nodes in the subcircuit assigned to the processor. These functions are relative to the corresponding subcircuit. `Computing_absolute_cuts` determines the absolute functions of the nodes where the cuts have been made to divide the circuit. The `Dynamic_allocation_lines` procedure performs a dynamic balancing of the load to compute the absolute functions. As a shared-memory multiprocessor is used, and all the data (i.e. basically the description of the circuit) can be accessed by all the processors, they can take the lines of the circuit dynamically as they conclude their present computation, thus providing

Copyright © 2000 John Wiley & Sons, Ltd.



```
Procedure 3
Begin
  #pragma one processor
     read_circuit
     divide_circuit
  #pragma parallel local (node) shared (circuit, test_vectors)
  begin
     #pragma concurrent call
     #pragma pfor iterate (i←0, npr, 1)
     begin
         Obtain_i-th_subcircuit
         Computing_relative_functions
       #barrier synchronization
         Computing_absolute_cuts (cuts_lines)
         Dynamic_allocation_lines
         Computing_absolute_internal (internal_lines)
         Determine_Test_Pattern
     #synchronization
       write test_vectors
    end_pragma_pfor
  end_pragma_parallel
End_procedure_3
```

Figure 7. Procedural description of algorithm based on circuit partitioning.

optimal load balancing. The procedure `Computing_absolute_internal` obtains the absolute functions f and g corresponding to the nodes assigned to the processor, while the procedure `Determine_Test_Pattern` builds the test equations and solves them as indicated in Section 2. During the concurrent execution of processes, there are situations in which the processes need to synchronize before the end of the execution. Synchronization is achieved by a barrier synchronization primitive. Each process waits at the barrier for every other process to reach its barrier primitive. After synchronization, all processes proceed with their execution. As can be seen, there are two points where the processors have to synchronize due to the sequential nature of the procedure, but the main advantage with respect to previously reported algorithms is that no redundant work is carried out.

In the next section we analyze the results obtained with these three procedures, when they are applied to various benchmark circuits.

4. EXPERIMENTAL RESULTS

The parallel test generators were run in an SGI Power Challenge computer with four R8000 CPUs at 75 MHz, 1 GB of main memory, and a 16 KB instruction cache and 16 KB data cache in each processor.

Copyright © 2000 John Wiley & Sons, Ltd.

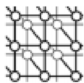


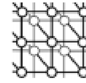
Table II. Speedup and coverage for some ISCAS circuits with 2 (S_2), 3 (S_3) and 4 (S_4) processors in a multiprocessor.

Circ.	Faults	Proced.	S_2_S	S_2_D	S_3_S	S_4_S	S_4_D	Cover
ALU	384	Proc_1	1.50	1.50	2.10	3.00	3.00	100%
		Proc_2	1.94	1.70	2.86	3.82	3.78	100%
		Proc_3	1.98	1.98	2.65	3.48	3.20	100%
c432	864	Proc_1	1.70	1.44	1.98	2.66	2.15	95%
		Proc_2	1.94	1.66	2.62	3.78	3.37	95%
		Proc_3	1.75	1.69	2.47	3.78	3.70	98%
c499	998	Proc_1	1.72	1.56	2.27	3.20	2.18	98%
		Proc_2	1.85	1.82	2.64	3.76	3.70	98%
		Proc_3	1.79	1.78	2.54	3.71	3.68	99%
c880	1760	Proc_1	1.77	1.65	2.31	2.72	2.39	100%
		Proc_2	1.94	1.93	2.85	3.94	3.65	99%
		Proc_3	1.88	1.82	2.56	3.91	3.87	100%
c1355	2710	Proc_1	1.75	1.66	2.14	2.52	2.36	97%
		Proc_2	1.94	1.92	2.80	3.86	3.82	97%
		Proc_3	1.98	1.90	2.74	3.47	2.93	98%
c1908	3816	Proc_1	1.43	1.33	1.83	2.49	2.15	97%
		Proc_2	1.92	1.90	2.94	3.66	3.50	97%
		Proc_3	1.96	1.86	2.85	3.39	2.50	98%
c3540	7080	Proc_1	1.87	1.67	2.41	3.20	2.87	97%
		Proc_2	1.73	1.64	2.78	3.80	2.79	96%
		Proc_3	1.81	1.78	2.69	3.79	3.67	98%
c6288	12570	Proc_1	1.41	1.24	2.21	3.10	2.96	96%
		Proc_2	1.74	1.54	2.72	3.66	3.21	95%
		Proc_3	1.76	1.67	2.67	3.52	2.75	97%

Some of the experimental results are better than those obtained in a multicomputer [19]. Table II shows some of the results obtained by Procedure_1 (Sect. 3.1), Procedure_2 (Sect. 3.2), and Procedure_3 (Sect. 3.3) for a representative subset of circuits from the sets ISCAS85 [21], ISCAS89 [22], plus the ALU74181 (noted as ALU in the table). In the table, **S_n_S** means the speedup achieved with *n* processors in the shared-memory multiprocessor and **S_n_D** indicates the speedup achieved with *n* processors in the distributed-memory multiprocessor (Intel Paragon) [19]. Cover indicates the percentage of detected faults.

Figure 8(a) shows that the speedups obtained by Procedure_2 are higher than those obtained by Procedure_1 and Procedure_3. Nevertheless, Procedure_3 provides slightly better fault coverage figures and, as it uses the memory more efficiently, it enables the processing of more complex circuits. Figure 8 (b, c and d) shows the speedups when the procedures here presented are compared with equivalent versions (there are some differences in the implementation) for a distributed-memory multicomputer.

Copyright © 2000 John Wiley & Sons, Ltd.



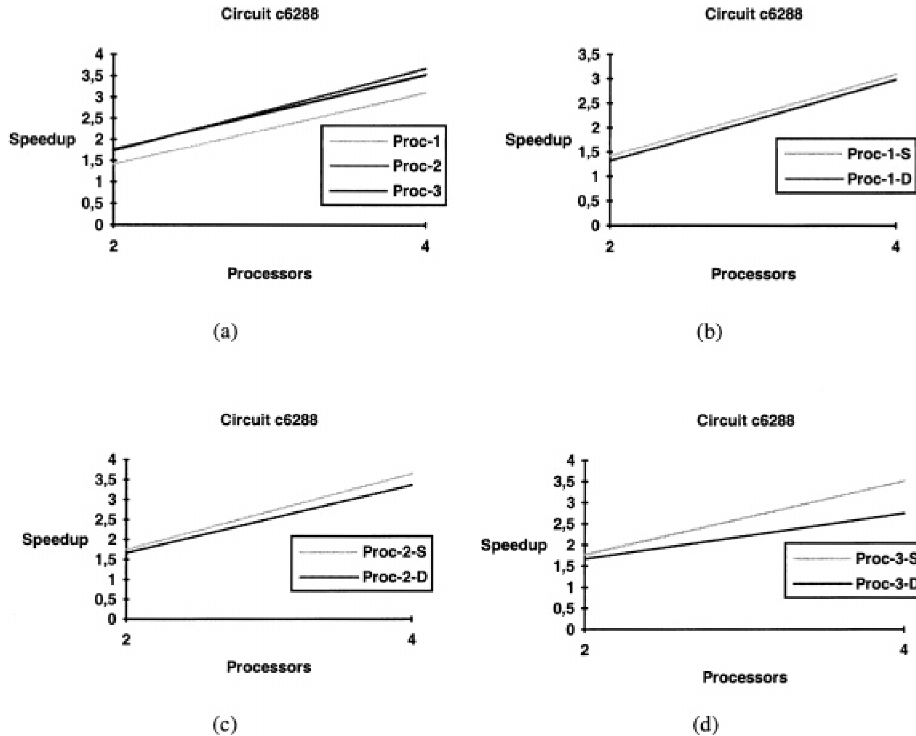


Figure 8. Speedups for circuit c6288 when (a) Procedure_1, Procedure_2 and Procedure_3 are compared in a shared-memory multiprocessor, and (b) Procedure_1, (c) Procedure_2 and (d) Procedure_3, respectively, are compared with a similar version for a distributed-memory multicomputer.

The main advantage is the use of memory and that in all procedures the speedups are better for a shared-memory address space.

5. CONCLUSIONS

We have presented three parallel procedures to speed up test pattern generation for digital circuits. These are based on a new test algorithm which operates in the Reed–Muller spectral domain, that allows an efficient determination of the test equations and facilitates their solution in order to determine the test patterns. As the amount of work involved in each fault is relatively easy to estimate, it is possible to implement an efficient static procedure to distribute the lines among the processors. The speedup values obtained show efficiencies close to one for our algorithm and good values for scalability.

When all the faults (not only the HTD faults) are considered, Procedure_2 shows better speedups than other parallel procedures. As our serial procedure requires similar amounts of time to the other

Copyright © 2000 John Wiley & Sons, Ltd.

serial procedures, such as PODEM, the speedups we have obtained represent a reduction in the time required to generate the test patterns with respect to parallel procedures.

The parallelization alternative presented in Procedure_3 is very interesting. Dividing the circuit among the processors allows the memory of the computer to be used in a very efficient way, and enables test patterns to be generated for circuits that cannot be analysed by serial procedures due to the great amount of time required, or by parallel procedures that would need a lot of memory. The results show good communication/computation rates, and thus high efficiency when the parallel procedure is applied to the circuits used as benchmarks.

By mixing a static circuit partitioning algorithm and a dynamic load balancing procedure, the speedup values obtained are better than those obtained by the parallel implementation of the same procedure in a distributed-memory multicomputer. To implement such a dynamic load balancing procedure in a distributed-memory multicomputer would result in a higher cost due to the computation requirements, while the speedup figures would be lower than those obtained by using only the static partitioning procedure.

To the best of our knowledge, only in [7] is the circuit under test divided among the processors of a specific purpose computer in order to determine, by circuit simulation, all the faults detected by a given test pattern, previously found by using a test procedure similar to PODEM [2]. In [23] a parallel test pattern generator is analysed which uses topological partitioning combined with the PODEM algorithm. The experimental results obtained show higher runtimes for the parallel test generator with respect to the execution in only one processor. As the authors of the paper indicate, this increase in the execution time is due to the overhead resulting from the implementation of the PODEM algorithm across the circuit partitioning because the number of elemental operations required by the PODEM algorithm is not decreased by the parallelization. These problems do not appear in our Procedure_3 because it is based on the new test algorithm we have developed, which is different from PODEM.

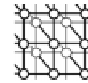
ACKNOWLEDGEMENTS

This paper has been partially supported by projects TIC97-1149 and TIC99-0361 (CICYT, Spain).

REFERENCES

1. Klenke RH, Williams RD, Aylor JH. Parallel-processing techniques for automatic test pattern generation. *Computer* 1992; **25**:71–84.
2. Goel P. An implicit enumeration algorithm to generate tests for combinational logic circuits. *IEEE Transactions on Computing* 1981; **30**:215–222.
3. Arvindam S, Kumar V, Rao N, Singh V. Automatic test pattern generation on multiprocessors: a summary of results. *Lecture Notes in Artificial Intelligence* 1990; 41–51.
4. Fujiwara H, Inoue T. Optimal granularity of test generation in a distributed system. *IEEE Transactions on Computer-Aided Design* 1990; **9**:885–892.
5. Fujiwara H, Inoue T. Optimal granularity and scheme of parallel test generation in distributed systems. *IEEE Transactions on Parallel and Distributed Systems* 1995; **6**:677–686.
6. Gil C, Ortega J. A parallel test pattern generator based on spectral techniques. *5th Euromicro on PDP*. IEEE Computer Society, 1997; 199–204.
7. Hirose F, Takayane K, Kamato N. A method to generate tests for combinational logic circuits using an ultra high speed logic simulator. *Proceedings 1988 International Test Conference*. CS Press: Los Alamitos, CA, 1988; 102–107.
8. Motohara A, Nishimura K, Fujiwara H, Shirakawa I. A parallel scheme for test pattern generation. *Proceedings IEEE International Conference Computer-Aided Design*. CS Press: Los Alamitos, CA, 1986; 156–159.

Copyright © 2000 John Wiley & Sons, Ltd.



9. Patil S, Banerjee P. Fault partitioning issues in an integrated parallel test generation/fault simulation environment. *International Test Conference*. IEEE Computer Society, 1989; 718–726.
10. Patil S, Banerjee P. A parallel branch and bound algorithm for test generation. *IEEE Transactions on Computer-Aided Design* 1990; **9**:313–322.
11. Smith S, Underwood B, Mercer MR. An analysis of several approaches to circuit partitioning for parallel logic simulation. *Proceedings IEEE International Conference on Computer-Design*. CS Press: Los Alamitos, CA, 1987; 664–667.
12. Wolf JM, Kaufman LM, Klenke RH, Aylor JH. An analysis of fault partitioned parallel test generation. *IEEE Transactions on Computer-Aided Design* 1996; **15**:517–534.
13. Gil C, Ortega J. Algebraic test-pattern generation based on the Reed–Muller spectrum. *IEE Proceedings Computer and Digital Techniques* 1998; **4**:308–316.
14. Stanion RT, Bhattacharya D, Sechen C. An efficient method for generating exhaustive test sets. *IEEE Transactions on Computer-Aided Design* 1995; **14**:1516–1525.
15. Damarla TR, Karpovsky M. Fault detection in combinational networks by Reed–Muller transforms. *IEEE Transactions on Computing* 1989; **38**:788–797.
16. Green DH. Families of Reed–Muller forms. *International Journal of Electronics* 1991; **70**:259–280.
17. Ortega J, Lloris A, Prieto A, Pelayo F. Test-pattern generation based on Reed–Muller coefficients. *IEEE Transactions on Computers* 1993; **42**:968–980.
18. Aarts E, Korst J. *Simulated Annealing and Boltzmann Machines. A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, 1990.
19. Glover F, Laguna M. Tabu search. *Modern Heuristic Techniques for Combinatorial Problems*, Reeves CR (ed.). McGraw-Hill: London, 1993; 70–150.
20. Gil C, Ortega J, Diaz AF, Montoya MG. Annealing-based heuristics and genetic algorithms for circuit partitioning in parallel test generation. *Future Generation Computer Systems Journal* 1998; **565**:1–13.
21. Brglez F, Fujiwara H. Neural netlist of ten combinational benchmark circuits and a target translator in FORTRAN. *Proceedings IEEE ISCAS'85*, 1985; 1929–1934.
22. Brglez F, Bryan D, Kozminski K. Combination profiles of sequential benchmark circuits. *Proceedings IEEE ISCAS'89*, 1989; 663–698.
23. Klenke RH, Williams RD, Aylor JH. Parallelization methods for circuit partitioning based parallel automatic test pattern generation. *IEEE VLSI Test Symposium*, 1993; 71–78.

Copyright © 2000 John Wiley & Sons, Ltd.