Accepted for publication in the IEEE Transactions on Computer-Aided Design, March 1990

A PARALLEL BRANCH AND BOUND ALGORITHM FOR TEST GENERATION

Srinivas Patil and Prithviraj Banerjee

Acknowledgment: This research was supported in part by the Joint Services Research Program under Contract N00014-84-C-0149 and in part by the Semiconductor Research Corporation under Contract SRC 87 DP-109.

S. Patil and P. Banerjee are with the Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

ABSTRACT

For circuits of VLSI complexity, test generation time can be prohibitive. Most of the time is consumed by hard-to-detect (HTD) faults which might remain undetected even after a large number of backtracks. We identify the problems inherent in a uniprocessor implementation of a test generation algorithm and propose a parallel test generation method which tries to achieve a high fault coverage for HTD faults in a reasonable amount of time. A dynamic search space allocation strategy is proposed which allocates disjoint search spaces to minimize the redundant work.The search space allocation strategy tries to utilize the partial solutions generated by other processors to increase the probability of searching in a solution area. The parallel test generation algorithm has been implemented on an Intel iPSC/2 hypercube. Results are presented using the ISCAS benchmark circuits which conclusively prove that parallel processing of HTD faults does indeed result in high fault coverage which is otherwise not achievable by a uniprocessor algorithm. The parallel algorithm exhibits superlinear speedups in some cases due to search anomalies.

ADDRESS FOR CORRESPONDENCE

Srinivas Patil
Computer Systems Group
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 W. Springfield Av.
Urbana, IL 61801
Phone: (217) 244-7166

LIST OF INDEX TERMS

Test generation, parallel processing, implicit enumeration, parallel search, hypercube multiprocessors.

LIST OF FIGURE CAPTIONS

Figure 1. Search space for test generation

Figure 2. Decision tree in PODEM

Figure 3. Splitting search spaces

Figure 4. Stacks after splitting

Figure 5. Implementation on the hypercube

Figure 6. Run times using randomized search

Figure 7. Speedups for HTD faults

LIST OF TABLE CAPTIONS

Table I. Uniprocessor results for RANDOM heuristic

Table II. Uniprocessor results for SCOAP heuristic

Table III. Parallel results using RANDOM heuristic

Table IV. Parallel results using SCOAP heuristic

Table V. Performance for redundant faults of c2670

Table VI. Dropped faults on 16 processors

1. INTRODUCTION

Given a logic circuit, the objective of a test generation algorithm is to derive a set of input patterns which are able to distinguish the fault-free circuit from the faulty one in the presence of a set of faults. Thus, we say that an input pattern is a test for a fault if in the presence of the fault, the faulty circuit produces an output different from the fault-free circuit when the input pattern is applied. The fault is then said to be detected by the input pattern. The fault model normally adopted is the stuck-at model where a line or gate output is assumed to be permanently set to a logic value in the presence of faults. Other fault models are available but test generation becomes more difficult as the fault model becomes complex. We shall assume the stuck-at model which is one of the simplest. We shall also assume combinational or memoryless circuits.

The search space for test generation is exponential in the number of primary inputs of a circuit. The problem of test generation has been proven to be NP-complete[1] even for combinational circuits and as a result, algorithms for test generation normally use heuristics to guide the search for a test. One of the most widely used algorithms is the D-algorithm [2]. But D-algorithm has been shown to be inefficient in the case of error correction and translation (ECAT) circuits which are characterized by a large number of Exclusive-Or gates. An algorithm called PODEM [3] was proposed to remedy this problem. This algorithm views the test generation process as an implicit enumeration of an n-dimensional boolean search space (n is the number of primary inputs of the logic circuit). A variant of PODEM called the FAN algorithm [4] was proposed which uses much better heuristics to guide the search. More recently, some simulation based methods have been proposed which use a cost function to guide the search for a test vector [5]. For circuits of VLSI complexity, the test generation for a whole chip may take days and even weeks to complete. Improved heuristics lead to only a marginal improvement in run time for most of these algorithms. The only way to speed up the task of test generation by orders of magnitude is to use multiprocessors [6, 7]. With general purpose multiprocessors becoming more commonplace, this seems to be a very attractive alternative.

Parallel search techniques have been applied to 0-1 Integer Linear Programming (0-1 ILP) [8] and the 15-puzzle problem [9, 10]. In this paper we present a parallel branch and bound formulation of test generation. We then present our results based on implementation of PODEM using this strategy on a 16 processor Intel iPSC/2 hypercube using the ISCAS benchmark circuits. We show through experiments the resulting improvement in fault coverage as well as time efficiency due to the proposed technique. Due to search anomalies, the parallel algorithm exhibits superlinear speedups in some cases. Throughout our discussion we shall assume that the reader is familiar

with test generation algorithms, specifically PODEM [3].

2. EXPLOITING PARALLELISM IN TEST GENERATION

There are various ways in which we can parallelize test generation. *Fault parallelism* refers to the evaluation of the given fault set in parallel. In this method, the fault set is divided equally among *n* processors. Each processor can now generate tests for its own fault set independently. But this method suffers from a serious drawback as far as the overall test length and run time is concerned. In an integrated test generation/ fault simulation environment, test generation and fault simulation are interleaved to reduce the overall run time and to produce a compact test set. Thus a parallel implementation should include fault simulation also. Each processor fault simulates test vectors derived during its own test generation phase and by other processors. Communication among processors for fault simulation can be reduced or altogether eliminated by dividing the fault sets into independent fault sets [11]. This reduces the likelihood of a test vector derived by one processor being a test for a fault on some other processor. The main advantage of fault parallelism is that the communication overhead is very low and it is possible to achieve linear speedups when the number of processors is very small compared to the number of faults. But the main disadvantage is that faults which are hard to detect for the uniprocessor algorithm (i.e., faults which require a large number of backtracks on the uniprocessor) remain undetected.

As mentioned earlier, almost all test generation algorithms use some heuristic or the other to guide test generation. Experiments reported have suggested that there is no clear cut advantage of using one heuristic over the other [12, 13]. In what we call *heuristic parallelism* each processor can use a different heuristic to guide the search for the same fault. Superlinear speedups have been predicted in some cases [14]. But the main disadvantage of this method is that the parallelism is limited by the number of heuristics available for search (at most 5-6). Also, no improvement is possible if a fault remains undetectable for all the heuristics. Communication overheads are low in this case also.

One major component of run time in test generation is forward implication. Every primary input assignment is followed by forward implication which takes $O(n)$ time in the worst case (n is the number of lines in the logic circuit). This makes backtracks expensive and the test generation process has to be aborted when a test cannot be generated after a certain backtrack limit. Parallel logic simulation methods can be used to speed up forward implication. The circuit can be partitioned across processors so that each processor simulates a part of the circuit. The partition-

ing strategy should minimize communication and maximize concurrency. We call this type of parallelism *simulation parallelism*. For gate level test generators, the communication latency might be an order of magnitude higher than the gate evaluation time. Also, the circuit activity in a test generation environment may be low. Thus on a hypercube multiprocessor, gate level simulation parallelism does not hold too much promise (speedups of at most 2-3). However, for switch level test generation and simulation based methods, where the activity per gate is much higher, simulation parallelism still holds promise.

Another way to view the test generation process is as a series of subtasks. This leads to an AND/OR type parallelism as described in [15, 16]. *AND parallelism* arises in algorithms using a divide and conquer type approach where a task is divided into a series of subtasks, all of which have to be completed. Thus these subtasks can be completed in parallel if they do not access or modify any common variables. If they have common variables, then execution of the AND-parallel subtasks becomes complicated due to variable binding conflicts. The overheads due to variable binding conflicts may essentially reduce the AND-parallel tasks to sequential execution due to consistency checking etc. On a hypercube, this may lead to excessive communication. In a logic circuit, reconvergent fanout is one source of variable binding conflict.

OR parallelism refers to evaluating several choice points in parallel. In the case of test generation, this essentially means that we are searching different portions of the search space concurrently. Each processor can be allocated a portion of the search space, either statically or dynamically and the search for the test vector can proceed concurrently. A brute force method of dividing the search space might entail useless search in non-solution areas by some processors. A heuristics based method aimed at increasing the probability of searching in a solution area would be more desirable but the search spaces may not be disjoint in such a situation. The advantage of OR parallelism is that communication overhead is very low and a large number of processors can be utilized efficiently. Also, the backtracks per processor can be reduced. Superlinear speedups might be possible due to search anomalies [17, 18]. A parallel logic verification algorithm using the implicit enumeration technique of PODEM was proposed in [19]. A dynamic search space allocation strategy was proposed which tries to minimize the redundant work done by different processors by assigning disjoint subspaces. The method proposed in this paper also uses a similar search space allocation strategy but the difference lies in the basic structure of the algorithm. The parallel algorithm proposed in [19] uses a divide-and-conquer strategy in which the whole input space of the of the logic circuit being verified has to be enumerated in most cases. However, in the case of test generation, the enumeration process is

stopped as soon as a solution is found. Thus, for efficient utilization of available processors, the search space allocation strategy becomes very important. Processors should be allocated only those portions of the original search space which maximize the chance of finding a solution. A parallel search method to search different portions of the decision tree in parallel was proposed in [6] but the search space allocation strategy did not use any heuristics to increase the probability of searching in a solution area. This may result in fruitless search for a vector in a non-solution area even though a solution may exist. A dedicated processor was used which schedules partially specified test vectors for test generation. The proposed parallel branch and bound method is based on the same principle as [6] i.e., searching different portions of the search space concurrently. However, the scheduling algorithms are much more efficient and a novel search space allocation strategy is proposed which tries to increase the probability of searching in a solution area. The search space allocation strategy is based on perturbing the partial solution already generated by some other processor. A search space splitting strategy is proposed which tries to ensure that the choice points for backtracking are distributed equally across all the available processors.

3. TEST GENERATION USING PARALLEL BRANCH AND BOUND

PODEM treats test generation as a branch and bound search on the n-dimensional Boolean input space of a logic circuit. A conceptual representation of the search space is shown in Figure 1. The big outer circle indicates the entire search space of size 2^n where n is the number of primary inputs of the logic circuit. The smaller circles inside the big circle indicate subspaces containing a test for the particular fault under consideration. The solid directed path indicates the path taken by the test generation algorithm to find a test for the fault. The length of the path is proportional to the time taken to find the test for the fault. The dotted paths indicate alternate paths which could have been taken to generate a test in shorter time. Even though a test may exist for a fault, the search may have to be terminated after a predecided backtrack or time limit to save on time. One way in which test generation algorithms try to follow paths of shorter lengths is by using testability measures[20, 21] to guide the search. But the problem of calculating exact testability measures for a logic circuit with reconvergent fanout has an exponential worst case time complexity [22]. Also, it might be necessary to recalculate testability measures during the test generation process as the circuit state changes. It has been experimentally shown that this does reduce the number of backtracks but the overheads due to recomputing the testability measures every time a primary input assignment is made overshadow any gains due to fewer number of backtracks [23]. Since it is not practical to calculate exact testability measures

and recomputing testability measures is expensive, most test generation algorithms use static and approximate testability measures. The heuristic guidance provided by approximate static testability measures may not lead to a solution along a shorter path even though one might exist. As suggested in [12-14] the alternative would be to use multiple testability measures. On a uniprocessor one may switch between different heuristics or on a multiprocessor implementation we may be able to pursue solutions using different heuristics in parallel. The main drawback of such a method is that there is no way to ensure that the search spaces are disjoint. Wrong paths taken by one heuristic might be repeated by another heuristic and the fault might remain undetected even if a test exists. As pointed out earlier, on a multiprocessor, the parallelism is limited by the number of heuristics available for search and two processors running on two different heuristics might end up doing redundant work because their search spaces may not be disjoint. The multiprocessor implementation offers no improvement if a set of faults remains undetected by all heuristics. The method we propose offers an improvement over the uniprocessor algorithm both in terms of time and fault coverage irrespective of the heuristic used and it is possible to utilize a large number of processors efficiently.

The decision tree used by PODEM is shown in Figure 2. A 0 or a 1 along a solid edge indicates the current assignment to a primary input. It is a binary tree maintained as a stack where each edge corresponds to a 0 or a 1 assignment to a primary input. Each stack entry has a three fields - a *val* field storing the current assignment to a primary input, an *input* field identifying the primary input and a *tried* field indicating whether the alternative value at the primary input has been tried. The dashed edges indicate unexplored subspaces. The algorithm proceeds in a depth-first manner to implicitly enumerate the search space for a test vector. If a conflict occurs during this process, the algorithm backtracks to a previous choice point to try another alternative (points with a 0 in the *tried* field). It has been shown experimentally in [12] that it is very unlikely that a fault which has not been detected by heuristic with a lower backtrack limit will be detected by increasing the backtrack limit. There is a very fundamental reason to this observation. When a conflict occurs, the assignment which gave rise to the conflict need not necessarily be the assignment made at the previous choice point. A blind backtrack to the previous choice point may lead to more fruitless backtracks. So, the algorithm would perform much better if one were to backtrack to the point which gave rise to the conflict. This is easier said than done. It is very difficult to determine which previous assignment gave rise to the conflict. The proposed parallel implementation tries to divide the search space so that intermediate choice points are skipped by one processor while at the same time the skipped choice points are evaluated by some other processor. This leads to a much better fault coverage figure for hard-to-detect (HTD) faults compared to the unipro-

cessor implementation. Also, the search space is divided so that all processors are searching disjoint subspaces at all times. Since the size of the search space is exponential in the number of primary inputs, many processors can be utilized efficiently.

Figure 3 shows how a decision tree belonging to processor i would be split between processor i and another processor j. The choice points are alternately split between processor i and processor j so that each processor has roughly equal number of choice points. The exact mechanism by which these search spaces are allocated is described in the next section. A 0 in the *tried* field of a stack entry for a processor indicates that the processor has been allocated the particular choice point. Figure 4. shows the stacks after splitting.

4. IMPLEMENTATION DETAILS

The PODEM-based test generator using parallel branch and bound was implemented on a 16 node Intel iPSC/2 hypercube. A dynamic search-space allocation strategy was used to keep the load balanced at all times. The nodes in an n-dimensional hypercube are numbered from 0 to $(2^d - 1)$ as shown in Figure 5. One of the nodes (node 0) is used for scheduling (known as a *scheduler*) while nodes 1 to $(2^d - 1)$ carry out test generation using parallel branch and bound (known as *processing* nodes). Initially, the circuit description and information about the heuristic to be used is broadcast by the host to all the nodes. For each fault, the host broadcasts the gate-under-test and the type of fault (s-a-1 or s-a-0) and waits for the result from the scheduler. The processing nodes communicate with the scheduler to get more work or to inform the scheduler of special conditions. The types of messages passed between the scheduler and the processing nodes are as follows:

NOTEST

Sent by a processing node to the scheduler to indicate that a test is not possible for a particular fault with the given search subspace. Also sent by the scheduler to the host to indicate that no test exists for the given fault.

ABORT

Sent by a processing node to a scheduler if a processing node aborts test generation after hitting the backtrack limit. Also sent by the scheduler to the host to indicate that test generation for the given fault had to be aborted because the backtrack limit was exceeded.

TESTED

Sent by a processing node to a scheduler if a processing node generates a test for the given fault. Also sent by

the scheduler to the host if a test is generated for the given fault.

WRK_REQ

Sent by the scheduler to a processing node to request for more work.

DONE

An abort message sent by the scheduler to all processing nodes to stop test generation for a particular fault. Also sent by the processing nodes to the scheduler to acknowledge a DONE message.

WORK

A message sent by a processing node to a scheduler in response to a WRK_REQ message. Also used by the scheduler to send work to a processor which requested for work through a NOTEST message.

The function of the scheduler is to coordinate the activities of the processing nodes. It receives messages from the processing nodes and carries out actions based on the type of message. The host broadcasts gate-under-test and the fault type (s-a-1 or s-a-0) to each node. The scheduler maintains a list of processing nodes which are busy generating a test for the current fault. This list is maintained in the form of a queue and is called a *donor queue*. Initially, node 1 starts generating a test for the fault assuming it has the whole search space. At the beginning of test generation for a particular fault, the donor queue is initialized to include node 1. The remaining processing nodes do not send an explicit message for requesting work but send a NOTEST message indicating that the search space is empty and that they require more work. The actions executed by the scheduler on receiving a NOTEST message is given by the following procedure:

procedure *allocate-work*:

(1) If there is an entry for the node that sent the NOTEST message in the donor queue then delete it from the donor queue.

(2) Select a donor from the top of the donor queue. If the donor queue is empty then enter the *wrap-up phase*. Otherwise, put the donor at the end of the queue. This is to make sure the same node is not asked repeatedly for work and the search spaces being processed by each node are uniformly distributed over the initial search space.

(3) Send a WRK_REQ message to the donor. Wait for work to arrive from the donor. Send the work to the node that sent a NOTEST message. If the work received is non-null, then put the node that sent the NOTEST message at the end of the donor queue. If the work received is null, then delete the donor from the donor queue.

end *allocate-work*.

On receiving an ABORT message, the scheduler deletes the node which sent the ABORT message from the donor queue. If the donor queue becomes empty after this action, then the scheduler enters the wrap-up phase. On receiving a TESTED message from a node, the scheduler enters the wrap-up phase unconditionally.

The actions executed by the scheduler in the wrap-up phase are as follows:

procedure *wrap-up*:

(1) Broadcast a DONE message to all the processing nodes. Then the scheduler waits for a DONE message from each of the nodes. This is to make sure all the messages from the processing nodes for the current fault have been flushed.

(2) If any of the processing nodes sent a TESTED message, then send the test vector to the host. Otherwise, if any of the processing nodes sent an ABORT message, then send an ABORT message to the host else send a NOTEST message.

end *wrap-up*.

The processing nodes can either be in the *waiting phase* or *PODEM phase* after receiving the gate-under-test and the fault type from the host. In the waiting phase, a node waits for work from the scheduler. In the PODEM phase a node executes PODEM on its own search phase to generate a test for the given fault. After receiving the current fault from the host, node 1 enters the PODEM phase directly assuming it has the whole search space. Other processing nodes send a NOTEST message to the scheduler indicating they need work.

The actions performed by the processing nodes in PODEM phase are as follows:

procedure *podem*:

(1) In the main PODEM loop periodically check for messages from the scheduler.

(2) If a test is generated for the given fault, send a TESTED message along with the test vector to the scheduler. Enter the *wind-up phase*.

(3) If backtrack limit is exceeded, send an ABORT message to the scheduler and enter the wind-up phase.

(4) If the search space is exhausted, send a NOTEST message to the scheduler and enter the waiting phase.

(5) If a WRK_REQ message is received, then split the stack as shown in Figure 4 and send it to the scheduler.

(6) If a DONE message is received, enter the wind-up phase.

end *podem*.

The actions performed by the processing nodes in the waiting phase are as follows:

procedure *wait-for-work*:

(1) If a WRK_REQ message is received, send a WORK message of size zero to the scheduler to indicate no work is available.

(2) If a WORK message is received and the size of the work is non-zero, then enter the PODEM phase. The stack in the PODEM phase is initialized to the stack received as a part of the WORK message. The processing node then performs a forward implication using the values in the received stack to initialize the logic values in the circuit properly. It then computes the D-frontier and enters the PODEM phase. If the size of the WORK message is zero, then send a NOTEST message to the scheduler.

(3) If a DONE message is received enter the wind-up phase.

(4) Go to 1.

end *wait-for-work*.

The actions performed by the processing nodes in the wind-up phase are as follows:

procedure *wind-up*:

(1) Send a DONE message to the scheduler. This is to indicate that the processing nodes are ready to process the next fault.

(2) Wait for the next target fault from the host.

(3) If the target fault is NULL (i.e., the host fault list is empty) then exit from the test generation routine else enter the PODEM phase.

end *wind-up.*

Dynamic load balancing is implicit in the scheduling algorithms given above. The scheduler is not allowed to execute PODEM to make sure that the scheduling node does not become a bottleneck as the number of processors is increased. The maximum length of messages between the scheduler and the processing nodes is limited to about 200 bytes to reduce the communication latency. Thus it can be expected that an idle processor requesting work is idle for a very short time. The delay between the time a request for work is sent out and the time when work is received from the scheduler is proportional to:

$$2C_{stack}(d_{is} + d_{sd})$$

where d_{is} is the distance in hops between the scheduler and the idle processor and d_{sd} is the distance between the scheduler and the donor. C_{stack} is the cost of sending the stack over one hop. For example, in the case of a hypercube with N processors, the maximum latency between request for work and the actual receipt of the work by the requesting processor can be shown to be:

$$2\lambda n[2\log_2(N) - 1]$$

where n is the number of primary inputs in the logic circuit. The value of C_{stack} has been replaced by the quantity λn and the quantity d_{is} has been replaced by the $\log_2(N)$ which is the maximum distance in hops between any two nodes in the hypercube. It follows that if the idle processor is at a distance $\log_2(N)$ from the scheduler then the donor can be at most $\log_2(N) - 1$ from the scheduler. We therefore substitute the quantity $\log_2(N) - 1$ for d_{sd}. It should be noted however that the processing time at the scheduler itself has been ignored to derive the above expressions and has been assumed to be negligible compared to the communication time. Due to search anomalies, superlinear speedups are possible as borne out by results presented in the following sections.

5. THEORETICAL ANALYSIS

It is very difficult to come up with a theoretical estimate of the speedups obtained by a non-deterministic method such as the parallel branch and bound method. However, it is possible to come up with analytical expressions for speedups based on some simplifying assumptions. Two such approaches are presented in this section.

5.1. Divide-and-conquer approach

Assume that the test generation algorithm searches the whole non-solution area before finding a test vector. This may not be a very unrealistic assumption for hard-to-detect and redundant faults. Consider an HTD (or redundant) fault f_i whose detection probability is p_d. If the number of primary inputs in the logic network is n, then the number of nodes expanded before a solution is found is:

$$(1 - p_d)2^n$$

If the computation cost per node is C_{node}, then the total computation cost on a uniprocessor is:

$$T_1 = (1 - p_d)2^n C_{node}$$

Also, if N processors are used, under assumptions of perfect load balance, the computation cost on N processors is:

$$T_N = \frac{(1-p_d)2^n C_{node}}{N}$$

To estimate the communication cost, assume that there are $N = 2^d$ processors over which the work load is to be distributed. Thus, if W was the work load on the uniprocessor, the final work load on each processor after each splitting should be $\frac{W}{2^d}$. It can be shown that the minimum number of stack splittings required such that each of the processors gets a work of $\frac{W}{2^d}$ is

$$2^d - 1 = N - 1.$$

If the cost of splitting the stack is C_{stack} then the total communication cost using N processors is:

$$(N-1)C_{stack}$$

After each processor processor receives the stack, it has to initialize the primary inputs to the values in the stack to restore the circuit status. This initialization cost is denoted by C_{init}. Thus the total initialization cost using N processors is:

$$(N-1)C_{init}$$

Here it has been assumed that all N processors participate in test generation while in reality only $(N-1)$ processors participate in test generation and one node acts as a scheduler. However, this approximation can be ignored for large N. Thus, the total time taken on N processors is:

$$T_N = \frac{(1-p_d)2^n C_{node}}{N} + (N-1)(C_{stack} + C_{init})$$

and the speedup on N processors for an HTD fault whose detection probability is p_d is given by:

$$S_N = \frac{T_1}{T_N} = \frac{(1-p_d)2^n C_{node}}{\frac{(1-p_d)2^n C_{node}}{N} + (N-1)(C_{stack} + C_{init})}$$

The quantity C_{node} is the cost of simulating the logic circuit after each primary input input assignment and can be substituted by αG where G is the number of gates in the logic circuit. The quantity C_{stack} is proportional to the number of primary inputs in the logic circuit (in the worst case) and can be replaced by the quantity λn. The quantity C_{init} is the cost of initializing primary inputs of the circuit to the values in the stack and can be replaced by the quantity βG. Thus the speedup is now given by:

$$S_N = \frac{(1-p_d)2^n \alpha G}{\frac{(1-p_d)2^n \alpha G}{N} + (N-1)(\lambda n + \beta G)}$$

Thus, it can be seen the if the number of primary inputs n is very large, the communication cost in the above

expression is negligible.

5.2. Probabilistic analysis

This analysis is very similar to the analysis for randomized parallel branch and bound presented in [24]. Let the probability distribution for solution time for a single fault on the uniprocessor be $F(t)$. On N processors, let the probability distribution for the solution time on processor i be $F_i(t)$. Thus, the expected solution time on the uniprocessor is:

$$\int_0^\infty [1-F(t)]dt$$

and the expected solution time on N processors is:

$$\int_0^\infty [1-F_1(t)][1-F_2(t)].....[1-F_N(t)]dt$$

Adding the communication and initialization costs, the speedup on N processors is given by:

$$S_N = \frac{\int_0^\infty [1-F(t)]dt}{\int_0^\infty [1-F_1(t)][1-F_2(t)].....[1-F_N(t)]dt + (N-1)(\lambda n + \beta G)}$$

Thus, speedups can be calculated by substituting the actual probability distributions for $F(t)$. However, it is very difficult get a closed form expression for speedup for most realistic probability distributions like normal distribution.

However, simpler closed form expressions can be derived based on experimental observations.It was shown experimentally in[12] that either a heuristic finds a test for a particular fault in a small amount of time or takes a prohibitively long time due to useless backtracking. Figure 6 shows the distribution of test generation times for a randomly selected fault for three of the ISCAS combinational benchmark circuits. Test generation was done for each fault using 100 different random seeds for random number generation to simulate randomized search. The numbers along the vertical axis give an estimate of the probability of finding a test vector during a given time assuming randomized search. It can be seen that the distribution of run times for all three circuits is not uniform. The run times are centered around two different times: a small CPU time t_l and a large CPU time t_h. This supports the experimental results reported in [12]. Thus, it can be assumed that each processor either finds a test vector for a fault (using randomized search) in a short time t_l with probability p or takes a long time t_h with probability $1-p$. Thus, the expected time it takes to find a test for a fault on a uniprocessor is given by:

$$pt_l + (1-p)t_h$$

On the multiprocessor, the expected time to find a solution on N processors is given by (ignoring communication and initialization costs):

$$[1-(1-p)^N]t_l + (1-p)^N t_h$$

Adding the communication and initialization costs, the expected speedup on N processors is given by:

$$S_N = \frac{pt_l + (1-p)t_h}{[1-(1-p)^N]t_l + (1-p)^N t_h + (N-1)(\lambda n + \beta G)}$$

It should be noted that the probabilistic analysis presented above ignores the fact that the search space is reduced for each processor after each splitting of the search space. For redundant faults, where the parallel algorithm reduces to a divide-and-conquer approach, the analysis presented in Section 5.1 is more appropriate.

6. EXPERIMENTAL RESULTS

The parallel algorithm was implemented on a 16 processor Intel iPSC/2 hypercube in C language. The hard-to-detect (HTD) faults were first filtered out using a uniprocessor implementation of PODEM on the host. The host and the nodes have a 4 MIPS Intel 80386 processor. A backtrack limit of 25 was used for the uniprocessor algorithm and all faults dropped by a particular heuristic were classified as HTD faults. To demonstrate that the efficiency of the parallel algorithm is independent of the testability measure used, both random selection (heuristic RANDOM) and the SCOAP [20] heuristic were used. It has been shown in [12, 13] that SCOAP performs better than other heuristics on the average. The results of the uniprocessor implementation on the ISCAS benchmark circuits[25] are shown in Tables I and II. Table I shows the results for RANDOM heuristic and Table II shows the results for SCOAP heuristic. The average time per fault was calculated by dividing the time taken by the algorithm for all the faults (including dropped or HTD faults) by the total number of faults. The fault coverage figure is not the fault coverage in the fault simulation sense since no fault simulation was used here. The fault coverage is rather an efficiency figure for the test generation algorithm and is equal to the sum of the faults for which tests were generated and faults which were found redundant divided by the total number of faults. As shown in Tables I and II, the uniprocessor algorithm is as efficient as the implementations reported in[13, 26] both in terms of fault coverage and time per fault.

The faults dropped by the RANDOM and SCOAP heuristics were used as HTD faults for the next part of our experiments. Circuit c880 was removed from further consideration since both heuristics generated tests for 100% of the faults. Tables III and IV show results for HTD faults using parallel branch and bound on 4, 8 and 16 processors.

Table III shows the results for RANDOM heuristic and Table IV shows the results for SCOAP. For a fair comparison with the uniprocessor results, the total backtrack limit over all processors was kept the same as the backtrack limit for the uniprocessor implementation. In the parallel implementation, the backtrack limit per processor was kept at 25. For example, since the total backtrack limit over 4 processors is 3×25=75 (one processor acts as a scheduler and does not take part in test generation), the uniprocessor backtrack limit was increased to 75. Similarly, for comparison with the 8 processor results, the uniprocessor backtrack limit was 175 and for the 16 processor results, the uniprocessor backtrack limit was 375. The parallel and the uniprocessor algorithms with the scaled up backtrack limits were run on the remaining 9 ISCAS circuits for the HTD faults given in Tables I and II. The purpose of these experiments was three-fold - (1) to show that increasing the backtrack limit for the uniprocessor algorithm does not yield substantial improvements in fault coverage for HTD faults, (2) the parallel implementation yields a better fault coverage in most cases than the uniprocessor algorithm for the same total backtrack limit and (3) the parallel algorithm runs much faster than the uniprocessor algorithm. The numbers in parenthesis in Tables III and IV are the corresponding numbers for the uniprocessor algorithm with the same backtrack limit as the parallel implementation. It is quite clear that the parallel implementation performs much better than the uniprocessor version in most cases, both with respect to the HTD fault coverage and the time per HTD fault. In some cases, the speedups are superlinear due to search anomalies (e.g., c1908 in the case of RANDOM and c6288 in the case of SCOAP) as shown by the graph in Figure 7. The message passing overheads are low since messages are passed only to get work and to inform the scheduler of completion. Due to the nondeterminism introduced by parallel execution, sometimes increase in the number of processors can cause a small decrease in the fault coverage (e.g. results for c1355 in Table III for 8 and 16 processors). Only in one case does the parallel algorithm have a fault coverage figure less than the uniprocessor algorithm (c2670 in Table III with 16 processors). But on the average, the parallel implementation performs much better than the uniprocessor implementation. The overall coverage figures are obtained by adding the HTD coverage to the corresponding coverage in Tables I or II. The overall coverage for the parallel implementation ranges between 97 to 99.9% in most cases which would be considered a very high figure for most test generation algorithms.

It should be pointed out here that for the special case of redundant faults, the parallel algorithm will have to search through the *same* nodes as the serial algorithm before proving the fault to be redundant. This is because for redundant faults, the entire search space will have to be implicitly enumerated. Thus, the speedup for redundant

faults will be at most N where N is the number of processors. Table V shows the performance of the parallel and serial algorithm for 24 of the known redundant faults for the circuit c2670, using the SCOAP heuristic. It can be seen that the parallel algorithm can speed up redundancy identification also. However, the efficiency of the parallel algorithm is limited by the efficiency of the serial algorithm and more efficient techniques for redundancy identification similar to the ones reported in[27, 28] will be required.

The total number of faults reported redundant and the total number of faults dropped by RANDOM and SCOAP after running the parallel algorithm on 16 processors are shown in Table VI. For comparison, the redundant faults detected by special redundancy identification techniques reported in[28] for the ISCAS circuits are also shown (under the column labeled SOCRATES). It can be seen that many of the dropped faults in Table V have in fact been found to be redundant in [28]. For example, all 4 dropped faults for c432 using SCOAP are redundant. Similarly, all 8 dropped faults for c499 using SCOAP are redundant. Thus, the fault coverage figures would be much higher for most circuits (100% for some circuits) if redundancy identification were used in the preprocessing phase. Also, the faults dropped after using a large number of processors are more likely to be redundant.

The splitting technique of Figure 4 could also be used as an effective heuristic on the uniprocessor to increase the fault coverage. A random number generator could be used to split the decision tree and to switch between search spaces in a non-deterministic manner to emulate the behavior of the parallel algorithm. Of course, the parallel algorithm would still do better in terms of the overall run time.

7. CONCLUSIONS

We identified the problems inherent in the implementation of a uniprocessor algorithm for HTD faults. Finally, we proposed a parallel branch and bound formulation of test generation which overcomes the deficiencies of the uniprocessor algorithm. Through experiments using the ISCAS benchmark circuits it was shown that the proposed parallel branch and bound technique is able to achieve a greater fault coverage than the uniprocessor algorithm at a fraction of the time consumed by the uniprocessor implementation. In some cases, the parallel implementation showed a superlinear speedup. Thus, parallel processing of HTD faults offers a very cost effective way to achieve high fault coverage. The search space splitting technique could be used as a very effective heuristic for the uniprocessor algorithm to increase the fault coverage. In our current implementation, the central scheduler coordinates the activities of the processing nodes. For larger number of processors, the scheduler could become a

bottleneck in the performance of the parallel algorithm. We are investigating techniques for distributed scheduling in a larger system. Our final goal is to build a very efficient parallel test generation system incorporating redundancy identification, parallel fault simulation and parallel test generation.

REFERENCES

[1] H. Fujiwara and S. Toida, ''The complexity of fault fetection: An approach to design for testability,'' *Proc. 12th Int. Symp. Fault Tolerant Computing*, pp. 101-108, June 1982.

[2] J. P. Roth, ''Diagnosis of automata failures: A calculus and a method,'' *IBM J. Research Development*, vol. 10, pp. 278-291, July 1966.

[3] P. Goel, ''An implicit enumeration algorithm to generate tests for combinational logic circuits,'' *IEEE Trans. Computers*, vol. C-30, pp. 215-222, March 1981.

[4] H. Fujiwara and T. Shimono, ''On the acceleration of test generation algorithms,'' *IEEE Trans. Computers*, vol. C-32, pp. 1137-1144, December 1983.

[5] K. T. Cheng and V. D. Agrawal, ''A simulation-based directed search method for test generation,'' *Proc. Int. Conf. Computer Design*, pp. 48-51, October 1987.

[6] A. Motohara, K. Nishimura, H. Fujiwara, and I. Shirakawa, ''A parallel scheme for test-pattern generation,'' *Proc. Int. Conf. Computer-Aided Design*, pp. 156-159, November 1986.

[7] G. A. Kramer, ''Employing massive parallelism in digital ATPG algorithms,'' *Proc. Int. Test Conf.*, pp. 108-114, 1983.

[8] T. S. Abdelrahman and T. N. Mudge, ''Parallel branch and bound algorithms on hypercube multiprocessors,'' *Proc. 3rd Conf. Hypercube and Concurrent Computers and Applications*, vol. II, pp. 1492-1499, January 1988.

[9] V. N. Rao and V. Kumar, ''Parallel depth first search, Part I: Implementation,'' *Int. J. Parallel Programming*, vol. 16, 1987.

[10] V. N. Rao and V. Kumar, ''Parallel depth first search, Part II: Analysis,'' *Int. J. of Parallel Programming*, vol. 16, 1987.

[11] S. B. Akers, C. Joseph, and B. Krishnamurthy, ''On the role of independent fault sets in the generation of minimal test sets,'' *Proc. Int. Test Conf.*, pp. 1100-1107, 1987.

[12] S. Patel and J. Patel, ''Effectiveness of heuristics measures for automatic test pattern generation,'' *Proc. 23rd Design Automation Conf.*, pp. 547-552, 1986.

[13] S. J. Chandra and J. H. Patel, ''Experimental evaluation of testability measures for test generation,'' *IEEE Trans. Computer-Aided Design*, vol. 8, pp. 93-98, January 1989.

[14] S. J. Chandra and J. H. Patel, ''Test generation in a parallel processing environment,'' *Proc. IEEE Int. Conf. Computer Design*, October 1988.

[15] D. DeGroot, ''Restricted AND-parallelism,'' *Proc. Int. Conf. Fifth Generation Computer Systems*, pp. 471-478, 1984.

[16] J. S. Conery, ''The AND/OR process model for parallel interpretation of logic programs,'' Technical Report 204, University of California - Irvine, June 1983.

[17] G. J. Li and B. W. Wah, ''MANIP-2: A multicomputer architecture for evaluating logic programs,'' *Proc. Int. Conf. Parallel Processing*, pp. 123-130, 1985.

[18] B. W. Wah, G. J. Li, and C. F. Yu, ''Multiprocessing of combinatorial search problems,'' *Computer*, vol. 18, pp. 93-108, June 1985.

[19] H. K. Tony Ma, S. Devadas, and A. Sangiovanni-Vincentelli, ''Logic verification algorithms and their parallel implementation,'' *Proc. 24th Design Automation Conf.*, pp. 283-290, 1987.

[20] L. H. Goldstein and E. L. Thigpen, ''SCOAP: Sandia controllability/observability analysis program,'' *Proc. 17th Design Automation Conf.*, 1980.

[21] R. G. Bennetts, in *Design of Testable Logic Circuits*. Reading, MA: Addison Wesley Publishing Company, 1984.

[22] S. C. Seth, L. Pan, and V. D. Agrawal, ‘‘PREDICT - Probabilistic estimation of digital circuit testability,’’ *Proc. Int. Symp. Fault Tolerant Computing*, pp. 220-225, June 1985.

[23] A. Ivanov and V. K. Agarwal, ‘‘Testability measures - what do they do for ATPG?,’’ *Proc. Int. Test Conf.*, pp. 129-138, September 1986.

[24] V. K. Janakiram, D. P. Agrawal, and R. Mehrotra, ‘‘Randomized parallel algorithms for Prolog programs and backtracking applications,’’ *Proc. Int. Conf. Parallel Processing*, pp. 278-281, 1987.

[25] F. Brglez and H. Fujiwara, ‘‘Neutral netlist of ten combinational benchmark circuits and a target translator in FORTRAN,’’ *Special session on ATPG and fault simulation, Proc. IEEE Int. Symp. Circuits and Systems*, June 1985.

[26] W. T. Cheng, ‘‘Split circuit model for test generation,’’ *Proc. 25th Design Automation Conf.*, pp. 96-101, 1988.

[27] T. Kirkland and M. Ray Mercer, ‘‘A topological search algorithm for ATPG,’’ *Proc. 24th Design Automation Conf.*, pp. 502-508, 1987.

[28] M. H. Schulz and E. Auth, ‘‘Advanced automatic test pattern generation and redundancy identification techniques,’’ *Proc. Int. Symp. Fault Tolerant Computing*, pp. 30-35, June 1988.

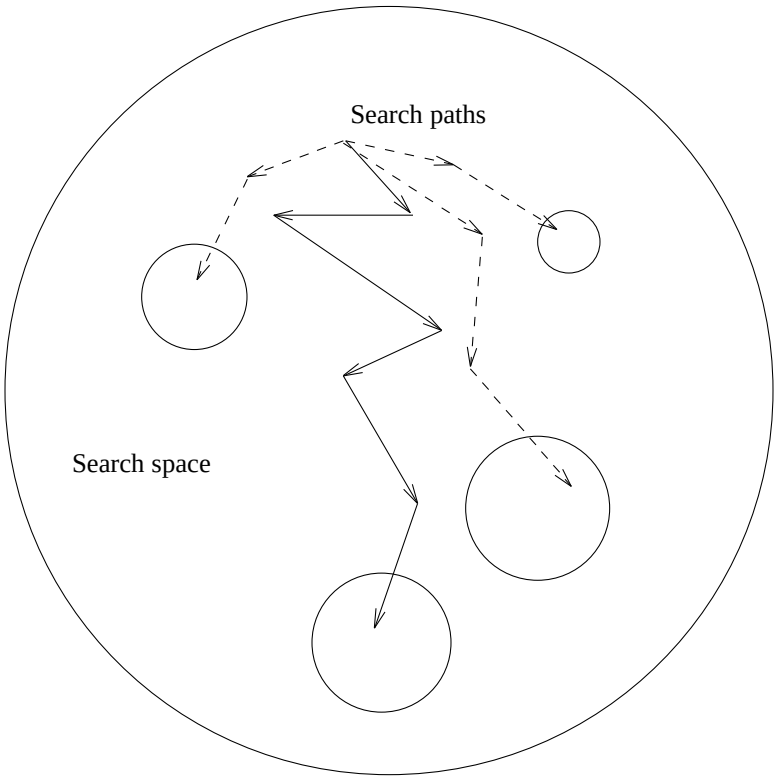
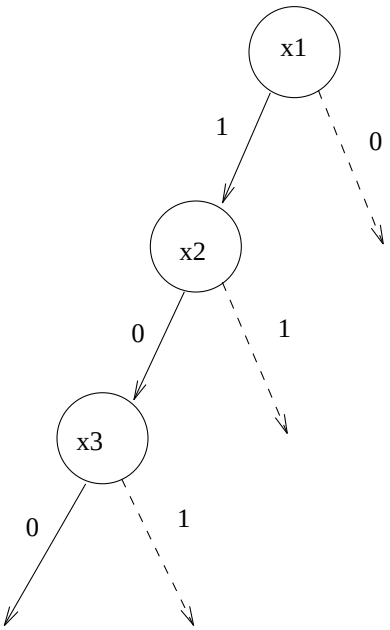


Figure 1. Search space for test generation



val	input	tried
1	x1	0
0	x2	0
0	x3	0

Figure 2. Decision tree in PODEM

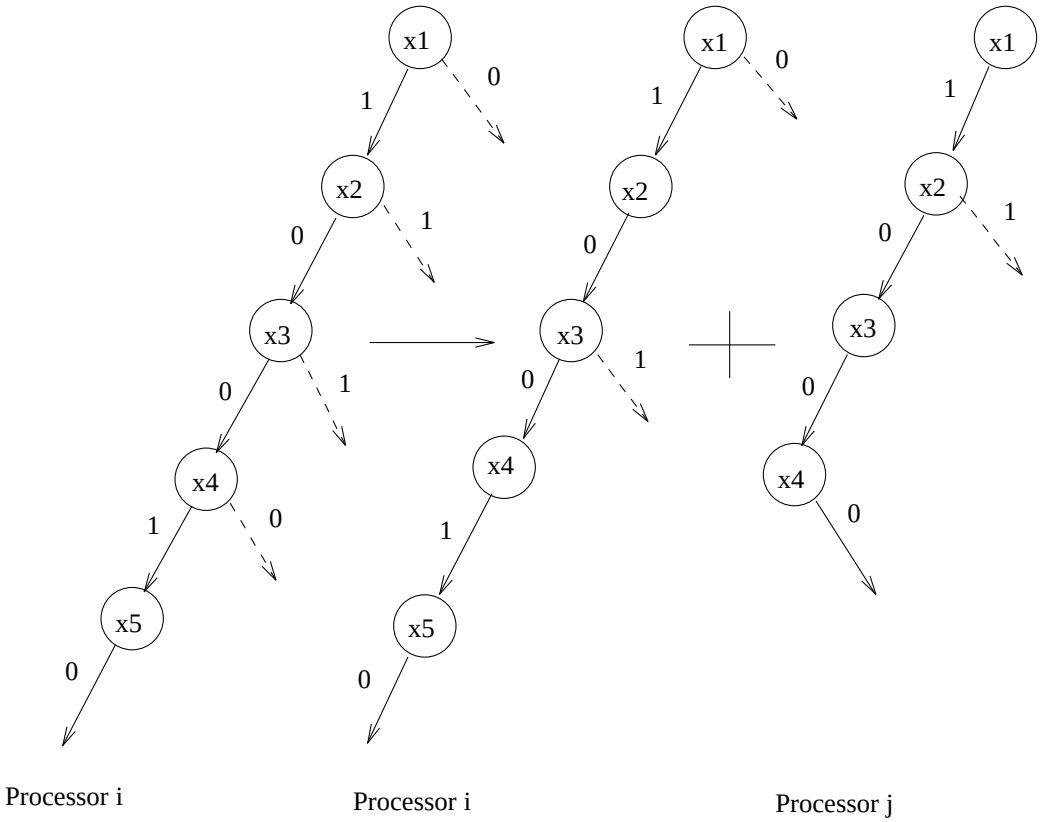


Figure 3. Splitting search spaces

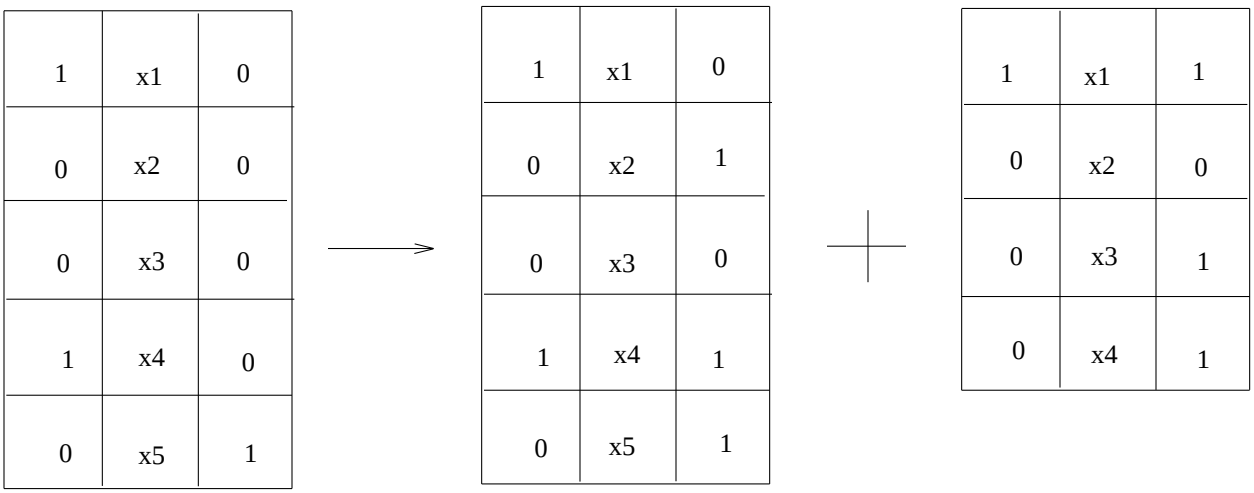


Figure 4. Stacks after splitting

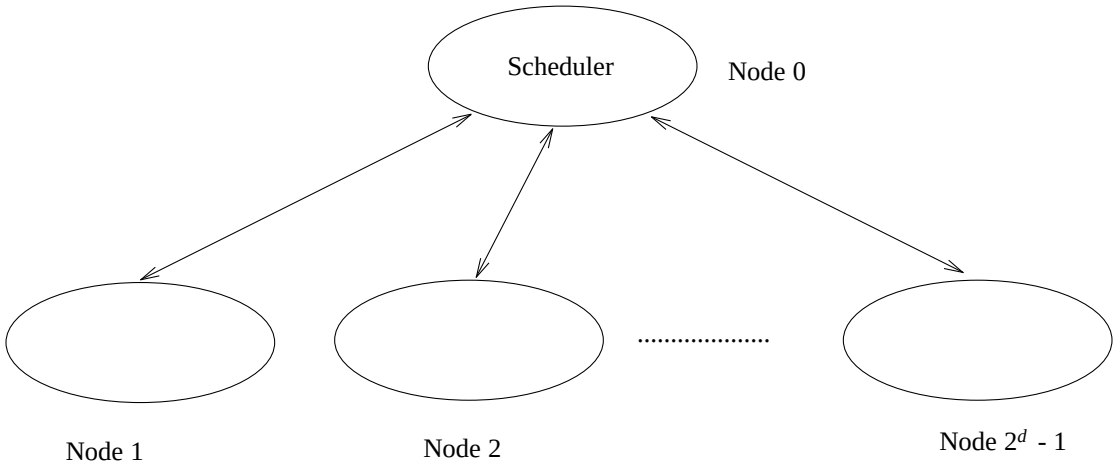


Figure 5. Implementation on the hypercube

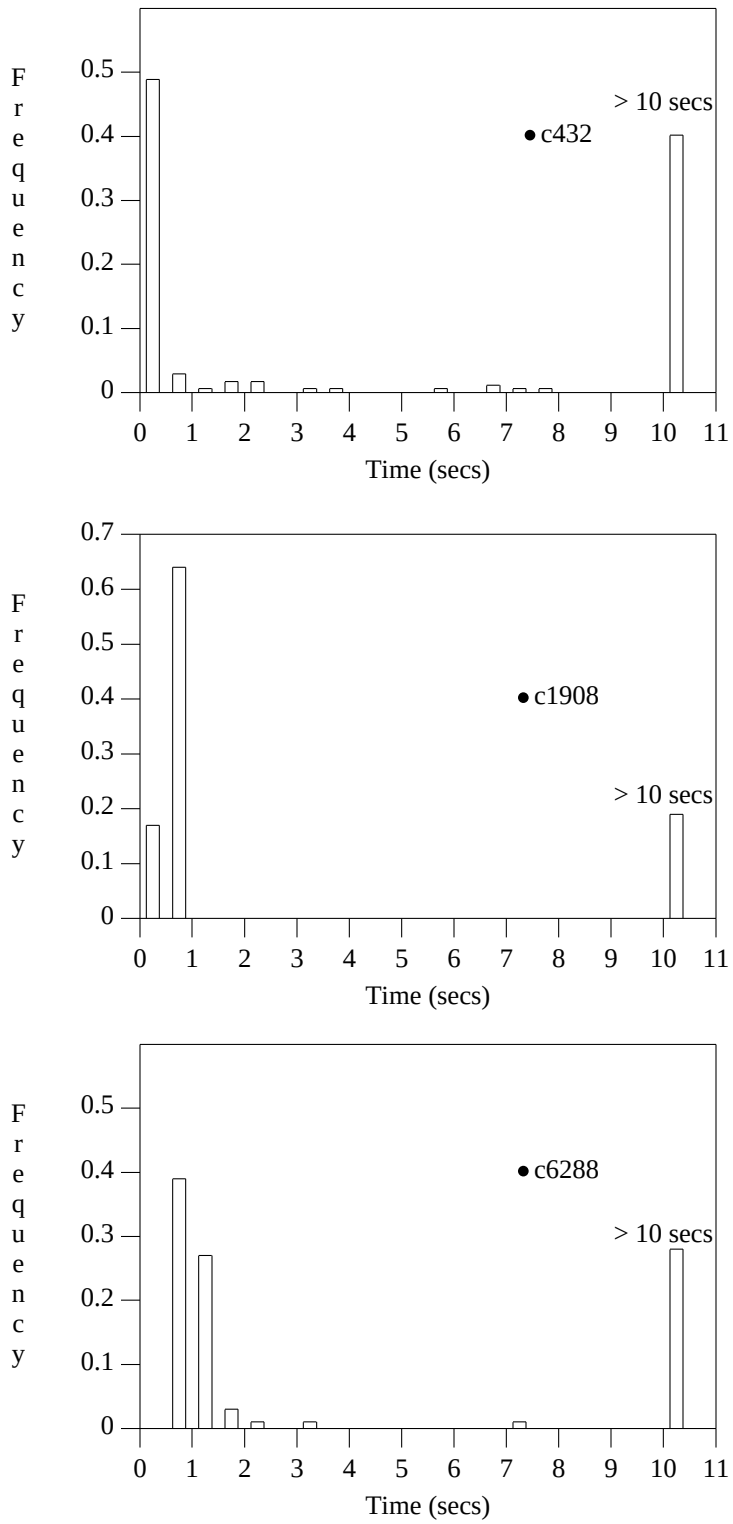


Figure 6. Run times using randomized search

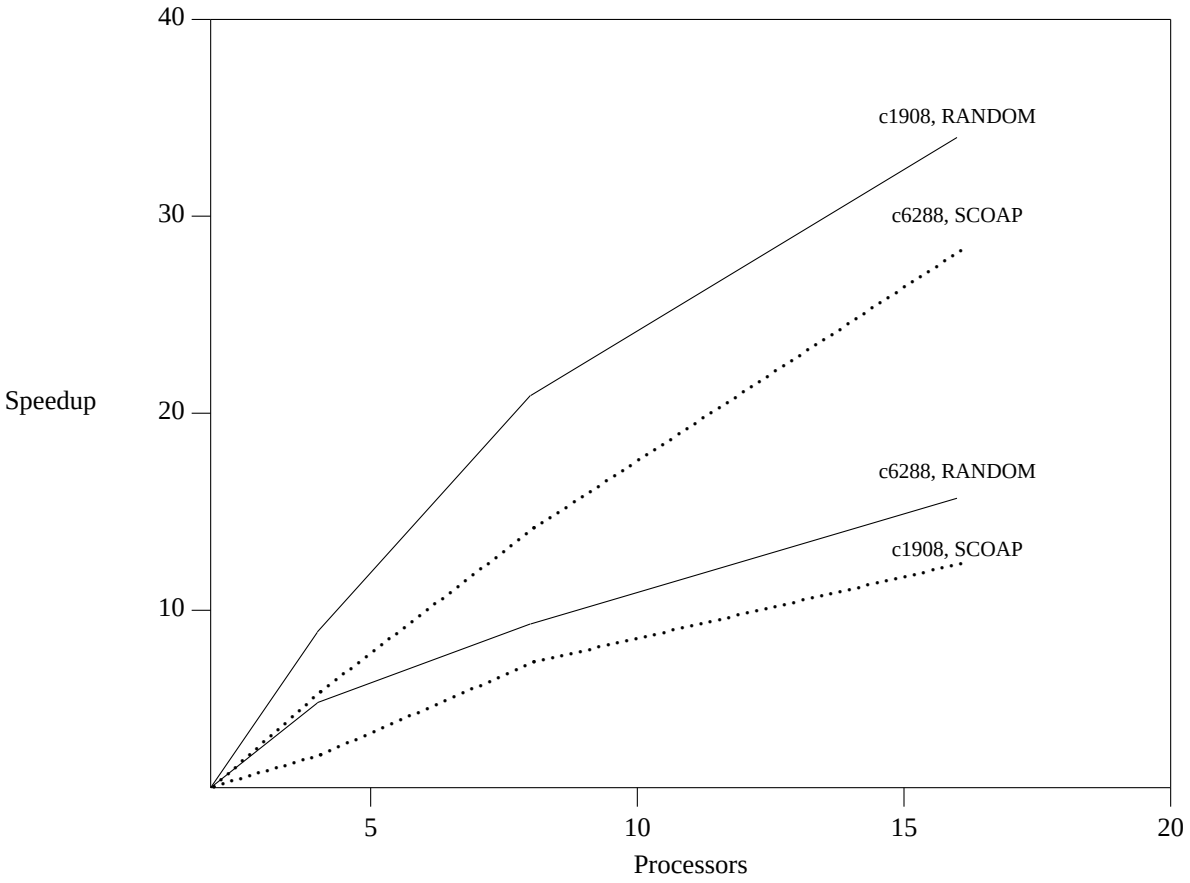


Figure 7. Speedups for HTD faults

Table I. Uniprocessor results for RANDOM heuristic

Circuit	Faults	Tested	Redundant	Dropped	Coverage	Time/Fault (ms)
c432	524	471	0	53	89.89	69
c499	758	725	0	33	95.65	80
c880	942	942	0	0	100.00	43
c1355	1574	1423	0	151	90.41	212
c1908	1879	1789	6	84	95.53	164
c2670	2747	2609	29	109	96.03	101
c3540	3428	3046	39	343	89.99	256
c5315	5350	5266	47	37	99.31	115
c6288	7744	7633	32	79	98.98	401
c7552	7550	7251	52	247	96.73	257

Table II. Uniprocessor results for SCOAP heuristic

Circuit	Faults	Tested	Redundant	Dropped	Coverage	Time/Fault (ms)
c432	524	519	0	5	99.05	40
c499	758	750	0	8	98.95	70
c880	942	942	0	0	100.00	34
c1355	1574	1560	0	14	99.11	150
c1908	1879	1863	6	10	99.47	98
c2670	2747	2518	26	203	92.61	109
c3540	3428	3092	47	289	91.57	290
c5315	5350	5236	44	70	98.69	113
c6288	7744	7688	32	24	99.69	369
c7552	7550	7064	52	434	94.25	258

Table III. Parallel results using RANDOM heuristic

Circuit	HTD Faults	Processors	HTD Coverage		Overall Coverage		Time/Fault (ms)		Speedup
c432	53	4	73.59	(15.09)	97.32	(91.41)	117	(628)	5.4
		8	77.36	(24.53)	97.71	(92.36)	116	(1232)	10.6
		16	79.24	(43.40)	97.90	(94.27)	142	(2208)	15.6
c499	33	4	72.73	(12.12)	98.81	(96.17)	174	(897)	5.2
		8	75.76	(21.21)	98.94	(96.57)	152	(1802)	11.9
		16	75.76	(27.27)	98.94	(96.83)	169	(3357)	19.9
c1355	151	4	91.39	(49.67)	99.17	(95.17)	200	(1318)	6.6
		8	93.37	(54.97)	99.36	(95.68)	188	(2262)	12.0
		16	92.72	(65.56)	99.30	(96.70)	206	(3634)	17.6
c1908	84	4	92.86	(17.86)	99.68	(96.33)	219	(1956)	8.9
		8	95.24	(17.86)	99.79	(96.33)	204	(4262)	20.9
		16	95.24	(22.62)	99.79	(96.54)	254	(8647)	34.0
c2670	109	4	38.53	(30.28)	97.56	(97.23)	453	(1147)	2.5
		8	42.20	(41.28)	97.71	(97.67)	488	(2096)	4.3
		16	45.87	(55.96)	97.85	(98.25)	584	(3493)	6.0
c3540	343	4	48.10	(18.95)	94.81	(91.89)	736	(2239)	3.0
		8	68.51	(34.40)	96.85	(93.44)	625	(4457)	7.1
		16	75.80	(44.61)	97.58	(94.46)	693	(8183)	11.8
c5315	37	4	70.27	(59.46)	99.79	(99.72)	441	(1418)	3.2
		8	81.08	(70.27)	99.87	(99.79)	398	(2339)	5.9
		16	86.49	(78.38)	99.91	(99.85)	467	(3643)	7.8
c6288	79	4	94.94	(29.11)	99.95	(99.28)	490	(2605)	5.3
		8	96.20	(29.11)	99.96	(99.28)	543	(5026)	9.3
		16	96.20	(49.37)	99.96	(99.48)	604	(9507)	15.7
c7552	247	4	61.54	(22.67)	98.74	(97.47)	549	(1621)	3.0
		8	66.80	(26.32)	98.91	(97.56)	619	(3098)	5.0
		16	64.78	(31.17)	98.85	(97.75)	884	(5911)	6.69

Table IV. Parallel results using SCOAP heuristic

Circuit	HTD Faults	Processors	HTD Coverage		Overall Coverage		Time/Fault (ms)		Speedup
c432	5	4	0.00	(0.00)	99.05	(99.05)	229	(582)	2.5
		8	20.00	(0.00)	99.24	(99.05)	227	(1316)	5.8
		16	20.00	(0.00)	99.24	(99.05)	241	(2805)	11.6
c499	8	4	0.00	(0.00)	98.95	(98.95)	283	(722)	2.6
		8	0.00	(0.00)	98.95	(98.95)	295	(1611)	5.5
		16	0.00	(0.00)	98.95	(98.95)	374	(3384)	9.0
c1355	14	4	14.29	(0.00)	99.24	(99.11)	408	(1007)	2.5
		8	14.29	(14.29)	99.24	(99.24)	403	(2075)	5.1
		16	28.57	(14.29)	99.36	(99.24)	556	(4159)	7.5
c1908	10	4	10.00	(0.00)	99.52	(99.47)	634	(1634)	2.6
		8	40.00	(0.00)	99.68	(99.47)	507	(3691)	7.3
		16	40.00	(10.00)	99.68	(99.52)	601	(7395)	12.3
c2670	203	4	23.15	(12.81)	94.32	(93.56)	453	(1241)	2.7
		8	40.89	(25.62)	95.63	(94.50)	448	(2516)	5.6
		16	56.65	(31.53)	96.80	(94.94)	457	(4694)	10.3
c3540	289	4	47.75	(22.84)	95.60	(93.49)	726	(2325)	3.2
		8	66.78	(33.91)	97.20	(94.43)	634	(4544)	7.2
		16	76.82	(38.48)	98.05	(94.81)	640	(8796)	13.7
c5315	70	4	60.00	(58.57)	99.48	(99.46)	440	(1362)	3.1
		8	71.43	(68.57)	99.63	(99.59)	414	(2236)	5.4
		16	80.00	(80.00)	99.74	(99.74)	515	(3544)	6.9
c6288	24	4	91.67	(8.33)	99.97	(99.72)	768	(4454)	5.8
		8	91.67	(12.50)	99.97	(99.73)	723	(10191)	14.1
		16	91.67	(20.83)	99.97	(99.75)	766	(21631)	28.2
c7552	434	4	53.69	(29.49)	97.34	(95.95)	612	(1848)	3.0
		8	73.50	(30.88)	98.48	(96.03)	580	(3455)	6.0
		16	75.12	(35.48)	98.57	(96.29)	862	(6589)	7.6

Table V. Performance for redundant faults of c2670

Processors	Time/Fault (ms)
2	2581
4	963
8	579
16	512

Table VI. Dropped faults on 16 processors

Circuit	RANDOM		SCOAP		SOCRATES
	Redundant	**Dropped**	**Redundant**	**Dropped**	
c432	0	11	0	4	4
c499	0	8	0	8	8
c1355	0	11	0	10	8
c1908	6	4	6	6	9
c2670	59	59	50	88	117
c3540	73	83	73	67	137
c5315	54	5	53	14	59
c6288	32	3	32	2	34
c7552	53	87	52	108	131