PARALLEL ALGORITHMS FOR TEST GENERATION AND FAULT SIMULATION

BY

SRINIVAS PATIL

B.Tech., Indian Institute of Technology, 1985
M.S., University of Iowa, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1991

Urbana, Illinois

CHAPTER 1.

INTRODUCTION

1.1. Parallel Processing for VLSI CAD

With the increased complexity of VLSI circuits, existing Computer-Aided Design (CAD) algorithms will not be able to handle large circuits in a reasonable amount of time. It is possible to propose better heuristics to speed up CAD applications running on a uniprocessor, but the speedup obtained is going to be only marginal compared to the significantly higher speedup possible on multiprocessors. Due to time limitation on the uniprocessor, many CAD algorithms may sacrifice quality to save on time. Due to the tremendous computing power available on multiprocessors, it may be possible to get better quality solutions for the same amount of time as that spent on the uniprocessor. In recent years, parallel processing has gained popularity due to the availability of high-level languages and primitives to specify parallelism, concurrent debugging tools and better user interfaces. Parallel processing hardware has also become more affordable and accessible. Thus, parallel processing is a very attractive and cost-effective alternative for speeding up VLSI CAD algorithms [1-7].

In many cases, modifying a CAD algorithm may be very simple since many CAD problems are inherently parallel. For example, many CAD algorithms are based on a divide-and-conquer approach where smaller subproblems are first solved and then merged together to get the final solution. In such cases, the serial program can be modified to run in parallel by recognizing independent subproblems and by introducing parallel processing primitives so that the subproblems can be solved in parallel. But there can also be instances where during the process of optimization of the serial program, sequential dependencies between different subproblems may be artificially introduced. In such cases analyzing the serial program may expose very little parallelism, if any. Thus, the best serial algorithm may not be the best parallel one. The problem may have to be looked at from a different perspective, and the algorithm may have to be drastically modified to expose the parallelism inherent in the CAD problem. The design of an efficient parallel algorithm to solve a CAD problem can be a very challenging task.

Available parallel processing hardware can be divided into different categories based on message latency between processors (loosely coupled versus tightly coupled), presence or absence of shared-memory (shared-memory versus distributed memory), types of instruction streams (SIMD versus MIMD) and the type of problems that can be mapped on to the machine (general-purpose versus special-purpose). A network of workstations is an example of a loosely coupled MIMD system, and the Connection Machine [8] is an example of a tightly coupled SIMD system. A specialized hardware accelerator for logic simulation, e.g., the Yorktown Simulation Engine [9], is an example of a special-purpose machine. In this dissertation, the main focus will be on general-purpose tightly coupled MIMD systems. Extension of the parallel algorithms to loosely coupled systems in most cases will be very simple as long as the computation to communication ratio is maintained.

The focus of the remainder of this thesis is going to be on parallel algorithms for test generation and fault simulation. This thesis research is a part of a much larger project called the HIPERCAD (*Hi*gh *Per*formance *CAD*) project whose objective is to design and implement efficient parallel algorithms for various CAD problems such as module placement, routing, circuit extraction, logic simulation, fault simulation and test generation. Eventually, it is hoped that the different CAD tools will be able to interface to a common database. A conceptual representation of the HIPERCAD project is shown in Figure 1.1.

1.2. Motivation

1.2.1. Need for parallel test generation

Given a logic circuit, the objective of a test generation algorithm is to derive a set of input vectors which are able to distinguish the fault-free circuit from the faulty one in the presence of a set of faults. An input vector or a vector sequence is called a test for a fault if in the presence of

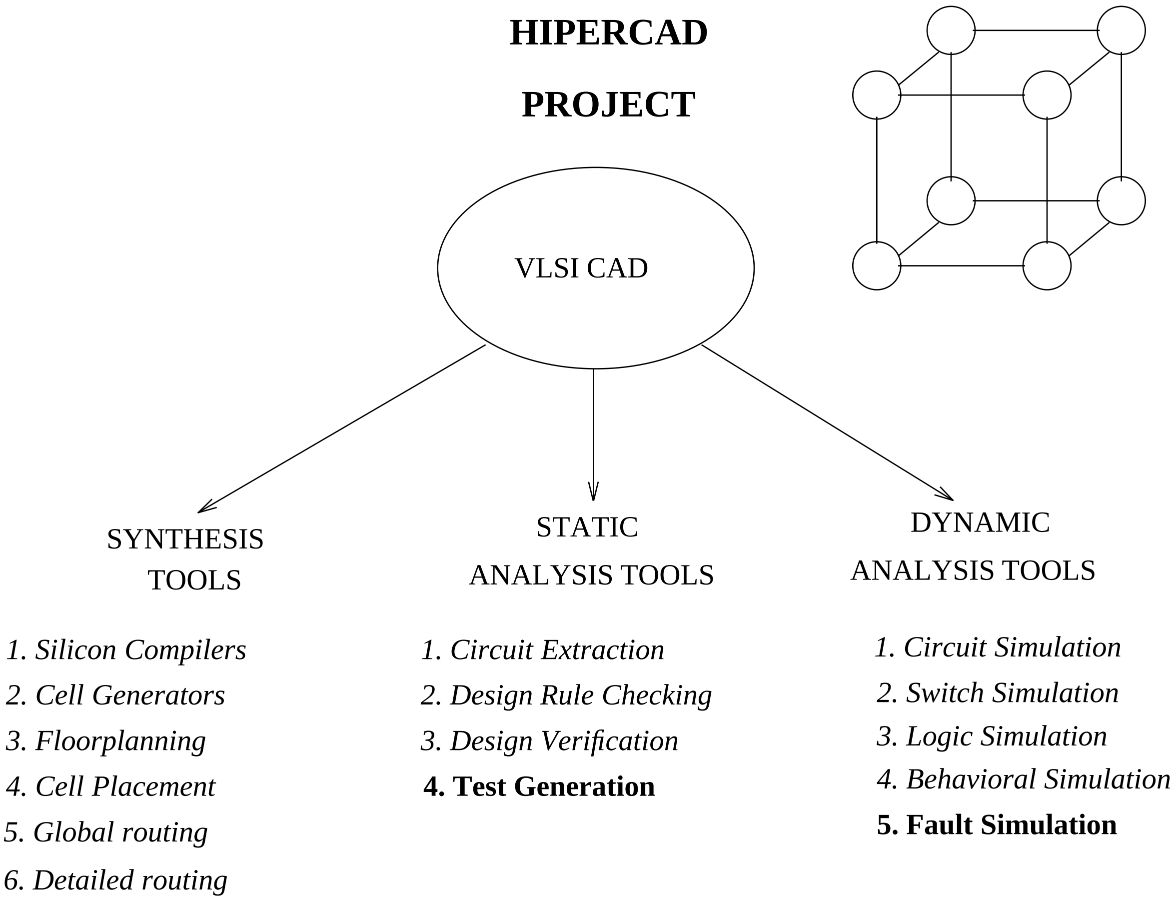


Figure 1.1. HIPERCAD Project

the fault, the faulty circuit produces an output different from the fault-free circuit when the input vector sequence is applied. The fault model normally adopted is the single 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 such as bridging faults, stuck-open faults and delay faults are available but test generation becomes more difficult as the fault model becomes complex. Throughout this thesis, we shall assume the single stuck-at model.

The search space for test generation is exponential in the number of primary inputs of a circuit. The problem of test generation has been shown to be NP-complete [10, 11] even for combinational circuits. 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 [12]. But the 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 [13] 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). The memory requirements of PODEM are also much lower than that for the D-algorithm. A variant of PODEM called the FAN algorithm [14] 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 [15]. Due to the exponential worst-case complexity of test generation algorithms, the test generation time for circuits of VLSI complexity can be prohibitive. Thus, parallel processing can be used to get very high speedups. The parallel test generation techniques we propose are based on PODEM.

1.2.2. Need for parallel fault simulation

The objective of a fault simulation algorithm is to find the fraction of total faults (also referred to as *fault coverage*) which are detected by a given set of input vectors. In its simplest form, a fault is injected into a logic circuit by setting a line or a gate to a faulty value and then the effect of the fault is simulated using logic simulation. Sophisticated methods such as deductive [16] , concurrent [17] and word-parallel [18] fault simulation methods have been proposed which have been able to reduce the time considerably for fault simulation. Most fault simulation algorithms are typically of $O(n^2)$ time complexity where n is the number of lines in the circuit graph. Studies have shown that there is little hope of finding a linear-time fault simulation algorithm [19].

Fault simulation algorithms may be used in two different environments: either in a deterministic test generation environment or in a random pattern test generation environment. In a deterministic test generation environment, the fault simulator helps in finding out the additional faults which are detectable by the same vector sequence. This helps reduce the effort required in test generation and also the test length. The test length determines the test application time and the memory necessary to store the vectors. Figure 1.2 shows the interplay between the test pattern generator and the fault simulator in a deterministic test pattern generation environment. We start with a fault list which is normally constructed by collapsing equivalent faults. The test pattern generator selects a fault from the fault list and tries to generate a test. If the test pattern generator is successful in generating a test, fault simulation is carried out on the entire fault list and the detected faults are deleted from the fault list. The test pattern generator may fail to generate a test if the fault is redundant or some pre-specified resource constraint is exceeded (e.g., the backtrack or time limit per fault). In a random pattern environment, the fault simulator helps in

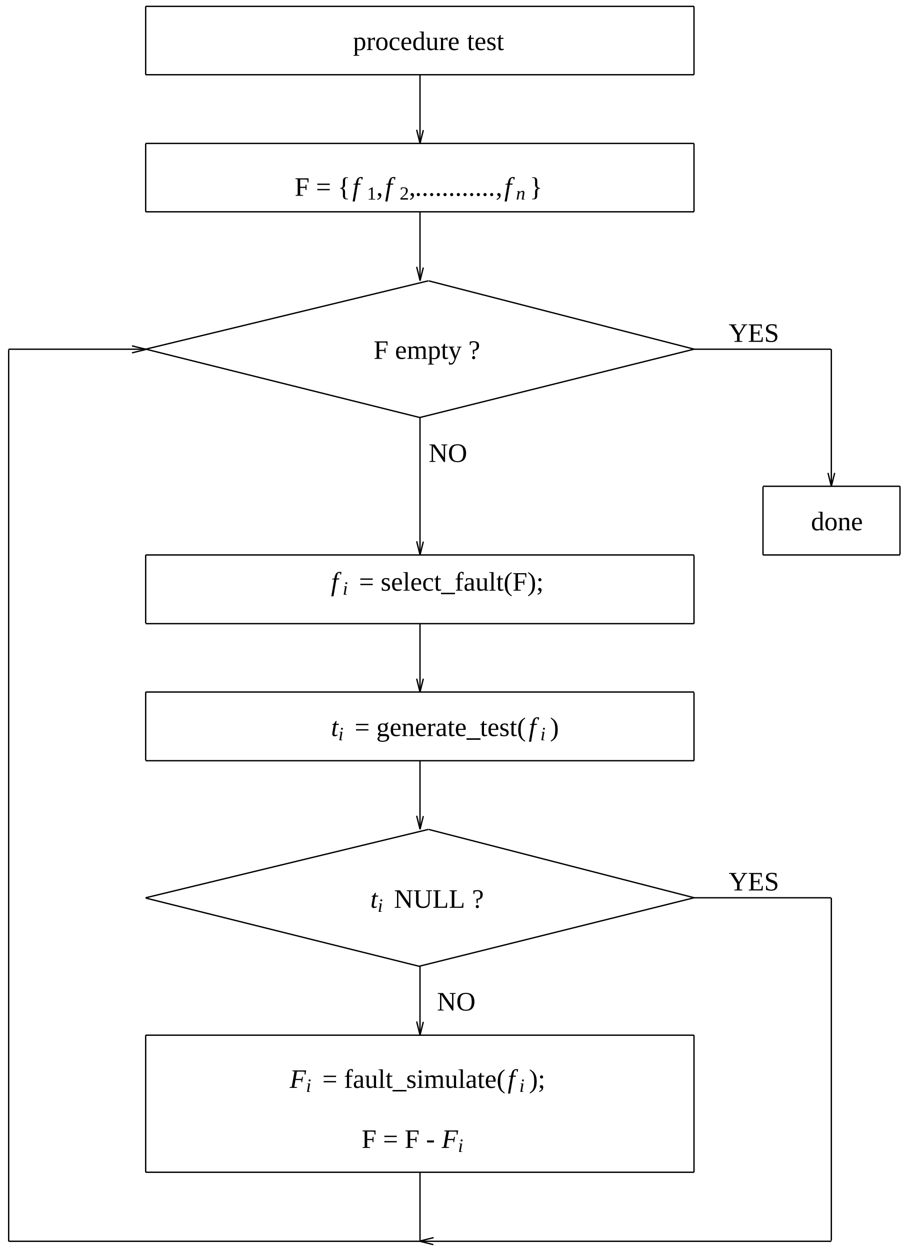


Figure 1.2. Deterministic test pattern generation

evaluating the fault coverage of a set of random patterns. In either of the two environments, fault simulation can consume a significant amount of time, especially random pattern testing where millions of vectors may have to be simulated. Thus, parallel processing can be used to reduce the fault simulation time significantly. The parallel fault simulation techniques proposed in this thesis are based on deductive fault simulation.

1.3. Thesis Overview

This thesis can be divided into two parts. Chapters 2 through 5 deal mainly with parallel test generation for combinational circuits. Though most of the techniques discussed in these chapters may be applicable to sequential circuits also, our main focus will be on combinational circuits. At the end of Chapter 4 we present a parallel test pattern generator for combinational circuits called HIPERTEST. The focus of Chapter 6 is on parallel test pattern generation for sequential circuits.

In Chapter 2 we categorize various parallel processing techniques available for test generation and fault simulation. We discuss the advantages and disadvantages of each of the proposed techniques. We also survey previous work done in the area of parallel test generation and fault simulation.

In Chapter 3 we propose a parallel search method to overcome the deficiencies of inaccurate search heuristics. It will be shown that this method scales very easily with the number of processors and can be implemented easily on configurations with a large number of processors. We show that this method not only results in faster execution of the test generation algorithm but also results in a better quality of the solution.

In Chapter 4 we propose fault partitioning techniques to speed up combinational circuit test generation for faults which are relatively easy to detect. The objective of the fault partitioning techniques is to maximize concurrency without affecting the quality of the overall solution. We propose load balancing techniques which try to minimize the processor idle time with very low communication overhead.

In Chapter 5 we propose performance models for parallel search and fault partitioning heuristics. The performance models take into account the various trade-offs in exploiting parallelism in a test generation/ fault simulation environment.

In Chapter 6 we present a parallel test generation system for sequential circuits. A parallel search technique is used to accelerate test generation for hard-to-detect faults and a circuit partitioned approach is used to accelerate fault simulation. It is shown that the parallel test generation system is comparable in efficiency as well as performance to most existing test pattern generators.

In Chapter 7 we present concluding remarks along with some suggestions for future research.

CHAPTER 2.

EXPLOITING PARALLELISM

2.1. Introduction

This chapter examines various options available to exploit parallelism in test generation and fault simulation algorithms. We shall try to carry out the discussion abstracted from actual implementation of the algorithms so as to be able to examine the parallelism inherent in the problem itself rather than in the particular serial algorithms used to solve the problem. Thus, most of the observations in this chapter should apply independent of the algorithm used for fault simulation or test generation.

Parallelism in an algorithm may be exploited in two different dimensions. If the actual functional structure of an algorithm is partitioned so that different functions can be evaluated in parallel, it is called *functional partitioning*. For example, in an event driven logic simulation algorithm, the event evaluation, scheduling, updates etc. can be done in parallel. However, for most applications, the number of functions which can be evaluated in parallel is limited and thus this method does not scale well with the number of processors. If data elements are partitioned so that they can be processed in parallel, it is called *data partitioning*. Again taking the example of logic simulation, different gates or modules in a circuit can be evaluated in parallel. Data partitioning scales very easily with the number of processors. The parallelization methods proposed are based on data partitioning.

Parallelism in an algorithm can also be characterized by the size of the individual tasks (granularity) which are executed in parallel. Depending on the size of the parallel task, the parallelism may be classified into coarse grain, medium grain or fine grain parallelism. The grain size depends on the relative size of the parallel task versus the communication latency on the multiprocessor. The parallel algorithms presented span the whole spectrum of granularity. For example, the parallel search method presented in Chapter 3 has medium grain granularity while the fault partitioning based methods presented in Chapter 4 are coarse grained. The circuit partitioned approach for fault simulation presented in Chapter 6 is fine grained.

In Section 2.2 we present techniques to exploit parallelism in test generation algorithms. We discuss the advantages and disadvantages of each of the proposed methods. In Section 2.3 we examine the options available to exploit parallelism in fault simulation. In Section 2.4 we review the previous work in the area of parallel test generation and fault simulation for logic circuits.

2.2. Exploiting Parallelism in Test Generation

2.2.1. Fault parallelism

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 are concerned. In an integrated test generation/fault simulation (TG/FS) 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 also include fault simulation. 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 eliminated altogether by dividing the fault sets into independent fault sets [20]. 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.

2.2.2. Heuristic parallelism

As mentioned earlier, almost all test generation algorithms use some heuristic to guide test generation. Experiments reported in [21, 22] suggested that there is no clear cut advantage of using one heuristic over the other. We can exploit *heuristic parallelism* by letting each processor use a different heuristic to guide the search for the same fault. 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, by using different heuristics there is no guarantee that the search spaces are disjoint. This may lead to redundant work. Also, no improvement is possible if a fault remains undetectable for all the heuristics. The main advantage of this method is that communication overheads are low.

2.2.3. Simulation parallelism

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. We can exploit *simulation parallelism* by using parallel logic simulation methods to speed up forward implication. The circuit can be partitioned across processors so that each processor simulates a part of the circuit. The partitioning strategy should minimize communication and maximize concurrency. 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. However, for switch level test generation and simulation based methods, where the activity per gate is much higher, simulation parallelism still holds promise.

2.2.4. AND parallelism

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 [23]. *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.

As an example of how AND parallelism may be used in a test generation environment, consider the objectives of exciting the fault and then propagating the fault to a primary output. Both objectives are mandatory to generate a test for a fault. Thus, we can try to realize these two objectives in parallel by using AND parallel techniques. However, both objectives will depend on some common primary input assignments or signal values in the internal lines of the logic circuit which may lead to variable binding conflicts.

2.2.5. OR parallelism

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. Super-linear speedups might be possible due to search anomalies [24, 25].

2.3. Exploiting Parallelism in Fault Simulation

Exploiting parallelism in fault simulation is relatively simple. At the beginning of fault simulation, we have n_t patterns which have to be simulated against n_f faults. We can exploit *fault parallelism* by partitioning n_f faults across the available processors. We can exploit *pattern parallelism* by partitioning the n_t patterns. It may not always be possible to exploit pattern parallelism. For example, in a test generation/fault simulation environment only one test vector is available at the beginning of each fault simulation step. It is possible to obtain almost linear speedups in the case of fault and pattern parallel implementations. We can exploit *simulation parallelism* by speeding up each pass by using parallel logic simulation techniques [26]. The parallelism available here is strongly dependent on the circuit topology and the partitioning strategy used. The communication overheads can be very high. A *pipeline*-type approach can be used in which all gates at a particular logic level form a stage of a pipeline. Either faults or patterns can then be pipelined through the logic circuit. In the case of sequential circuits, since the test vectors have to be applied in sequence, it is not possible to exploit pattern parallelism. Also, due to the presence of feedback paths through memory elements, the speedups possible through pipelined parallelism may be severely limited.

2.4. Previous Work

2.4.1. Parallel test generation

Though the types of parallelism listed in Sections 2.2 may not be identified as such, instances of different types of parallelism may be found in work done previously. Most of the previous work in the area of parallel test pattern generation and fault simulation has been restricted to combinational circuits. Combinational circuits should be assumed throughout the discussion in this section unless otherwise specified.

A parallel test pattern generator was implemented on the Connection Machine [8] in [27]. The parallel implementation essentially uses exhaustive enumeration of the input search space of a combinational circuit and produces *all possible* test patterns for a particular fault. Due to the exhaustive enumeration method used, the memory requirement used increases exponentially with the number of primary inputs of a logic circuit and is thus not very efficient for circuits with a large number of primary inputs.

The parallel test pattern generation system reported in [28] uses a large multi-microcomputer system where each processor is programmed to perform dedicated tasks, e.g., test generation, fault simulation and scheduling. The parallel test generation algorithm is based on PODEM [13] and the parallel fault simulation algorithm is based on the concurrent fault simulation technique [17]. The implementation uses fault parallelism in the first phase and a form of OR parallelism in the second phase. However, the implementation lacks a systematic way of splitting the search space and the fault partitioning issues are not addressed. Also, inefficient utilization of the available processors occurs due to dedicated tasks being assigned to each processor. In [29] a theoretical analysis of the optimal granularity for maximum speedup in the parallel test generation system reported in [28] was provided.

A parallel logic verification technique utilizing parallel search similar to the implicit enumeration technique used by PODEM was reported in [30]. The logic verification technique relies on comparing two machines by enumeration of cubes of one machine using implicit enumeration and simulating the enumerated cubes on the other machine to see whether the two machines are equivalent. Both the cube enumeration phase and simulation phase were carried out in parallel.

In [31] a parallel search technique based on the technique proposed in Chapter 3 has been used for combinational circuit test generation on a variety of multiprocessor configurations. The authors have been able to implement the parallel search method even on a loosely coupled system such as a network of SUN workstations.

An empirical estimate of the speedups available using fault and heuristic parallelism using actual measured parameters in a networked workstation environment was reported in [32]. Uniprocessor results and latencies in a network of workstations were used to estimate the speedups.

2.4.2. Parallel fault simulation

A fault parallel approach was used to exploit parallelism in fault simulation in [33]. A distributed version of a hierarchical fault simulator using fault partitioning was implemented in [34] on a network of workstations. The pipelining approach has been implemented in [35] on the hypercube and in [36] on the connection machine. Fault simulation using the simulation parallelism approach for both combinational and sequential circuits has been reported in [37].

2.5. Summary

In this chapter, we discussed various techniques to exploit parallelism in test generation and fault simulation algorithms. Fault, heuristic, simulation and AND/OR types of parallelism can be used in the case of test generation. Fault, pattern, simulation and pipelined parallelism can be used for fault simulation. We also discussed the advantages/disadvantages of each of the methods. Finally, we discussed the previous and on-going research in the area of parallel test generation and fault simulation.

CHAPTER 3.

PARALLEL SEARCH

3.1. Introduction

Parallel search has been a very active area of research in the area of Artificial Intelligence [38, 39] and has been applied to a variety of problems including 0-1 Integer Linear Programming (0-1 ILP) [40] and the 15-puzzle problem [41, 42]. The parallel search techniques can be bundled under OR parallel techniques where search for several feasible solutions can be performed concurrently (see Chapter 2). Due to the deficiencies of the search heuristics used for test generation, accelerating test generation through parallel search becomes a very attractive alternative. The focus of this chapter is a parallel branch and bound technique, which is not only able to accelerate test generation for logic circuits, but is also able to obtain a better quality of solution in a time less than or comparable to that taken by the uniprocessor. Superlinear speedups (efficiency greater than 1) might be possible due to search anomalies [24, 25].

Parallel search and enumeration techniques have been applied to a variety of CAD problems in the past, including test generation. A parallel logic verification algorithm using the implicit enumeration technique of PODEM was proposed in [30]. 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 chapter also uses a similar search space allocation strategy but the difference lies in the basic structure of the algorithm. The parallel algorithm proposed in [30] 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 to those portions of the original search space which maximize the chance of finding a solution.

A parallel search method for test generation to search different portions of the decision tree in parallel was proposed in [28] 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 a 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 [28], 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.

An approach based on the parallel branch and bound technique presented in this chapter was also implemented and evaluated in [31]. However, the scheduling and task allocation algorithms presented in [31] were markedly different from those presented in this thesis. The authors were also able to implement and get speedups on a variety of machines, including a mesh-connected multiprocessor and a network of SUN workstations.

In this chapter we present a parallel branch and bound formulation of test generation. In Section 3.2 we motivate the need for test generation using parallel branch and bound. In Section 3.4 we present algorithms for search space allocation. In Section 3.5 we present results based on implementation of PODEM using the parallel branch and bound technique. All algorithms have been implemented on a 16 processor Intel iPSC/2 hypercube and evaluated using the ISCAS [43] combinational 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 [13].

3.2. 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 3.1. 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 [44, 45] to guide the search. But the problem of calculating exact

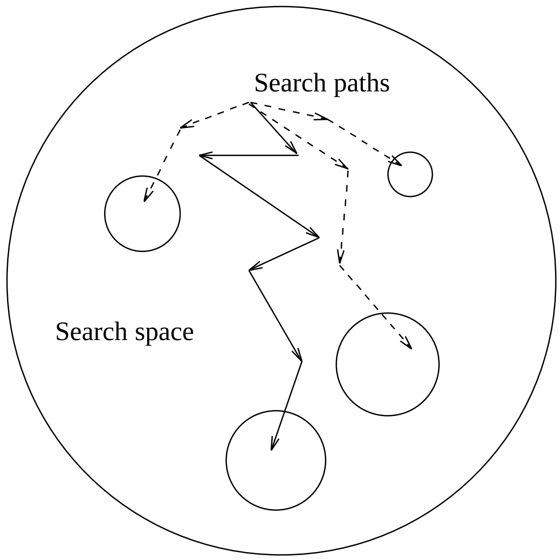
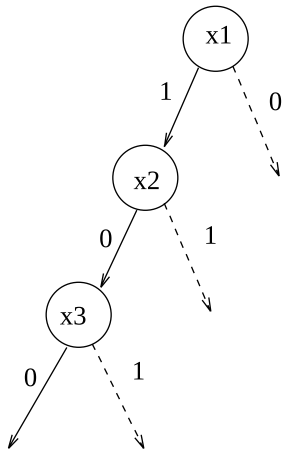


Figure 3.1. Search space

testability measures for a logic circuit with reconvergent fanout has an exponential worst-case time complexity [46]. 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 a fewer number of backtracks [47]. Since it is not practical to calculate exact testability measures and it is expensive to recompute testability measures, 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 [21, 22, 32], 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 in Chapter 2, 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 3.2. 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 three fields - a *val* field storing the current assignment to a primary input, an *input* field identify-



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

Figure 3.2. Decision tree in PODEM

ing 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 [21] 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. Therefore, the algorithm would perform much better if one were to backtrack to the point which gave rise to the conflict. However, 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 that for the uniprocessor 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 3.3 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 an equal number of choice points. The exact mechanism by which these search spaces are allocated is described in the next section. A 0 in

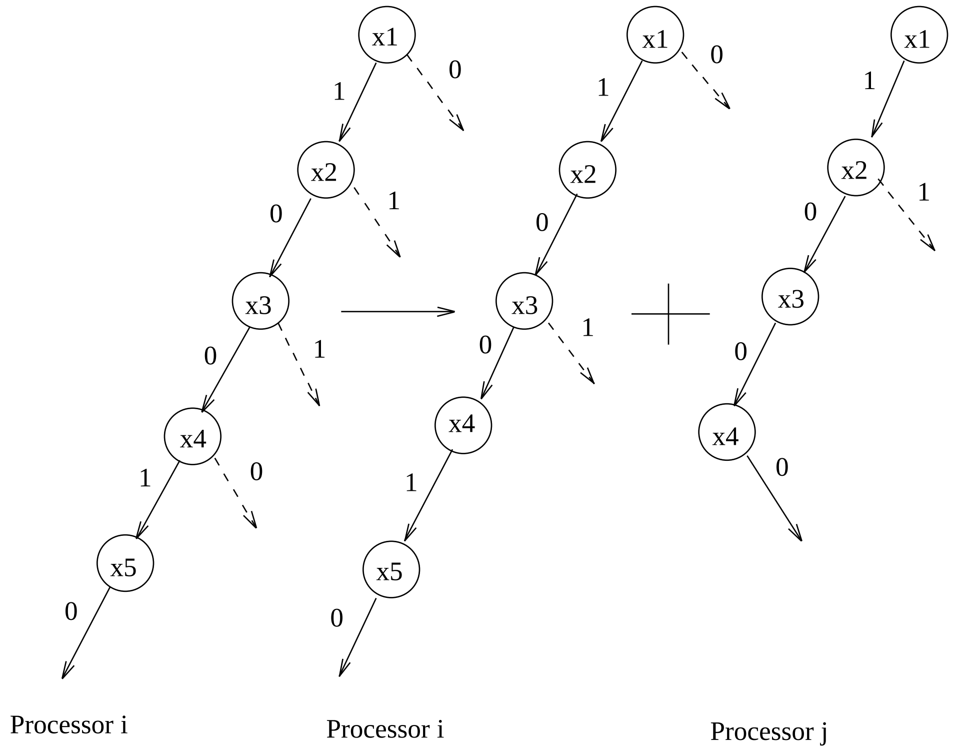


Figure 3.3. Splitting search spaces

the *tried* field of a stack entry for a processor indicates that the processor has been allocated the particular choice point. Figure 3.4 shows the stacks after splitting.

3.3. Algorithms for Search Space Allocation

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 3.5. 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 are broadcast by the host to all the nodes. For each fault, the host broadcasts the

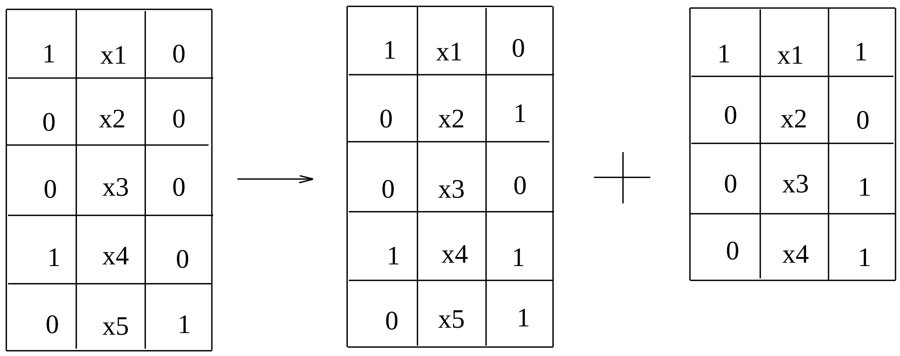


Figure 3.4. Stacks after splitting

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:

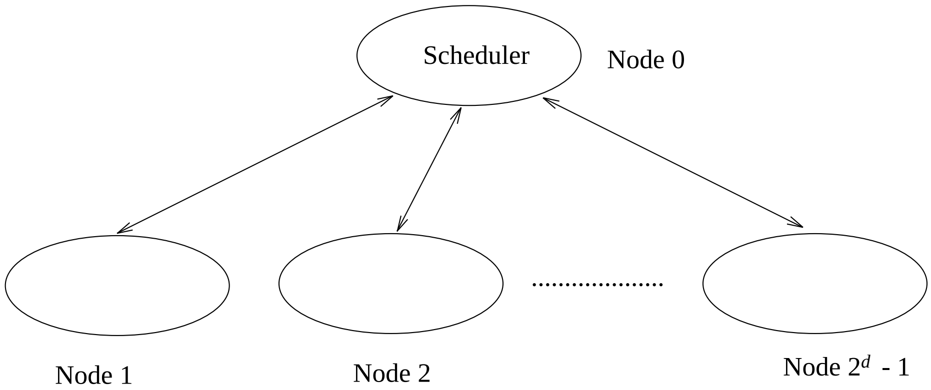


Figure 3.5. Implementation on the hypercube

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 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 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 that 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 3.4 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 when compared to the communication time. Due to search anomalies, superlinear speedups are possible as borne out by results presented in the following sections.

3.4. 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 [44] heuristic were used. It has been shown in [21, 22] that SCOAP performs better than other heuristics on the average.

The results of the uniprocessor implementation on the ISCAS benchmark circuits [43] are shown in Tables 3.1 and 3.2. Table 3.1 shows the results for the RANDOM heuristic and Table 3.2 shows the results for the 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

Table 3.1. Uniprocessor results using 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 3.2. Uniprocessor results using 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

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 3.1 and 3.2, the uniprocessor algorithm is as efficient as the implementations reported in [22, 48] 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 3.3 and 3.4 show results for HTD faults using parallel branch and bound on 4, 8 and 16 processors. Table 3.3 shows the results for the RANDOM heuristic and Table 3.4 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 3.1 and 3.2. 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

Table 3.3. 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 3.4. 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

uniprocessor algorithm for the same total backtrack limit and (3) the parallel algorithm runs much faster than the uniprocessor algorithm. The numbers in parentheses in Tables 3.3 and 3.4 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 3.6. 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 an increase in the number of processors can cause a small decrease in the fault coverage (e.g., results for c1355 in Table 3.3 for 8 and 16 processors). Only in one case does the parallel algorithm have a fault coverage figure less than that for the uniprocessor algorithm (c2670 in Table 3.3 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 3.1 or 3.2. 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, 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 3.5 shows the performance of the parallel and serial algorithm for 24 of the

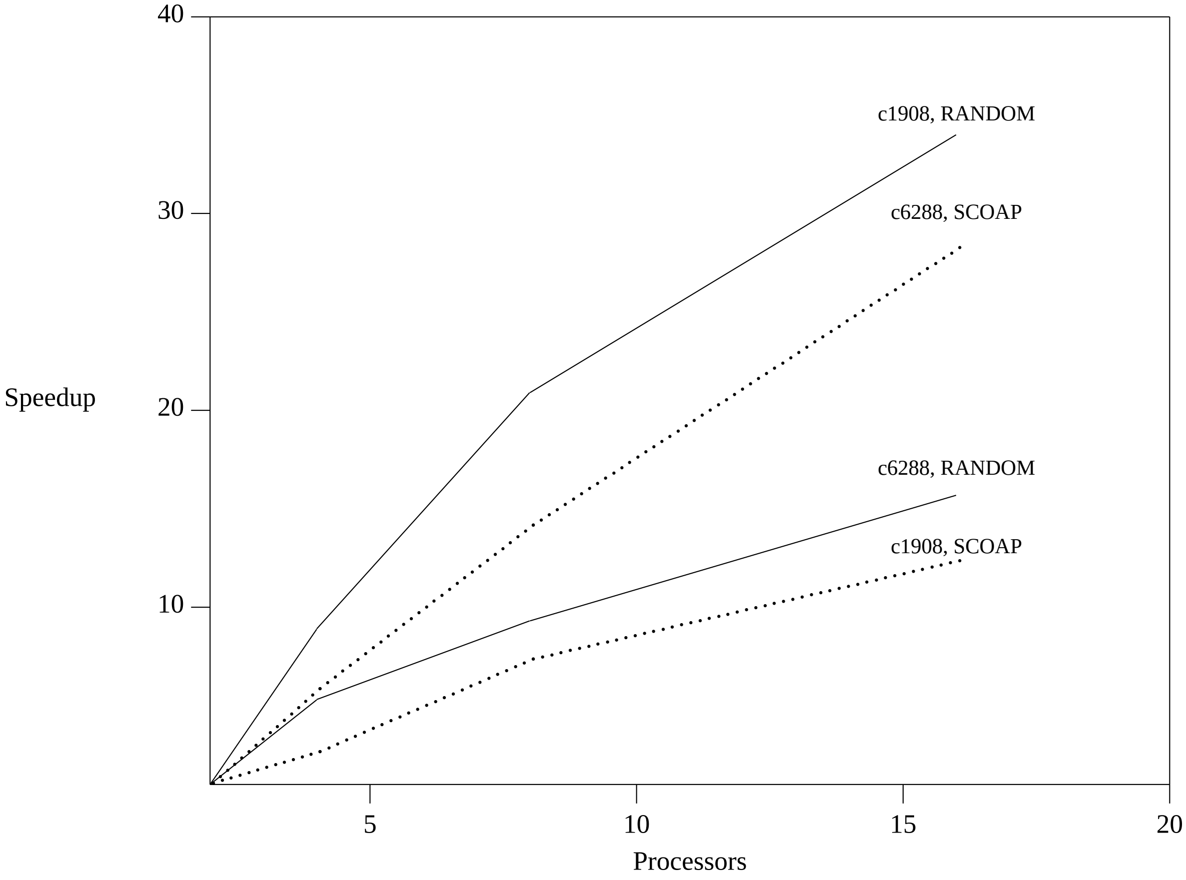


Figure 3.6. Speedups for HTD faults

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 [49, 50] 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 3.6. For comparison, the redundant faults detected by special redundancy identification techniques reported in [50] for the ISCAS circuits are also shown (under the column labeled

SOCRATES). It can be seen that many of the dropped faults in Table 3.6 have in fact been found to be redundant in [50]. 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.

Table 3.5. Performance for redundant faults of c2670

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

Table 3.6. 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

The splitting technique of Figure 3.4 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.

3.5. Summary

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 method 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 a larger number of processors, the scheduler could become a bottleneck in the performance of the parallel algorithm. A distributed scheduling technique would be desirable for large processor configurations. The parallel search techniques presented in this chapter are used to implement a very efficient parallel test generation system for combinational circuits discussed (Chapter 4) and for synchronous sequential circuits (Chapter 6). We analyze the performance of parallel search in Chapter 5.

CHAPTER 4.

FAULT PARTITIONING ISSUES

4.1. Introduction

In the previous chapter we presented methods to accelerate test generation using parallel search. However, the parallel search techniques can be applied to only those faults which are hard-to-detect (HTD). A majority of the faults can be detected within very few backtracks (called *easy-to-detect*). As shown in Tables 3.1 and 3.2, at most 10 percent of the faults are hard-to-detect while a majority of the faults (greater than 90 percent) are easy-to-detect. In this chapter we study how to accelerate test generation and fault simulation for easy-to-detect faults. Fault parallelism is used to speed up test generation for easy-to-detect faults We examine the performance trade-offs in the implementation of a parallel test generation/ fault simulation (TG/FS) system for combinational circuits based on fault parallelism.

In Section 4.2 we motivate the need for good fault partitioning techniques to exploit fault parallelism. We show that the overall test length and speedup are strongly dependent on the fault partitioning method used. In Section 4.3 we present static fault partitioning methods used in the preprocessing phase. In Section 4.4 we propose a dynamic work load distribution strategy based on a distributed work load allocation and termination detection scheme. In Section 4.5 we present experimental results based on an implementation on the Intel iPSC/2 hypercube using the ISCAS combinational benchmark circuits [43]. In Section 4.6 we present HIPERTEST, a parallel test pattern generation system for combinational circuits based on the fault partitioning techniques presented in this chapter and the parallel search techniques presented in the previous chapter.

4.2. Exploiting Fault Parallelism

If fault simulation is not a part of a test generation system, i.e., only a deterministic test pattern generation program is used, the problem of partitioning faults for test generation is relatively straightforward. To obtain the maximum amount of speedup on a multiprocessor, the work load must be balanced on all processors at all times. A simple dynamic allocation scheme in which an idle processor requests a target fault from some other processor/host would suffice. It is then very easy to predict linear speedups as reported in [32]. But most test generation systems use a test generation/ fault simulation loop. Each test generation step is followed by a fault simulation step to detect additional faults from the fault list. The purpose of this step is two-fold. First, since the complexity of fault simulation is much less than that of test generation, the fault simulation step saves the effort in test generation for faults having the same test vector. Second, the overall test length is reduced due to the additional faults detected during the fault simulation phase (also referred to as *dynamic test compaction*). The overall test generation time for the fault set is much less if fault simulation is used.

Due to the use of fault simulation, the problem of fault partitioning in a parallel TG/FS environment becomes non-trivial. A simple dynamic scheme in which each processor is allocated one target fault at a time reduces to the case where no fault simulation is used at all. Thus for fault simulation to be useful, each processor has to be allocated more than one fault at a time. The best allocation strategy should minimize the overall run time and test length. An ideal allocation would be such that (a) if a test vector for fault f_i can also be a test for f_j then they are assigned to the same processor and (b) the load on all processors is balanced. If fault f_i is assigned to processor p_i and fault f_j is assigned to processor p_j and both have a common test vector, then p_i and p_j might end up doing some redundant work which could otherwise have been avoided on the uniprocessor by using fault simulation. Two faults are called *compatible* if a vector exists which can detect both the faults. It is easy to see that a compatibility relationship is much weaker than a fault dominance or equivalence relationship. If fault f_i is dominant over f_j or f_i is equivalent to f_j, then f_i and f_j are compatible but not vice versa.

As an example, consider the six faults shown in Figure 4.1. An edge between f_i and f_j means that fault f_i is compatible with fault f_j. It should however be noted that compatibility is not a transitive relation, i.e., if f_i is compatible with f_j and if f_j is compatible with f_k it does

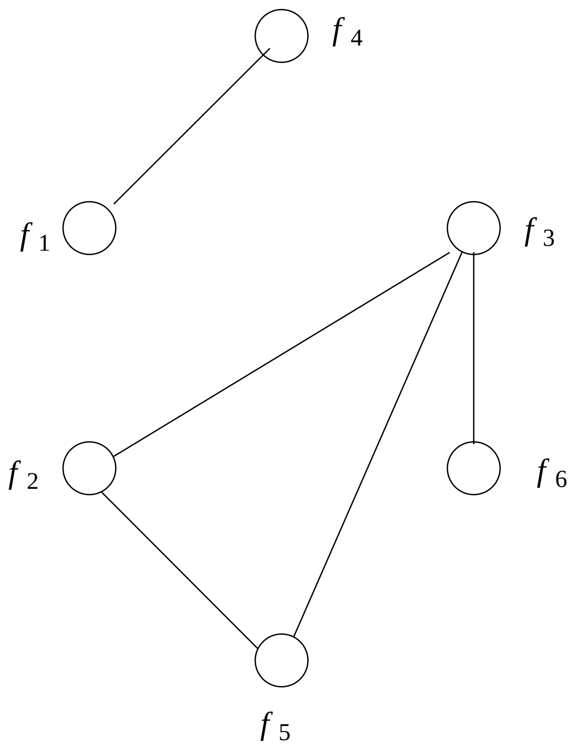


Figure 4.1. Compatible fault sets

not mean that f_i and f_k are compatible. For our example, we shall assume that if fault f_i is compatible with faults f_j and f_k, at least one vector exists which is a test for all the faults $\{f_i, f_j, f_k\}$ which in the general case may not be possible. Also, we shall assume that whenever the test generator generates a test for a fault f_i, it generates a vector which detects not only fault f_i but also all faults which are compatible with f_i. These assumptions are only for illustration purposes and may not hold true in the general case. Assume the faults are processed on the uniprocessor as listed, i.e., $\{f_1, f_2, \ldots, f_6\}$. Now assume it takes 10 time units to generate a test for a fault and after each test generation step it takes R time units for fault simulation where R is the number of remaining faults. Therefore, if the faults are considered in the order f_1 followed by f_2 and so on, it takes 10 time units to generate a test for f_1 and 5 time units to do fault simulation for the remaining faults. Fault f_4 is deleted from the fault list assuming it is detected during fault simulation. It takes 10 time units to generate a test for f_2 and 3 time units for fault simulation and faults f_3 and f_5 are deleted from the fault list. It takes another 10 time units to generate a test for the fault f_6 and the TG/FS process terminates. Thus, the total time for TG/FS on the uniprocessor is $(10 + 5 + 10 + 3 + 10 = 38)$ time units and the test length is 3.

Now assume there are two processors p_1 and p_2. Let faults $\{f_1, f_2, f_6\}$ be assigned to p_1 and faults $\{f_3, f_4, f_5\}$ be assigned to p_2. Using the compatibility relationships shown in Figure 4.1, p_1 takes 30 time units to complete TG/FS for its own fault list and p_2 takes 12 time units for its own fault list. Thus the overall completion time is 30 time units and the test length is 4. Here it has been assumed that the processors are computing test vectors independent of each other and have no knowledge of vectors generated by other processors. This restriction will be relaxed later. Now consider another assignment. Let $\{f_1, f_4, f_6\}$ be assigned to p_1 and $\{f_2, f_3, f_5\}$ be assigned to p_2. The completion time now is 22 time units and the test length is 3, same as that on the uniprocessor. Since $\{f_2, f_3, f_5\}$ belong to the same *compatible fault set*, they were assigned to the same processor thus reducing the overall run time.

We define a compatible fault set as follows:

Definition: A set of faults is called a compatible fault set if for every pair of faults belonging to the same fault set, there exists at least one test vector which detects both the faults. Thus all faults in a compatible fault set are pair-wise compatible.

A compatible set is a complete subgraph in the graph shown in Figure 4.1 (or a clique) [51]. A complete subgraph is one in which there is an edge between any two distinct vertices which belong to the subgraph. Finding out whether two faults f_i and f_j have a test vector in common is a computationally hard problem and is as difficult as test generation itself. In the worst case, it might be necessary to enumerate all possible test vectors for f_i to find out whether there is a vector which detects f_j also. Since our objective is to find out compatible fault sets without actually using test generation and fault simulation, approximate heuristic methods have to be used. If heuristic methods are used, it cannot be guaranteed that two faults belonging to the same partition or set will have a test vector in common. Due to this reason, the partitions generated by any of the heuristic partitioning methods will be referred to as *pseudo-compatible fault sets*. Assigning each pseudo-compatible fault set to a different processor results in a shorter run time and test length due to more faults being detected during the fault simulation phase.

We consider the overall run time to be more important than the test length because the test length in deterministic test pattern generation is orders of magnitude lower than that generated using random patterns. Since the test length affects the memory utilization for storage of test vectors as well as the test application time, because of low test lengths obtained using deterministic test pattern generation, the memory utilization as well as the test application time are going to be very low. Thus, even an order of magnitude increase in the test length is not going to affect the memory utilization and test application time significantly. In the case of random pattern testing however, since the test length is of the order of a few hundred thousand vectors (sometimes more), test length plays a very important part. The run time for test generation, on the other hand, plays a very important part in the overall turn-around time during the design phase when parts of a circuit may have to be redesigned due to low fault coverage or testability to reduce the defect rate in the field.

As the number of processors is increased, since we consider minimization of the overall run time for TG/FS as our primary objective, some pseudo-compatible fault sets may have to be split across processors to keep the load balanced. Thus, with an increase in the number of processors, it can be expected that the test length is also increased. The heuristic partitioning algorithms presented in this chapter try to minimize this increase. As the number of processors approaches the number of faults, it can be expected that the test length approaches the number of faults also (since we assume the primary objective is to reduce the run time).

Apart from using better partitioning methods to reduce the test lengths, test length can also be reduced by broadcasting the vectors generated on one processor (through test generation) to all other processors. These test vectors can be used during the fault simulation process by each processor to reduce the test length as long as the associated overheads are not high. Due to the approximate nature of the fault partitioning techniques used, it is still very likely that a test vector generated for a target fault f_i assigned to processor p_i is also a test for a fault f_j assigned to processor p_j. Thus, if we communicate the test vector to processor p_j then it does not have to generate a test for fault f_j. Since each processor is using test vectors generated on its own and those sent by other processors, the number of test vectors used for fault simulation by each processor may be greater than or equal to the number of vectors used for fault simulation on the uniprocessor. It can so happen that the fault simulation phase dominates the overall run time and each processor ends up spending about the same time as the uniprocessor. Also the communication overhead is increased and the multiprocessor may quickly run out of buffer space for messages because of test vectors arriving from other processors at a rate faster than the fault simulation rate. This can cause further degradation in performance. Thus, we have to find an optimal value of some communication parameter p_c, called the communication cutoff (defined later) factor, which results in a small test length without producing a severe degradation in run time.

Partitioning methods which are used as a preprocessing step before the actual TG/FS phase will be referred to as *static* partitioning methods. A static partitioning method may not result in balanced work loads on all the processors. A *dynamic* load balancing technique may have to be used to keep the processor loads balanced at run time. Based on our discussions above, we come to the following conclusion:

A *good* partitioning method to exploit fault parallelism should do the following:

(1) Avoid potential increase in test length by assigning all faults in the same compatible fault set to the same processor.

(2) Control the degree of communication based on an optimal value of the communication cutoff factor p_c.

(3) Use dynamic load balancing if necessary to keep the load balanced on all processors.

(4) Use a static partitioning technique which requires a very small computation time when compared to the overall TG/FS time.

It should be noted, however, that test length and run time anomalies can occur due to faults being processed in an order different than on the uniprocessor. To reduce the likelihood of such artifacts in our experimental results, we used the following ordering of faults on the multiprocessor:

If faults f_i and f_j are assigned to the same processor, fault f_i occurs earlier in the fault list than f_j if f_i occurred earlier than f_j in the fault list of the uniprocessor.

As an example of how different orderings of faults can produce anomalous results, consider again the six faults shown in Figure 4.1. Consider the following assignment: $p_1 \leftarrow \{f_1, f_4\}$ and $p_2 \leftarrow \{f_3, f_2, f_5, f_6\}$. The order of the faults in each fault list is the order in which they are processed. Using this particular partition, processor p_1 takes 11 time units and p_2 takes 13 time units and the overall completion time is 13 time units. Since it took 38 time units on the uniprocessor, we have a speedup greater than the number of processors. Also, the test length is 2 which is less than that on the uniprocessor. If we follow the ordering given above, the test length would have been 3 and the run time would be 22 time units.

4.3. Static Fault Partitioning Methods

All the methods described below were used in the static partitioning or the preprocessing step to initially assign faults to the processors before the parallel TG/FS process is started. Each of the methods assigns $\lceil n_f / N \rceil$ or $\lfloor n_f / N \rfloor$ faults to each processor where n_f is the total number of faults and N is the number of processors. Since each processor is assigned roughly an equal number of faults, if the number of sets generated by any of the following methods is greater than the number of processors, multiple sets are assigned to each processor. This may sometimes require splitting the sets between two processors.

4.3.1. Random

Faults are assigned randomly to the processors. This method is used for comparison with other approaches.

4.3.2. Input cones

Since each processor has to be assigned approximately n_f/N faults, each processor is allocated a bucket of size n_f/N. A depth-first search is conducted starting from the inputs, where each input is selected in some arbitrary order. This method assumes that all compatible faults lie in the same fanout cone of an input. In the process of traversal from some other input, if it is found that a node (or fault) has already been visited, the depth-first search does not proceed any further for that node since faults in the fanout cone of that node have already been assigned. Thus, faults belonging to more than one topological input cone will be assigned to only one processor. During the traversal of the circuit using depth-first search, the parity of the gates during the search is taken into account. For example, if a s-a-0 fault was selected at the input of an inverting gate, then a s-a-1 fault will be selected at the output. This approach tries to avoid assigning a s-a-0 and a s-a-1 fault on the same line to the same processor (since a s-a-0 and a s-a-1 fault on the same line cannot be detected by the same test vector). As the bucket for a processor becomes filled during the depth-first traversal, another processor is selected for assignment. Thus, input cones may be split across processors during the depth-first traversal.

4.3.3. Output cones

This method is similar to that for input cones except that the depth-first search traversal of the network is performed starting at the outputs and proceeding towards the inputs.

4.3.4. Mandatory constraint propagation (MACP)

A set of constraints is derived based on the testing requirements imposed by each fault. Two faults are considered compatible if their testing constraints match, i.e., the testing requirements do not conflict. A combination of techniques proposed in [14, 20, 49] is used. The method involves the following key steps:

(1) A preprocessing phase in which the flow dominators of all the gates are generated [52]. Gate G is a dominator of a gate g if all paths from g to any output in the logic circuit pass through G.

(2) Starting at the fault site and by using backward and forward implication, a set of uniquely implied logic values is generated. The initial assignments which trigger backward and forward implication are based on both the excitation and propagation requirements of a fault. For example, if we are testing for a stuck-at-0 fault at the output of a gate, then a logic value 1 will be required at the gate output to excite the fault. This procedure also gives a set of faults in the vicinity of a fault α which may be detected by a test for α. All such faults are put in the same *pseudo-compatible fault set* as α (*pseudo-compatible* since two faults in the same set may not really be compatible). This procedure is very similar to the backward and forward implication procedures described in [20]. Also, from information about the flow dominators of the gate under test, additional initial assignments are found as discussed under Step 3.

(3) This particular step tries to generate more constraints from the global flow dominator information. If G is a dominator of g and we are testing for a fault α at the output of g, then it must be observed at G. Also, all inputs of G which are not reachable from the fault site must be set to non-controlling values. A test for a fault at the output of g thus must detect

at least one fault at the output of G (either s-a-0 or s-a-1). Backward and forward implication is performed for G and its inputs. This is repeated for all dominators G of g and additional faults are added to the pseudo-compatible set for α generated in Step 2.

As an example consider the circuit C17 shown in Figure 4.2 which is an example circuit for the ISCAS benchmarks. Assume we are trying to generate a compatible fault set for the fault 11 s-a-1. After forward and backward implication of logic values, the sensitive values on various lines is shown in Figure 4.2. By a line being *sensitive* we mean that a fault associated with that line will be detected. A "+" indicates a line is sensitive and "-" indicates a line is not. The polarity of the fault detected depends on the logic value implied on that line. Thus, a s-a-0 fault will be detected for a line labeled "1+" and a s-a-1 fault will be detected on a line labeled "0+". Either a s-a-1 or s-a-0 fault may be detected on a line labeled "+". Using the sensitivity values shown in Figure 4.2, we have the following pseudo-compatible fault set: {1 s-a-0, 1 s-a-1, 8 s-a-0, 8 s-a-1, 16 s-a-0, 16 s-a-1, 12 s-a-0, 14 s-a-0, 17 s-a-1, 15 s-a-0, 11 s-a-1}. It can be seen that

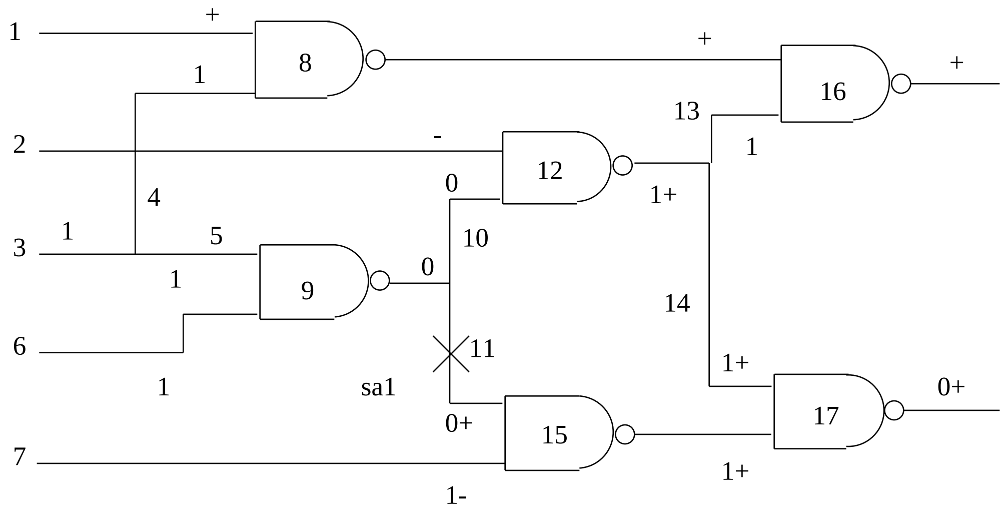


Figure 4.2. Mandatory constraint propagation

s-a-0 and s-a-1 faults on the same line may be included in the same pseudo-compatible fault set even though they are not compatible, hence, the name *pseudo-compatible*.

If faults f_i and f_j are in the same pseudo-compatible fault set, it is very *likely* that they have a common test vector. But the heuristic procedures given above do not guarantee that two faults will be compatible if they are included in the same pseudo-compatible set (by definition). For example, in the example shown in Figure 4.2, the faults 8 s-a-1 and 8 s-a-0 are included in the same pseudo-compatible faults set while clearly they are not compatible. All faults belonging to the same pseudo-compatible fault set are assigned to the same processor. It can be seen that finding pseudo-compatible fault sets by mandatory constraint propagation requires much more computation time than any of the methods discussed above. All partitioning methods presented in this section require $O(n)$ computation time, where n is the number of lines in the logic network, except partitioning by MACP, which takes $O(n^2)$ time which is equivalent to one pass for fault simulation. From the above discussion, it would seem that partitioning by MACP is the best static partitioning method. However, due to its computational complexity, it may take a large amount of preprocessing time for large circuits. This will be validated in the experimental results presented in Section 4.5.

It is easy to see that partitioning by input cones has some inherent advantages over partitioning by output cones if one considers the test generation process itself. During test generation, starting at the fault site, sensitized paths are created which fan out from the fault site towards the output. Thus, additional faults will be detected during fault simulation along these sensitized paths. Since partitioning by input cones will lump all these faults in the same set, it can be expected that partitioning by input cones will do better than partitioning by output cones on the average. We will validate this claim through experimental results presented in Section 4.5.

Also, the partitioning methods presented above may have a tendency to lump all the hard-to-detect or redundant faults into the same set. Thus, a processor may end up spending too much time on such faults resulting in skewed work loads. Since it is difficult to come up with an *a priori* estimate of how difficult it is to generate a test for a particular fault, additional load balancing is done at the run time by using techniques described in the next section. If a processor ends up spending too much time on redundant or hard-to-detect faults, it will be left with a backlog of faults yet to be processed. In such a case, the faults will be reallocated to processors which are idle if the dynamic load balancing technique detailed in the next section is used, thus improving the overall system performance. The dynamic load balancing technique would be effective irrespective of what the backtrack limit is.

4.4. Dynamic Load Balancing

We will show through experiments that the number of faults is a very poor estimate of the work load. It was found that even if an equal number of faults is allocated to different processors, the variation in work load (measured as the amount of time spent doing fault simulation and test generation) was substantial. Since it is very difficult to judge *a priori* how difficult it is to generate a test for a particular fault, a static work load allocation scheme may result in unbalanced work loads. Some processors may become idle before others. However, a dynamic work load distribution scheme may result in excessive communication which may offset any performance gains due to dynamic load balancing.

We propose a method which is a compromise between static and dynamic load balancing and combines the advantages of both. Initially, the fault list is distributed using one of the static partitioning schemes given above. If a processor is done with its own fault list, it broadcasts a request for additional work from other processors. Each of the processors receiving such a message sends the number of faults still to be processed (an estimate of the remaining work load) to the processor requesting work. The processor requesting work then sends an explicit work request message to the processor having the largest fault list. The processor having the largest fault list then splits its own fault list into half and sends it off to the requesting processor. When splitting the fault list, care is taken to ensure that faults are partitioned according to the static partitioning criterion used initially, i.e., for example, if the static fault partitioning criterion was partitioning by input cones, the idle processor gets faults which are in the same input cone.

Termination detection is very simple in this scheme. When all the fault lists have been exhausted, all processors will try to request work from each other. When a particular processor finds that all other processors have exhausted their own fault lists, it terminates. This particular load distribution and distributed termination detection algorithm results in a high message traffic only at the end of the TG/FS process.

The exact algorithmic details of the load balancing algorithm follow. There are two types of messages which are used by an idle processor when requesting work: a QUERY message, which is broadcast by an idle processor to query the other processors for the length of the remaining fault list, and a WRK_REQ message, which is an explicit message requesting work from a busy processor. A parameter τ called the *threshold* is maintained which controls the granularity of the load balancing algorithm and prevents excessive communication at the end of the TG/FS process. Each of the busy processors checks for messages after each TG/FS step by executing the procedure *check_msgs* given below:

procedure *check_msgs*;

(1) If there are no pending messages then return.

(2) If there is a message of the type QUERY, then send the length of the remaining fault list to the requesting processor if the length is greater than the threshold τ, else send zero as the length.

(3) If there is a message of type WRK_REQ, then split the remaining fault list into half and send it to the requesting processor.

end *check_msgs*.

An idle processor p_i enters a waiting phase when it exhausts its own fault list. The actions performed during the waiting phase are given by the procedure *get_work* given below:

procedure *get_work*;

(1) Broadcast a QUERY message to all other processors.

(2) Get the length of the remaining fault list on all other processors. Let p_d be the processor ID of the processor having the largest fault list. If the length of the largest fault list is zero, then exit from the TG/FS process.

(3) Send a WRK_REQ message to processor p_d and wait for work from p_d. If the length of the fault list received from p_d is null, then exit from the TG/FS process else enter the TG/FS loop with the new fault list.

(4) If a QUERY message is received from some other processor, then send zero as the length of the fault list to the requesting processor.

(5) Go to 2.

end *get_work*;

Figure 4.3 shows the dynamic load balancing process. The processor labeled p_{idle} is the idle processor and $p_{\max}$ is the processor having the largest fault list. Q are the QUERY messages

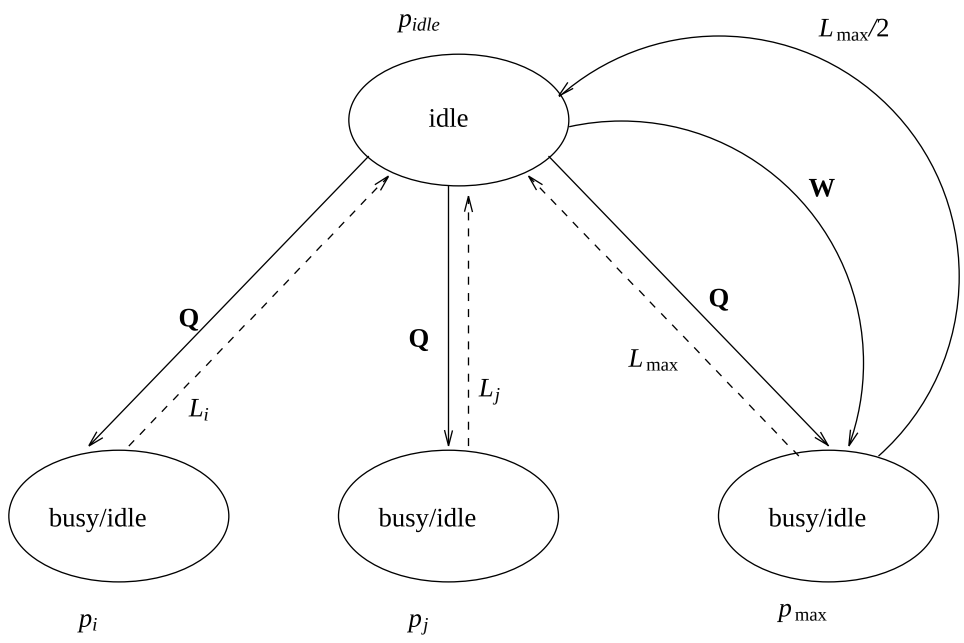


Figure 4.3. Dynamic load balancing

broadcast by p_{idle} and the messages shown with dashed edges are messages which contain the length of the remaining fault list. W is the WRK_REQ message sent to processor p_{max}. Finally, processor p_{max} splits its own fault list into half and sends one portion to p_{idle} (indicated by the message labeled $L_{max}/2$.

In the procedure *get_work*, the idle processor p_i exits the TG/FS loop if the fault list received from the selected donor processor p_d is null. This is done for two basic reasons. First, since the processor with the longest fault list has exhausted its own fault list, it is very likely that the remaining processors have also exhausted their fault lists. Second, this saves the extra communication overhead in termination of the TG/FS loop. Otherwise, each processor will have to send at least $(N-1)$ WRK_REQ messages before terminating. All message passing is done in an asynchronous fashion so that a processor does not explicitly wait for a message at any time as long as it has some other work to do. Thus, a busy processor is able to overlap computation with communication. The efficiency of the load balancing method will be evident when we present experimental results.

4.5. Experimental Results

4.5.1. Implementation

A parallel TG/FS program using PODEM [13] as a deterministic test pattern generator, and a deductive fault simulator [16] was implemented in the C programming language on a 16 processor Intel iPSC/2 hypercube. The SCOAP [44] heuristic was used for guidance. The ISCAS combinational benchmark suite [43] was used to evaluate the TG/FS program. A backtrack limit of 25 was used for the deterministic test pattern generator. The performance of the program on a single node (80386 CPU) of the hypercube is given in Table 4.1. All times are in seconds and include fault simulation and test generation times. All speedup results will be calculated relative to the times shown in Table 4.1. Random fill-in was used during fault simulation for primary inputs with unspecified logic values. It can be seen that the number of test vectors is well below

Table 4.1. Uniprocessor results

Circuit	Faults	Test vectors	Coverage	Time(s)
c432	524	71	99.24	6.76
c499	758	72	98.95	9.23
c880	942	85	100.00	9.05
c1355	1574	145	99.49	38.02
c1908	1879	150	99.73	49.42
c2670	2747	141	95.52	116.81
c3540	3428	206	96.82	235.38
c5315	5350	161	99.20	150.41
c6288	7744	39	99.94	70.81
c7552	7550	251	97.71	437.00

the total number of faults considered. The fault coverage (in percent) figure is the fraction of faults which were detected or were declared redundant.

Table 4.2 shows the uniprocessor and the parallel processor results for some of the ISCAS circuits *when no fault simulation was used*. It can be seen that the uniprocessor results are much inferior to those obtained in Table 4.1 by using fault simulation, both in terms of the test lengths and run times. Also, even though random partitioning was used to partition the faults, the results show almost linear speedups. For the remaining part of our discussion, we shall assume that fault simulation is always used with deterministic test pattern generation.

4.5.2. Static fault partitioning and dynamic load balancing

Table 4.3 shows the time taken by the partitioning methods discussed in Section 3. It can be seen that except for partitioning by MACP, most partitioning methods (which constitutes a pre-processing overhead) take a negligible time compared to the total test generation time. However, for large circuits, partitioning by MACP takes a time comparable to that of test generation. These run times should be kept in mind while comparing the speedup figures presented for vari-

Table 4.2. Parallel results without fault simulation

			Time (seconds)				
Circuit	**Vectors**	**Coverage**	**p = 1**	**p = 2**	**p = 4**	**p = 8**	**p = 16**
c432	519	99.05	14.74	7.46	3.88	2.52	1.53
c499	750	98.95	43.14	22.56	11.28	6.23	3.23
c880	942	100.00	22.12	11.55	5.77	3.13	1.75
c1355	1560	99.11	220.75	115.58	59.36	30.34	16.22
c1908	1863	99.47	164.36	84.06	46.23	23.74	12.32
c6288	7688	99.69	2920.05	1498.71	756.64	386.60	198.14

Table 4.3. Run times for different partitioning methods

Circuit	Time (sec)			
	Random	Input	Output	MACP
c432	0.13	0.06	0.05	2.85
c499	0.05	0.08	0.07	2.24
c880	0.40	0.11	0.11	4.09
c1355	0.12	0.57	0.16	15.37
c1980	0.16	0.24	0.23	18.80
c2670	0.22	0.32	0.96	17.84
c3540	0.35	0.58	0.54	44.30
c5315	0.43	0.70	0.65	39.50
c6288	0.58	0.83	0.97	70.72
c7552	0.60	1.22	0.92	109.19

ous partitioning methods since the speedups do not include the fault partitioning time. Table 4.4 shows the speedups obtained by using different partitioning methods on 16 processors. The test lengths are higher than the corresponding figures shown in Table 4.1 for the uniprocessor in all cases. No communication for test vectors was allowed and only static partitioning was used. The fault coverage figures are sometimes marginally lower than those on the uniprocessor since each processor is doing only local fault simulation, i.e., it is not using test vectors generated by other processors. The actual fault coverage figures may be much higher than just a simple sum of coverage figures obtained by different processors. The multiprocessor fault coverage figures may in fact be equal to or greater than the corresponding numbers on the uniprocessor.

Despite the reason given above, the variation in fault coverages from the uniprocessor and across different partitioning methods is only marginal. Table 4.4 also shows the ranks for each partitioning method for test lengths and speedups. Each partitioning method was given a separate rank for test length and speedup for each of the ISCAS benchmark circuits. Thus a

Table 4.4. Results without dynamic load balancing

Ckts	Random			Input Cones			Output Cones			MACP		
	Tests	Cov	S_{16}	Tests	Cov	S_{16}	Tests	Cov	S_{16}	Tests	Cov	S_{16}
c432	285	99.05	4.0	209	99.05	4.97	249	99.05	4.0	239	99.05	4.66
c499	240	98.95	4.73	195	98.95	3.69	192	98.95	3.86	216	98.95	2.95
c880	290	100.00	5.06	248	100.00	4.66	292	100.00	4.31	251	100.00	3.69
c1355	580	99.11	4.37	362	99.11	5.94	325	99.49	6.48	361	99.49	6.72
c1908	535	99.52	5.79	381	99.63	5.59	453	99.73	4.25	420	99.68	5.90
c2670	622	94.25	6.55	442	93.99	4.24	489	94.36	3.71	448	94.39	3.88
c3540	869	94.81	6.72	608	94.69	4.04	672	95.60	6.68	549	95.71	4.11
c5315	846	98.99	5.02	653	99.08	5.60	711	99.01	3.97	592	99.22	5.55
c6288	323	99.87	3.36	311	99.90	2.50	408	99.86	2.56	337	99.90	2.53
c7552	1061	96.49	6.04	762	96.46	6.01	925	96.64	3.09	790	96.37	4.38
Rank	3.7		1.8	1.5		2.5	2.8		3.0	2.0		2.6

partitioning method which leads to a lower test length gets a higher rank for test length. Similarly, a partitioning method which results in higher speedups is given a higher rank. The ranks were then averaged for all of the ISCAS circuits for each of the partitioning techniques resulting in the composite rank given at the bottom of each column in Table 4.4. For example, consider the row for circuit c432 in Table 4.4. The ranks assigned to each of the partitioning methods with respect to the test lengths are 4, 1, 3, and 2 for random, input cones, output cones and MACP partitioning, respectively. The ranks assigned in each row are not shown because of lack of space. Similarly, the ranks assigned for speedups are 3, 1, 3 and 2 for random, input cones, output cones and MACP, respectively. After assigning ranks in each of the rows, the composite rank is obtained by averaging the rank in each of the columns. For example, the composite rank of 3.7 for the test length for random partitioning (bottom of second column in Table 4.4) was found by averaging all ranks for test length for random partitioning across all the circuits.

In Table 4.4 partitioning by input cones has the highest rank for the test length and random partitioning has the highest rank for the speedup even though it has the lowest rank for test length. These results are in some sense counterintuitive since one would expect that speedup should be higher for the partitioning method which produces a lower test length. This particular behavior is in fact not due to a deficiency in the partitioning technique but due to unbalanced loads on different processors. Consider the load profile shown in Figure 4.4 for the circuit c7552 when output partitioning was used. The load was measured as the sum of fault simulation and test generation times including the time for communication. The dotted line shows the average of the loads on all processors which is 57.77 seconds. If the processor loads were perfectly balanced, all processor loads would have been near the dotted line. The ideal speedup in case of

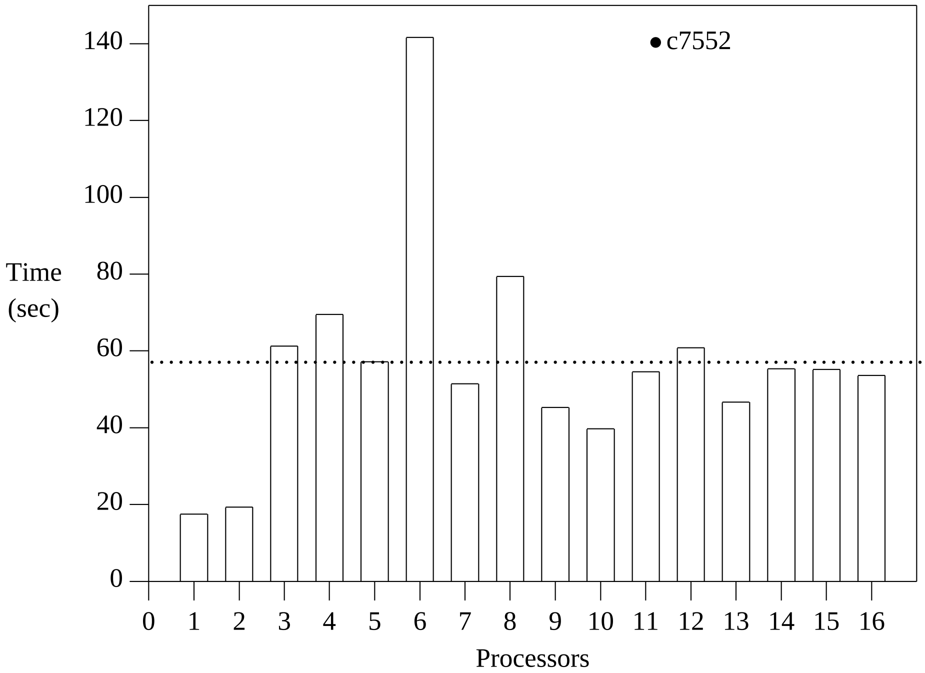


Figure 4.4. Load profile w/o load balancing

balanced loads would have been $\frac{437}{57.77} = 7.56$. It can be seen that even though an equal number of faults were assigned to each processor, the processor loads are widely skewed with the ratio of maximum to minimum load being about 8.

Table 4.5 shows the results obtained when the dynamic load balancing scheme discussed in Section 4 was used. The value of parameter τ was fixed at 10, i.e., the fault list was not split if the length of the remaining fault list was less than or equal to 10. The results show a significant change in speedups obtained for various partitioning techniques. Since we have two separate ranks for each partitioning method, we can measure the "goodness" of each partitioning technique by assigning proper weights which reflect the relative importance of the test length and speedup. If we assign equal weights to test length and speedup, partitioning by input cones and MACP turns out to be the best partitioning method with overall ranks of $(1.5+2.1)/2 = 1.8$ and

Table 4.5. Results with dynamic load balancing

	Random			**Input Cones**			**Output Cones**			**MACP**		
Ckts	Tests	Cov	S_{16}	Tests	Cov	S_{16}	Tests	Cov	S_{16}	Tests	Cov	S_{16}
c432	287	99.05	4.17	213	99.05	4.07	252	99.05	4.86	241	99.05	4.76
c499	242	98.95	5.19	196	98.95	5.27	200	98.95	5.07	224	98.95	4.96
c880	294	100.00	5.11	258	100.00	5.59	310	100.00	4.81	264	100.00	5.48
c1355	581	99.11	4.96	371	99.11	7.33	333	99.49	7.86	370	99.49	7.59
c1908	537	99.52	6.55	409	99.63	7.76	479	99.63	6.93	443	99.68	8.17
c2670	625	94.25	6.79	471	93.92	7.42	543	93.70	6.99	481	94.32	7.50
c3540	870	94.81	6.87	615	94.31	7.49	686	95.07	7.44	565	95.68	8.79
c5315	852	98.99	5.21	673	99.08	6.41	757	99.03	5.60	628	99.22	6.95
c6288	323	99.87	3.57	335	99.90	3.45	431	99.86	2.98	367	99.90	3.74
c7752	1066	96.49	6.51	812	96.41	7.75	957	96.25	6.79	824	96.15	7.31
Rank	3.6		3.4	1.5		2.1	2.9		2.8	2.0		1.7

(2.0+1.7)/2 = 1.85, respectively. However, considering the large amount of preprocessing time required by MACP, which is an order of magnitude higher than the total TG/FS time on a multiprocessor, partitioning by input cones turns out to be the best partitioning method.

Figure 4.5 shows the load profile when the dynamic load balancing technique described in Section 4 was used with output partitioning. Comparing Figures 4.4 and 4.5, the change in the load profile is dramatic. The processor loads are slightly higher than the ideal load of 57.77 seconds due to the communication overheads of dynamic load balancing. The speedup obtained in this case is 6.79 which is very close to the ideal speedup of 7.56. Thus, the dynamic load balancing scheme is able to achieve uniform loads with very little communication overhead. Figures 4.6 and 4.7 show the variation in test length and speedup with the number of processors for

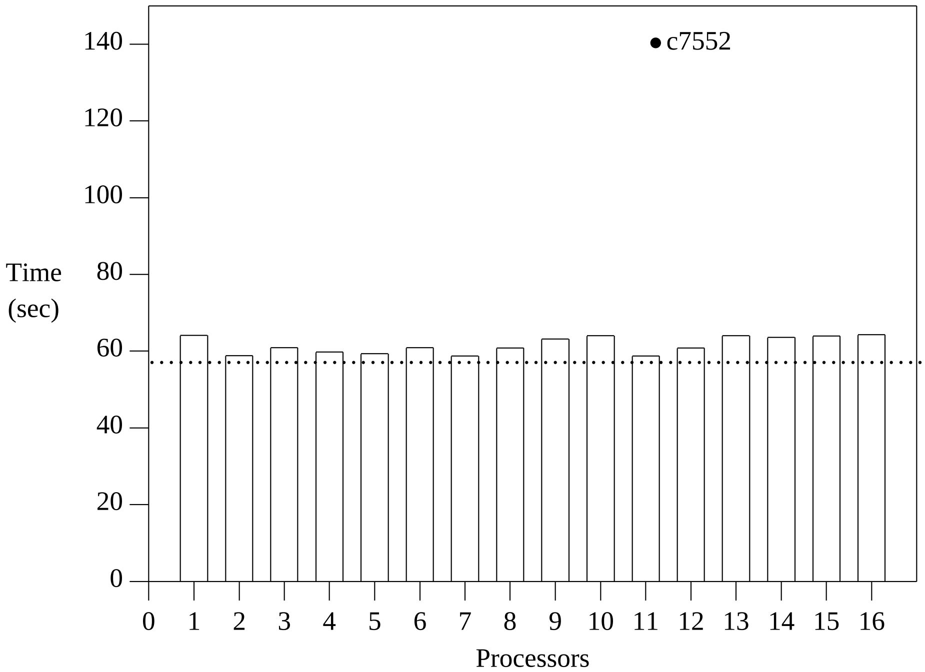


Figure 4.5. Load profile with load balancing

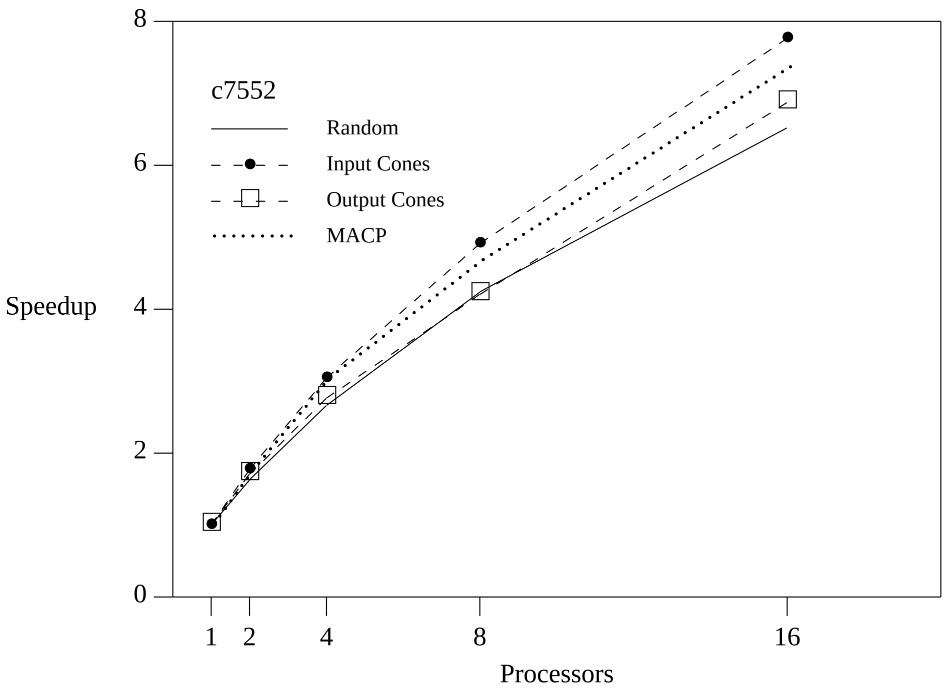


Figure 4.6. Test lengths for c7552

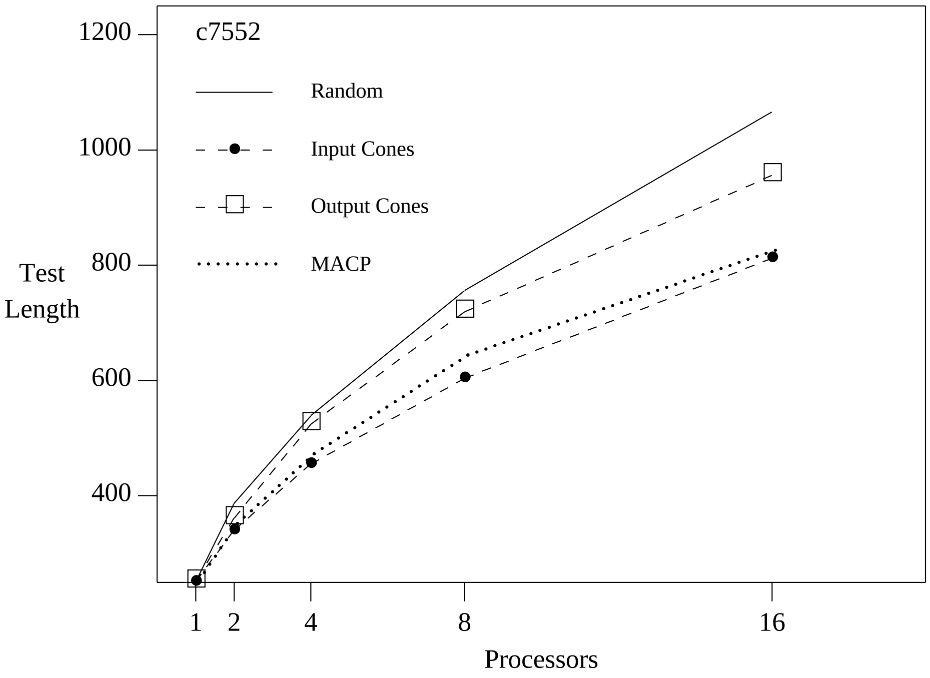


Figure 4.7. Speedups for c7552

c7552 when dynamic load balancing is used. It can be seen that partitioning by input cones does much better than all other methods.

4.5.3. Communication of test vectors among processors

For the purpose of discussion in this section, it is helpful to know the behavior of the TG/FS process. Figure 4.8 shows a plot of the fraction of faults left versus the fraction of total number of test vectors generated for four of the ISCAS benchmark circuits. T_1 is the uniprocessor test length. It can be seen that the curve closely resembles a decaying exponential for each of the circuits. In [53], a combination of a decaying exponential and a straight line was used to approximate the relationship between the number of untested faults and number of test vectors. The reasoning behind this assumption is that in a TG/FS process, initially a large number of faults are detected by the first few vectors. These are usually the faults amenable to random pattern testing. This initial phase (called Phase I in [53]) closely resembles a decaying exponential. After Phase I, the faults remaining are *random pattern resistant* faults. For such faults, test generation has to be done explicitly and very few additional faults are detected with each additional test vector apart from the fault for which the test vector was explicitly generated. The curve in this portion (called Phase II in [53]) closely resembles a straight line. In Figure 4.8 the vertical dashed line segment approximately shows the regions of the curve corresponding to Phases I and II. It can be seen that Phase I ends when approximately 20-30% of the test vectors have been generated and a fault coverage of 60-80% has been reached.

Let each processor broadcast test vectors derived by itself until a fraction p_c of its assigned faults has been detected. Since the decision about when to stop broadcasting test vectors is based on each processor's local fault coverage, no extra communication cost is incurred. The parameter p_c is referred to as the *communication cutoff factor*.

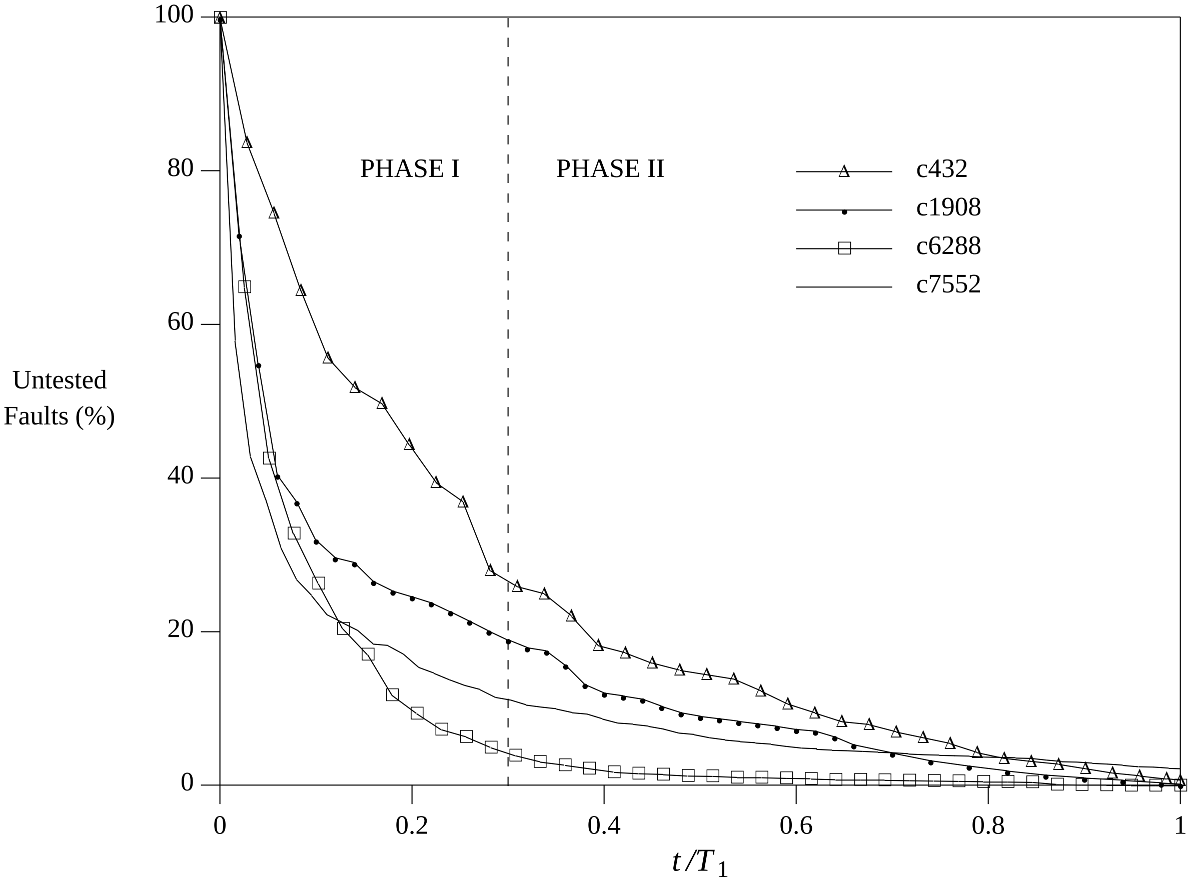


Figure 4.8. TG/FS profiles

As pointed out earlier, most partitioning methods do not guarantee that all faults in the same compatible fault set are assigned to the same processor. Thus, it may be possible to lower the test length by broadcasting the test vectors derived on one processor to all other processors. However, for large values of the communication cutoff factor p_c, the fault simulation time and the communication overhead may become dominant resulting in an a decrease in speedup. However, there may exist an optimal value of p_c, which results in a substantial decrease in test length without any significant increase in run time. Intuitively, since the number of test vectors transmitted varies only as $log(1-p_c)$ during the exponential part of the fault simulation curve

(Figure 4.8), very few test vectors need to be broadcast (about 20-30% of the total vectors for the circuits shown in Figure 4.8) if we confine values of p_c to Phase I of the fault simulation curve. It has been shown in [53] that Phase I ends when the fault coverage is between 60 to 85%. Thus, p_c should lie between 0.6 to 0.85. We shall try to derive a near optimal value of p_c through experiments that follow.

Figure 4.9 shows the variation of test length and run time for circuit c5315 for two different partitioning methods as the communication cutoff factor is varied from 0 to 100% in steps of 10%. For both partitioning methods, it can be seen that the test length is almost a monotonically decreasing function of p_c. However, the behavior of the run time is markedly different. Initially, there is a slight decrease in run time due to the effect of a reduced test length. However, after the value of p_c reaches about 80%, there is a sharp increase in run time indicating that the value of p_c belongs to Phase II of the TG/FS curve. Figure 4.10 shows the variation of test lengths and run times for two different circuits when input partitioning was used. It can be seen that the curves are very similar to the ones obtained in Figure 4.9. Based on experimental results shown in Figures 4.9 and 4.10, it can be seen that a value of p_c between 0.6 and 0.85 produces significantly lower test lengths without a significant increase in run time. From the graphs shown in Figures 4.9 and 4.10, it can be seen that when p_c is equal to 1.0, the test lengths are still higher than the uniprocessor test lengths, because two processors may be generating tests for two faults f_i and f_j simultaneously when in fact they were detected by the same test vector on the uniprocessor. Communicating test vectors in such a case does not help reduce the test length.

Table 4.6 shows the results for all the ISCAS circuits when a communication cutoff factor of 0.6 was used. It can be seen that test lengths are significantly smaller than those in Table 4.5 with only a marginal decrease in speedup. In some cases, there is a slight increase in speedup

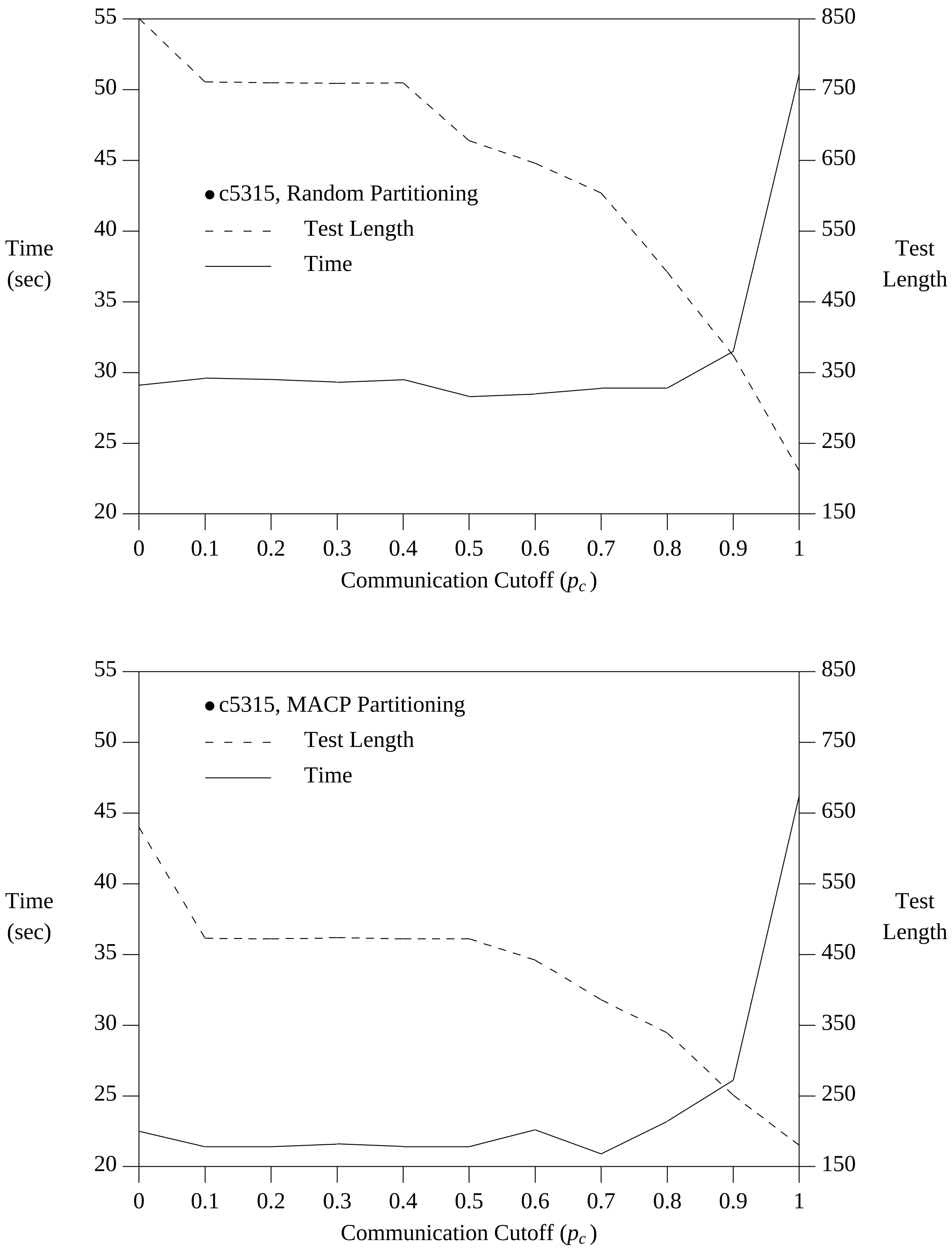
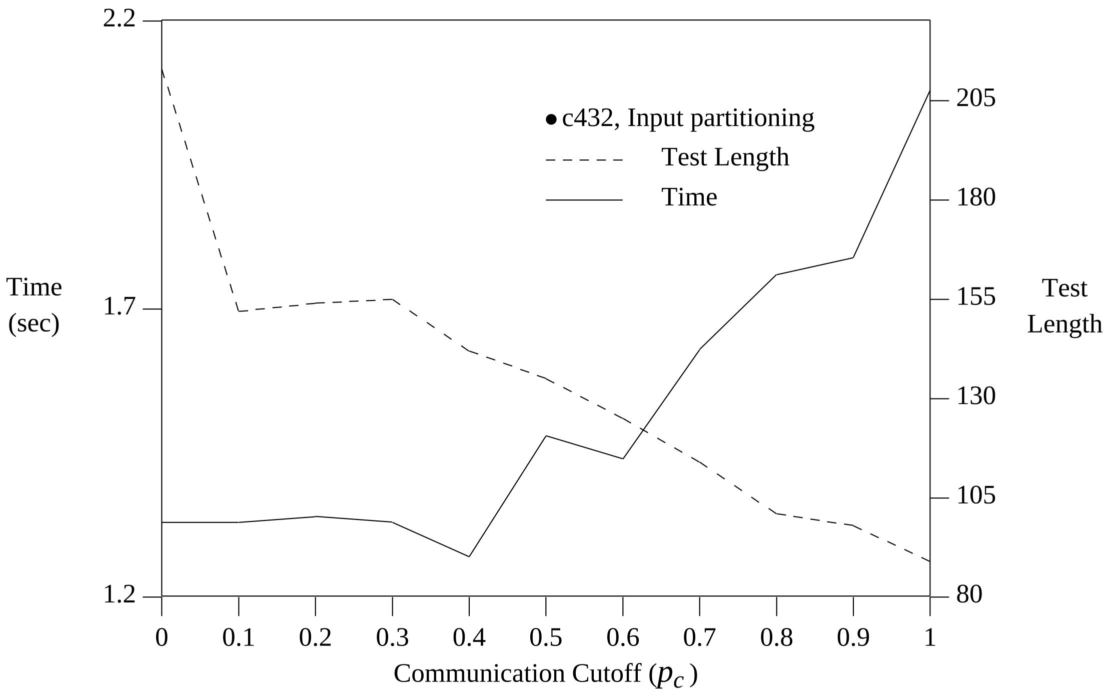


Figure 4.9. Communication of test vectors for random and MACP partitioning



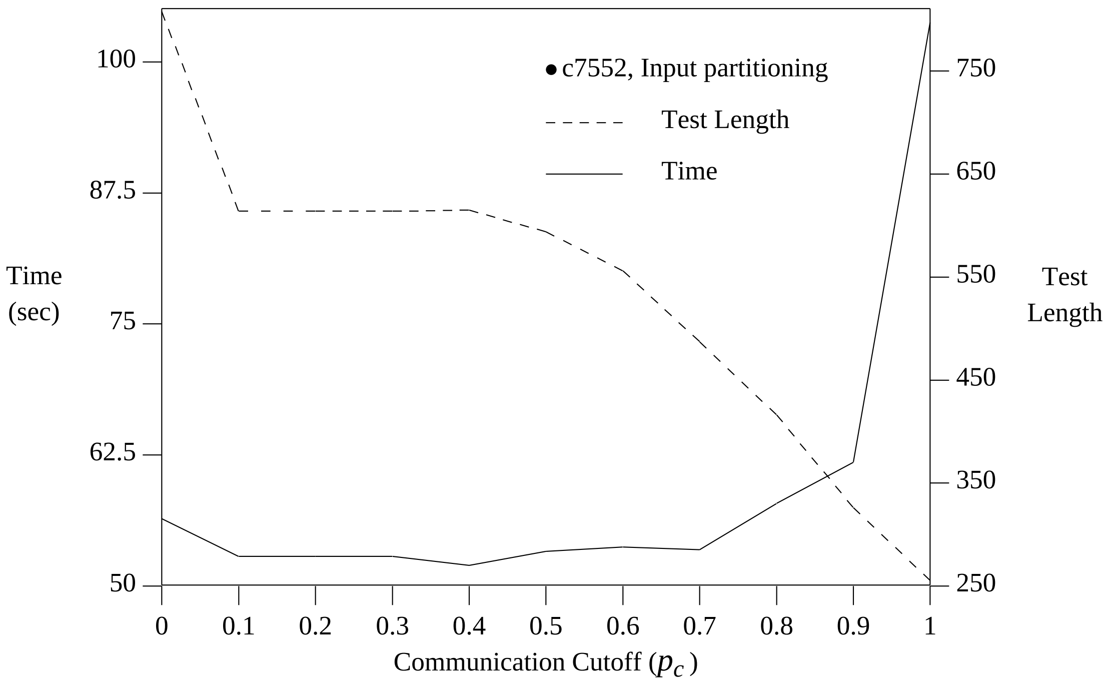


Figure 4.10. Communication of test vectors for input partitioning

Table 4.6. Results with communication cutoff = 0.6

	Random			Input Cones			Output Cones			MACP		
Ckts	Tests	Cov	S_{16}	Tests	Cov	S_{16}	Tests	Cov	S_{16}	Tests	Cov	S_{16}
c432	155	99.05	4.12	125	99.05	4.76	151	99.05	3.80	141	99.05	4.00
c499	143	98.95	5.24	123	98.95	5.30	121	98.95	5.33	125	98.95	4.55
c880	214	100.00	4.27	169	100.00	5.14	184	100.00	4.41	151	100.00	4.69
c1355	440	99.24	5.46	250	99.17	6.67	235	99.49	8.07	239	99.37	7.51
c1908	418	99.57	5.65	289	99.68	6.97	325	99.63	5.34	296	99.68	6.84
c2670	435	93.96	6.95	280	94.14	6.88	311	93.67	6.82	279	94.54	7.67
c3540	639	95.48	7.31	407	96.03	8.96	417	95.48	7.42	368	95.80	8.00
c5315	646	98.95	5.26	486	99.10	6.57	504	99.01	6.09	440	99.10	6.65
c6288	119	99.90	4.60	121	99.94	4.92	162	99.88	4.54	97	99.94	5.28
c7552	924	96.25	6.32	556	96.52	8.13	561	96.46	7.38	511	96.46	7.73
Rank	3.9		3.3	1.9		1.7	2.7		3.0	1.5		2.0

due to smaller test lengths. It can be seen again that partitioning by input cones gives better results than all other partitioning methods. Even though the circuit c6288 is one of the largest circuits, the speedup numbers obtained are low (about 4-5) due to the unusually low uniprocessor test length and high test lengths on the multiprocessor.

4.6. HIPERTEST

The fault partitioning techniques presented in this chapter and parallel search techniques presented in the previous chapter are used in a high performance parallel test generation system called HIPERTEST. The general flow of the program is shown in Figure 4.11. The test generation program is divided into two phases. During the first phase, fault parallelism is used to generate tests for faults which are easy to detect. The fault partitioning and load balancing algorithms detailed in Section 4.4 are used. A very low backtrack limit is used during this phase (25 backtracks). A communication cutoff factor (p_c) of 0.6 was used. The faults dropped during PHASE 1 are used as hard-to-detect (HTD) faults during PHASE 2. Each processor performs a

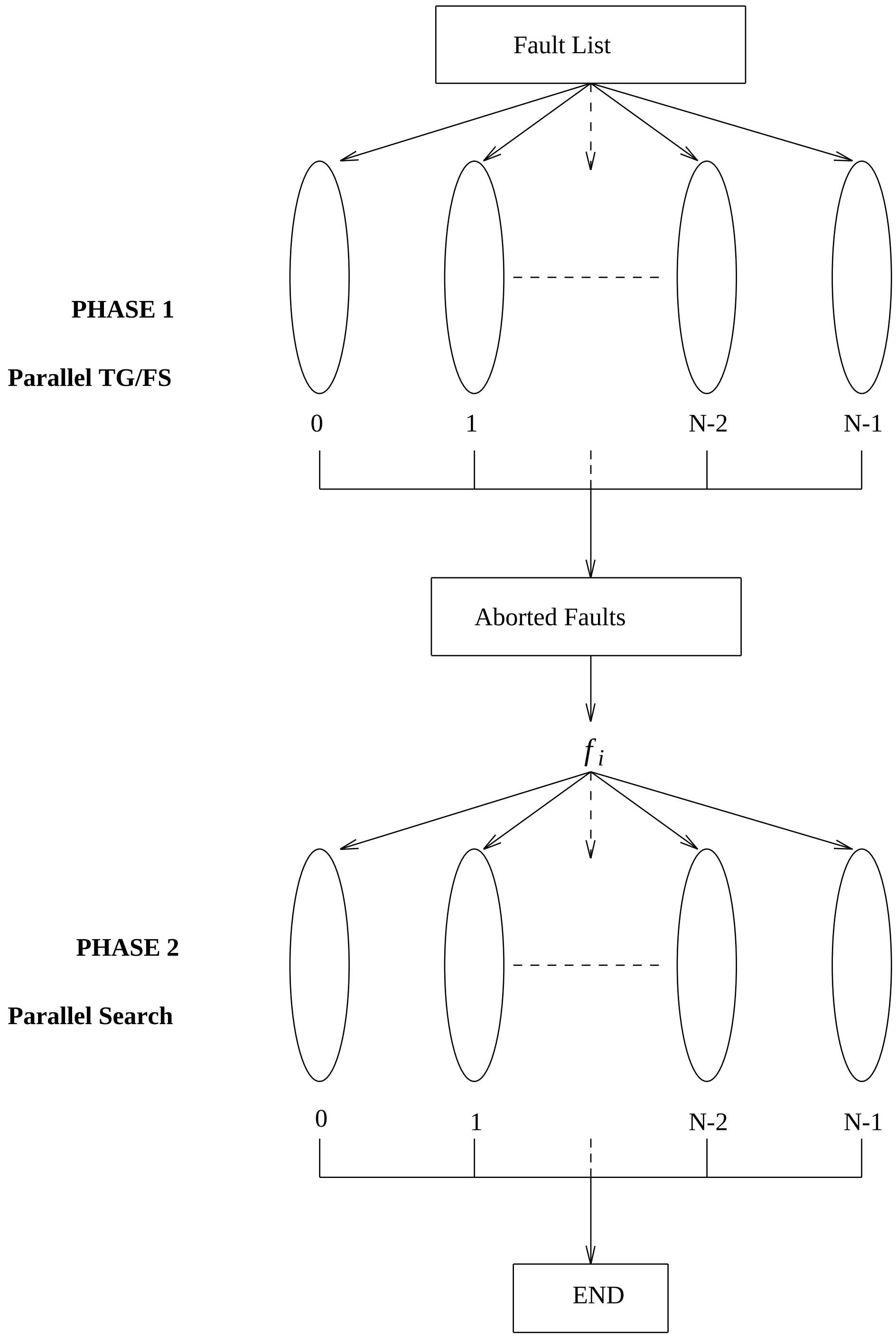


Figure 4.11. HIPERTEST

simple redundancy identification on the HTD faults before starting PHASE 2. The procedure used for redundancy identification is very similar to the techniques used in [14, 49]. It was found that many faults were identified to be redundant in this manner. Redundant faults identified during this step are removed from the HTD fault list. During PHASE 2, one of the hypercube processors is used as a scheduler and the rest act as processing nodes. Since the scheduler does not participate in the test generation process, the HTD fault list "owned" by the scheduler at the end of PHASE 1 is split equally among the remaining processors. The processing nodes then perform parallel branch and bound on one HTD fault at a time. After a test is generated for an HTD fault, each processor carries out fault simulation to find out whether any of the remaining HTD faults are detected by the same test vector. PHASE 2 terminates after all HTD faults have been processed. The performance of HIPERTEST based on runs on a 16 processor Intel iPSC/2 hypercube are shown in Table 4.7.

Table 4.7. Performance of HIPERTEST

	Phase 1			**Phase 2**			**Overall**		
Circuit	**Tests**	**Cov**	**Time(s)**	**Tests**	**Cov**	**Time(s)**	**Tests**	**Cov**	**Time(s)**
c432	254	99.24	1.43	1	0.19	0.91	255	99.43	2.34
c499	203	98.95	1.77	0	0.00	3.11	203	98.95	4.88
c880	309	100.00	1.89	0	0.00	0.00	309	100.00	1.89
c1355	333	99.49	4.92	0	0.00	5.13	333	99.49	10.05
c1908	476	99.68	7.27	2	0.11	4.20	478	99.79	11.47
c2670	543	95.49	16.13	37	1.56	75.35	580	97.05	91.48
c3540	686	97.55	31.56	23	0.70	100.30	709	98.25	131.86
c5315	758	99.29	27.04	24	0.45	25.20	782	99.74	52.25
c6288	431	99.88	24.09	9	0.12	6.62	440	100.00	30.71
c7552	956	96.44	64.21	170	2.26	239.76	1126	98.70	303.96

4.7. Summary

In this chapter we addressed the fault partitioning issues in an integrated parallel test generation/fault simulation environment. It was shown that that if one is not careful in the design of a parallel algorithm, apart from inefficient utilization of available processors, degradation in the quality of solutions can occur. Issues in exploiting fault parallelism for parallel test generation/fault simulation were identified and discussed. It was shown that coupling fault simulation with test generation results in a better utilization of the available processors. Requirements of an efficient fault partitioning scheme were discussed and heuristic schemes for partitioning faults were presented and evaluated. It was found that fault partitioning using input cones yielded better results both with respect to the overall test length as well as to the overall run time. A dynamic load balancing scheme which exploits the advantages of both static and full dynamic load balancing was discussed. It was shown that the dynamic load balancing scheme resulted in a more efficient utilization of the available processors. Also, a near optimal value of the communication cutoff factor was found experimentally which results in smaller test lengths without a significant increase in run time.

Finally, we combined the parallel search techniques provided in Chapter 3 with the fault partitioning techniques presented in this chapter to implement a high performance test generation system for combinational circuits called HIPERTEST.

CHAPTER 5.

PERFORMANCE ANALYSIS

5.1. Introduction

In this chapter we present techniques to analyze the performance of parallel search and fault partitioning techniques presented in Chapters 3 and 4. In Section 5.2 we present a performance analysis of parallel search and try to relate it to the size of the search space and the efficiency of the uniprocessor search heuristic. Two different approaches are presented. The first approach analyzes the performance of parallel search for redundant faults where the nodes expanded by the parallel algorithm are exactly the same as those expanded by the serial algorithm. In such a case, the analysis reduces to the analysis of a parallel algorithm using a simple divide-and-conquer approach. The second approach analyzes the performance of parallel search when the search space explored by the parallel algorithm is not the same as that explored by the uniprocessor algorithm. In such a case, a probabilistic analysis has to be used. In Section 5.3 we present an analysis of fault partitioning techniques. The effect of increase in test length and communication for test vectors is examined. The theoretical model is then used to predict the speedup for an example benchmark circuit.

5.2. Parallel Search

Two approaches are presented in this section. The first approach analyzes the performance of parallel search for redundant faults where the nodes expanded by the parallel algorithm are exactly the same as those expanded by the serial algorithm. In such a case, the analysis reduces to the analysis of a parallel algorithm using a simple divide-and-conquer approach. The second approach analyzes the performance of parallel search when the search space explored by the parallel algorithm is not the same as that explored by the uniprocessor algorithm. The behavior of the search algorithm can be analyzed by assuming that there are N random processes executing on N different processors whose objective is to find a test vector.

5.2.1. 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 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 W/N. It can be shown that the minimum number of stack splittings required such that each of the processors gets an amount of work equal to W/N is

$$N - 1$$

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 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 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 that if the number of primary inputs n is very large, the communication cost in the above expression is negligible.

5.2.2. Probabilistic analysis

This analysis is very similar to the analysis for randomized parallel branch and bound presented in [38]. 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 to get a closed-form expression for speedup for most realistic probability distributions such as the normal distribution.

However, simpler closed-form expressions can be derived based on experimental observations. It was shown experimentally in [21] 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 5.1 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 [21]. 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,

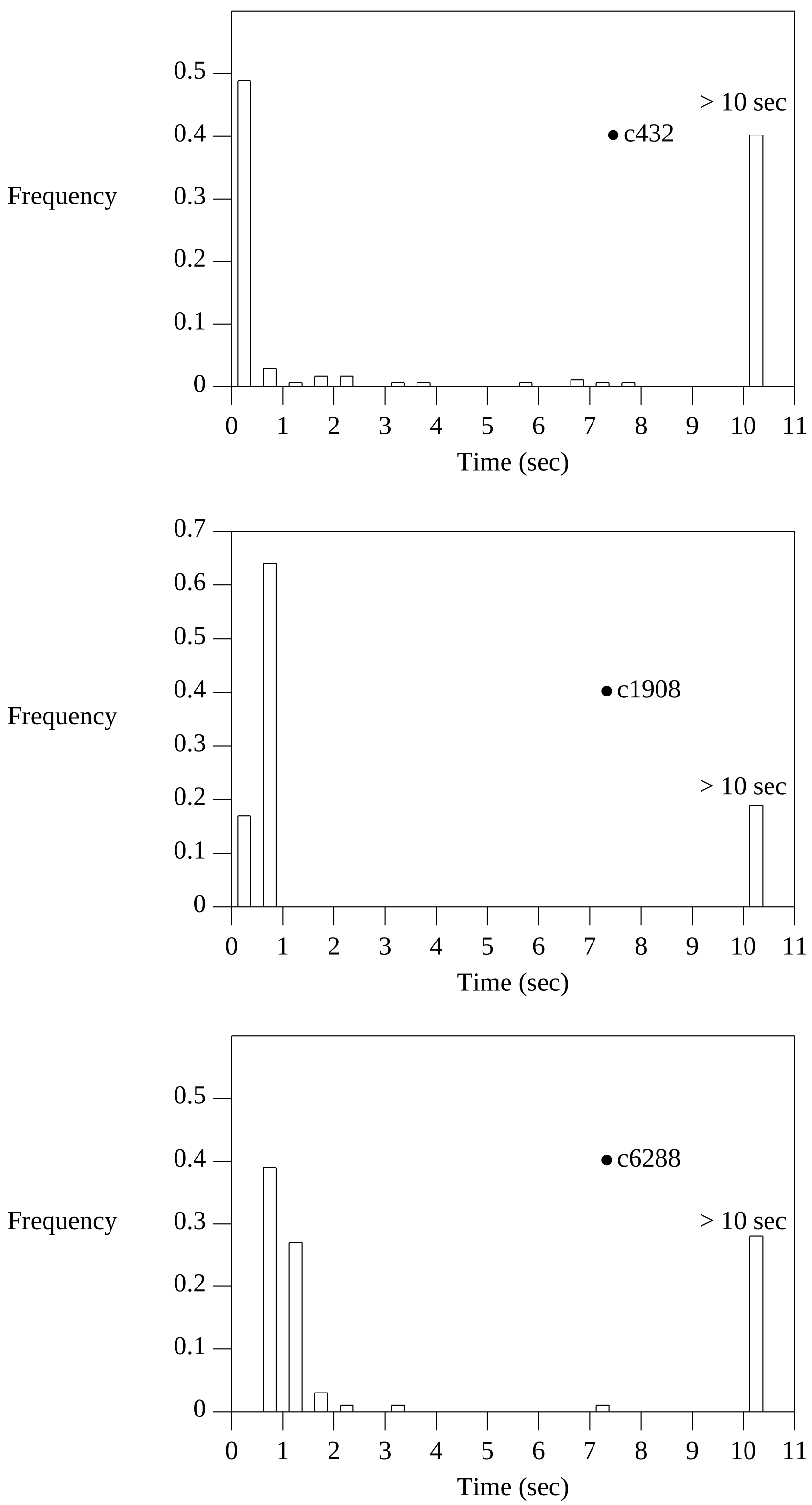


Figure 5.1. Run times using randomized search

where the parallel algorithm reduces to a divide-and-conquer approach, the analysis presented in Section 5.2.1 is more appropriate.

5.3. Fault Partitioning Issues

Let us first model the TG/FS process on the uniprocessor based on some simplifying assumptions. Figure 5.2 shows a plot of the fraction of faults left versus the fraction of total number of test vectors generated for four of the ISCAS benchmark circuits. This figure is the same as Figure 4.8 and is reproduced here for easy reference, where T_1 is the uniprocessor test

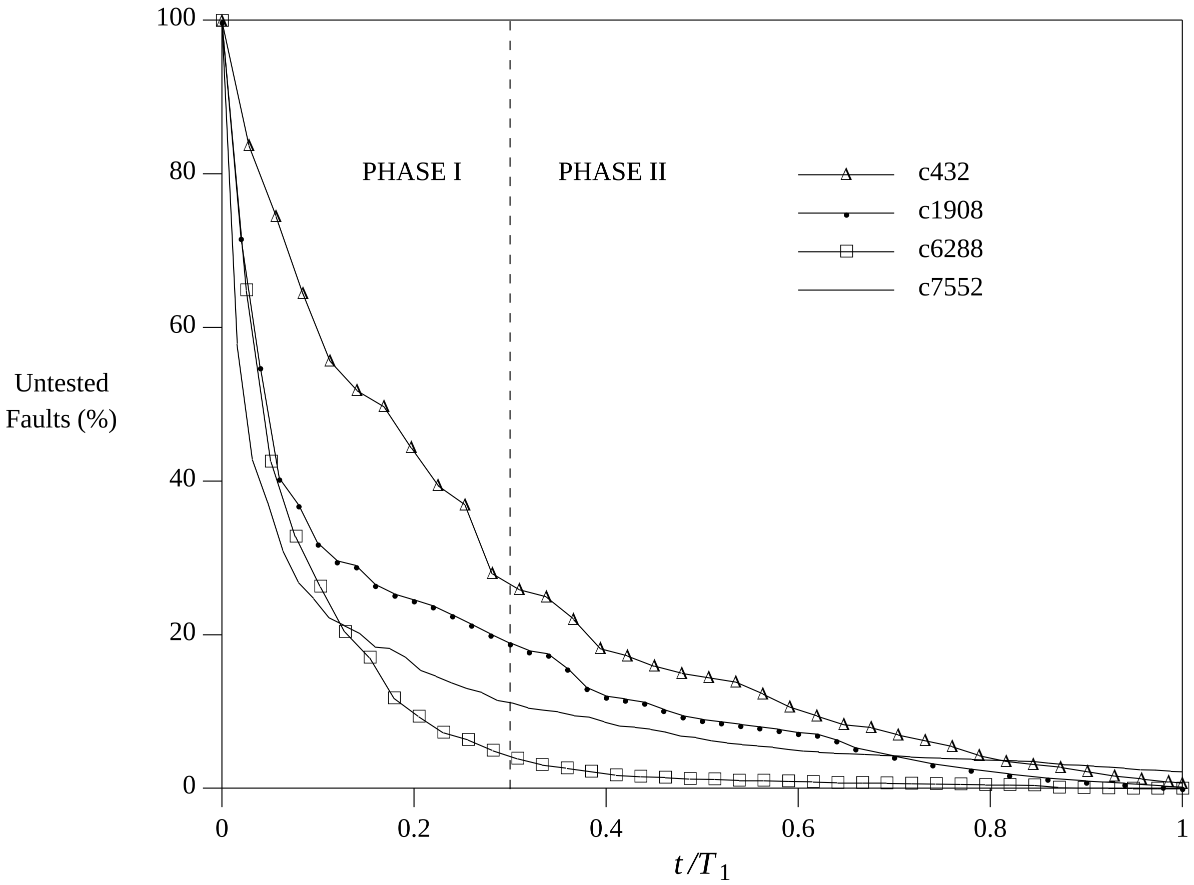


Figure 5.2. TG/FS profiles

length. It can be seen that the curve closely resembles a decaying exponential for each of the circuits. In [53] , a combination of a decaying exponential and a straight line was used to approximate the number of untested faults versus number of tests curve. The reasoning behind this assumption is that in a TG/FS process, initially a large number of faults are detected by the first few vectors. These are usually the faults amenable to random pattern testing. This initial phase (called Phase I in [53]) closely resembles a decaying exponential. After Phase I, the faults remaining are *random pattern resistant* faults. For such faults, test generation has to be done explicitly and very few additional faults are detected with each additional test vector apart from the fault for which the test vector was explicitly generated. The curve in this portion (called Phase II in [53]) closely resembles a straight line. In Figure 5.2 the vertical dashed line segment approximately shows the regions of the curve corresponding to Phases I and II. It can be seen that Phase I ends when approximately 20-30% of the test vectors have been generated and a fault coverage of 60-80% has been reached. Though this would model the actual TG/FS process more accurately, the error incurred by modeling the whole TG/FS curve by a decaying exponential is quite small as is apparent from the actual TG/FS curves shown in Figure 5.2. Thus, for the sake of simplicity, we shall assume that the TG/FS curve can be approximated by a decaying exponential. Throughout our analysis, we shall use the CPU time estimates as an estimate of the cost. The terms "cost" and "time" will be used interchangeably.

5.3.1. Estimating uniprocessor run time

Assuming an exponential fault simulation profile, let the fraction of faults left after t test vectors have been generated (i.e., after t TG/FS steps) be given by

$$F_0 e^{-kt/T_1}$$

where F_0 and k are constants and T_1 is the number of tests required to achieve the maximum possible fault coverage on the uniprocessor. The constants F_0 and k can be chosen according to some error minimization criterion. For example, F_0 and k could be chosen so that the sum of squares of errors

$$\sum_{i=1}^{T_1}(f_i - F_0 e^{-kt/T_1})^2$$

is minimized where f_i is the actual fraction of faults left after the i'th test vector.

Let G be the number of gates in the circuit under test and let n_f be the number of faults for which tests have to be generated. Let n_I be the number of primary inputs of the circuit under test and α be the fraction of gates evaluated after each primary input assignment. Let b_{av} be the average number of backtracks per fault and let β be the fraction of gates evaluated after each backtrack. Hence, the time spent at each test generation step on the uniprocessor is

$$\alpha n_I G + \beta b_{av} G = (\alpha n_I + \beta b_{av})G$$

In the process of generating T_1 test vectors, the test pattern generator may abort test generation for faults for which the backtrack limit is exceeded and also declare a few faults to be redundant. Let the number of faults dropped on the uniprocessor be D_1 and the number of faults found redundant be R_1. Thus, the total number of faults processed by the test pattern generator is $(T_1 + R_1 + D_1)$ and the total time spent in test generation on the uniprocessor, denoted by $C_{test}(1)$, is given by $(\alpha n_I + \beta b_{av})G(T_1 + R_1 + D_1)$. The number of faults to be simulated for the t'th test vector is the number of faults left after simulating the $(t-1)$'th vector, which is

$$F_0 n_f e^{-k(t-1)/T_1}$$

Let the fraction of gates evaluated at each fault simulation step for each fault being simulated be

γ and the fault simulation cost for the t'th vector is given by

$$F_0 n_f e^{-k(t-1)/T_1} \times \gamma G$$

The total fault simulation cost on the uniprocessor ($C_{sim}(1)$) is

$$\sum_{i=1}^{T_1} F_0 n_f e^{-k(t-1)/T_1} \gamma G$$

which simplifies to

$$C_{sim}(1) = F_0 n_f \gamma G \frac{1 - e^{-k}}{1 - e^{-k/T_1}}$$

and the total uniprocessor run time (for test generation and fault simulation) is given by

$$(\alpha n_I + \beta b_{av}) G (T_1 + R_1 + D_1) + F_0 n_f \gamma G \frac{1 - e^{-k}}{1 - e^{-k/T_1}}$$

5.3.2. Estimating the speedup on a multiprocessor

To calculate the multiprocessor run time, assume there are N processors. Let the test length on N processors be T_N. The ratio of test lengths T_N/T_1 is in some sense a measure of the redundant work done on the multiprocessor. A high value of T_N/T_1 indicates a large amount of redundant work. In general, T_N is dependent on the following:

(1) The static partitioning strategy: the better the strategy, the lower is the test length.

(2) The number of processors N: excluding test length anomalies of the kind discussed in Section 4.2, the test length normally increases with increase in N as demonstrated by experimental results presented in Section 4.5.

(3) The degree of communication allowed for test vectors among processors: the higher the communication, the lower is the value of test length (see Section 4.5).

Let each processor broadcast vectors to other processors for fault simulation until it has reached p_c fault coverage for its own faults. The parameter p_c is called the communication cutoff factor. Thus, each processor not only uses the test vectors generated during its own test generation phase for fault simulation, but also those generated by other processors. Until each processor reaches a fault coverage of p_c, it uses N vectors, where N is the number of processors, at each fault simulation step. We referred to the parameter p_c as the communication cutoff factor in Section 4.2. Since p_c is calculated relative to each processor's own fault list, no communication is required to decide when to stop sending test vectors to other processors. Also, at each fault simulation step, there is no explicit wait for vectors from other processors. Each processor processes whatever vectors have arrived at that time from other processors (all of them may not have arrived) in an asynchronous fashion.

Under perfect load balancing conditions, assume that each processor processes the same number of faults during the test generation phase. Also assume that each processor generates T_N/N vectors for n_f/N of the faults. In general, the above assumptions will not be true so the speedup estimates will be optimistic which will serve as an upper bound on the actual speedups on the multiprocessor. In reality, the speedup depends on the maximum work done by any processor rather than on the average work load. The test generation time for each processor is given by

$$C_{test}(N) = (\alpha n_I + \beta b) G (T_N + R_1 + D_N)/N$$

D_N is the total number of dropped faults on N processors. The number of faults found redundant on the multiprocessor will be the same as that on the uniprocessor since the backtrack limit is the same. Until the coverage reaches p_c, each processor also uses test vectors generated by other processors for fault simulation. The fault simulation cost is thus multiplied by a factor of N till a coverage of p_c is reached. Assuming an exponential relationship between the number of untested faults and the number of tests, the number of test vectors generated when a coverage of p_c is reached is given by

$$1 - F_0 e^{-Nkt/T_N} = p_c$$

or

$$t = t_c = \left\lceil \frac{T_N}{kN} \log \frac{F_0}{1 - p_c} \right\rceil$$

The total fault simulation cost is found by adding the costs for the first t_c vectors separately with that of the final $T_N - t_c$ vectors. The fault simulation cost is thus

$$N \sum_{i=1}^{t_c} F_0 \frac{n_f}{N} \gamma e^{-Nk(t-1)/T_N} + \sum_{i=t_c+1}^{T_N} F_0 \frac{n_f}{N} \gamma G e^{-Nk(t-1)/T_N}$$

which simplifies to

$$C_{sim}(N) = F_0 n_f \gamma G \left[\frac{1 - e^{-Nkt_c/T_N}}{1 - e^{-Nk/T_N}} + \frac{1}{N} e^{-kt_c/T_N} \frac{1 - e^{-Nk(T_N - t_c)/T_N}}{1 - e^{-kN/T_N}} \right]$$

We now try to estimate the communication overheads on the multiprocessor. The cost of communicating a test vector can be denoted by δn_I where n_I is the number of primary inputs. The cost of broadcasting a test vector is thus $(N-1) \times \delta n_I$. The analytical expression for communication we are going to obtain below is on the pessimistic side since we are assuming communication cannot be overlapped. The cost for communicating vectors using this expression should be considered an upper bound on the actual communication cost. Since each processor broadcasts t_c vectors, the total communication cost is

$$C_{vect}(N) = N \times (N-1) \delta n_I t_c = N \times (N-1) \delta n_I \left\lceil \frac{T_N}{kN} \log \frac{F_0}{1 - p_c} \right\rceil$$

It can be seen that every time a processor becomes idle, it initiates the dialog given in Section 4.4 under the procedure *get_work* which results in $2N$ messages where N is the number of processors. Of the $2N$ messages, $2N-1$ are short messages of the type QUERY, WRK_REQ or messages giving the length of the fault list. Only one message requires sending a part of the fault list. If the latency for short messages is C_{short} and the average delay in sending the fault list is proportional to the total number of faults n_f, the communication overhead when the dialog given above is invoked is given by

$$(2N-1)C_{short} + \omega n_f$$

where ω is some proportionality constant. For the parallel TG/FS process to terminate successfully, each processor has to execute the procedure *get_work* at least once. Thus, the total communication overhead due to dynamic load balancing is at least

$$C_{load}(N) = N \times [(2N-1)C_{short} + \omega n_f]$$

If the latency due to short messages is ignored, the communication overhead for load balancing grows only as fast as the number of processors N.

In calculating the communication cost above we have again ignored the fact that communication can be overlapped with computation and it can also take place in parallel. The communication cost expressions we obtained are biased more towards the pessimistic side. The speedup on N processors is given by

$$S_N = \frac{C_{test}(1) + C_{sim}(1)}{C_{test}(N) + C_{sim}(N) + C_{load}(N) + C_{vect}(N)}$$

where $C_{test}(1)$ and $C_{sim}(1)$ are test generation and fault simulation costs respectively, on the uniprocessor and $C_{test}(N)$ and $C_{sim}(N)$ are the corresponding values on N processors. $C_{load}(N)$ is the cost of dynamic load balancing on N processors and $C_{vect}(N)$ is the cost of communicating test vectors for fault simulation. To simplify the final speedup expression, assume $t_c = 0$ for the time being (i.e., no communication for test vectors takes place). Substituting expressions for the quantities given above, we obtain

$$S_N = \frac{\left[(\alpha n_I + \beta b_{av})G(T_1 + R_1 + D_1) + F_0 n_f \gamma G \frac{1 - e^{-k}}{1 - e^{-k/T_1}}\right]}{\left[(\alpha n_I + \beta b_{av})G \frac{T_N + R_1 + D_N}{N} + F_0 \frac{n_f}{N} \gamma G \frac{1 - e^{-Nk}}{1 - e^{-Nk/T_N}} + N[(2N-1)C_{short} + \omega n_f]\right]}$$

It can be seen from the discussion here that the analysis presented is quite different from the one presented in [29]. The analysis in [29] primarily dealt with the estimation of the size of the target faults that need to be sent by the central server so as to minimize communication and maximize speedup. The analysis is thus based on speedup alone and not the quality of the solution (e.g., test length). The analysis presented in this section tries to take into account both the speedup and the resultant effect on the quality. While a central scheduler is assumed in [29] we make no such assumptions and assume a completely distributed environment for load balancing and fault partitioning. By judicious combination of static and dynamic load balancing and by overlapping computation with communication, the communication overheads are kept low. Also, the above analysis can accommodate most multiprocessor MIMD architectures (shared or distributed memory). Architecture-dependent parameters can in fact be absorbed as a part of different constants presented above. However, for architectures based on synchronous communication and execution, e.g., an SIMD architecture such as the Connection Machine, an analysis different from the one presented above will have to be adopted. The parameters used in the above analysis are summarized in Table 5.1.

Table 5.1. Parameters in the performance model

G	Number of gates in the logic circuit
n_I	Number of primary inputs (PIs) of the logic circuit
n_f	Number of faults in the fault list
N	Number of processors
T_N	Test length on N processors
R_1	Number of faults found redundant
D_N	Number of faults dropped on N processors
F_0, k	Constants in the exponential fault simulation curve
α	Fraction of gates evaluated after each PI assignment
β	Fraction of gates evaluated after each backtrack
b_{av}	Average number of backtracks during test generation
p_c	Communication cutoff factor
C_{short}	Message latency for short messages
ω	Communication cost per fault for load balancing

5.3.3. Example

To demonstrate the utility of the analysis presented above, consider the following simple example. The TG/FS program was run on the uniprocessor on ten randomly selected faults for the circuit c6288 from the ISCAS combinational benchmark suite [43]. All parameters calculated here are for an Intel iPSC/2 hypercube. The test vectors generated during the test generation phase were used in the fault simulation phase for the whole fault list which consisted of 7744 faults ($n_f = 7744$). It was found that the test pattern generator was able to generate tests for all the ten faults without any backtracks ($b_{av} = 0$). The average test generation time was found to be 0.4 second per fault ($[\alpha n_I + \beta b_{av}]G = 0.4$ sec). Also, the total fault simulation time for all the ten vectors was found to be 28.65 seconds. The fault coverage for the ten vectors was 93.23%. The number of faults simulated during each fault simulation step were added up for all the ten vectors and the the value of the parameter γG was found to be 0.0012 second.

Thus, based on the parameters obtained from this partial TG/FS profile on a uniprocessor we are trying to estimate the speedup on a multiprocessor. The exponential fault simulation curve will not be able to fit cases where the final fault coverage is 100% because the exponential model will predict the test length to be infinite. In such a case the exponential-linear model of [53] will have to be used. However, the pure exponential model can be used if we assume the fault coverage to be less than but very close to 100%. Let us assume that the final fault coverage for c6288 is 99.99%. Thus, at $t = T_1$, the fraction of faults left is 0.0001. Assume the value F_0 is 1 so that $F_0 e^{-kt/T_1}$ becomes 1 when t is 0. Since, at $t = T_1$ the fraction of faults left is 0.0001,

$$e^{-k} = 0.0001$$

which means:

$$k = 9.21.$$

Having estimated the value of k, let us now try to estimate the uniprocessor test length (T_1) for c6288. Since a fault coverage of 93.23% was reached using ten vectors, the fraction of faults left at the end of ten vectors is 0.0677 which means:

$$F_0 e^{-10k/T_1} = 0.0677.$$

Substituting the values of F_0 and k we get the value of T_1 to be 34 which was found to be very close to the actual uniprocessor test length of 39 (for a fault coverage of 99.94%). In Figure 5.3 the theoretical fault simulation profile is compared with the actual profile for the circuit c6288. It can be seen that the theoretical fault simulation profile matches the actual one very well. Assume there are no dropped or redundant faults, i.e., $R_1 = D_1 = 0$. Thus the uniprocessor test generation time is

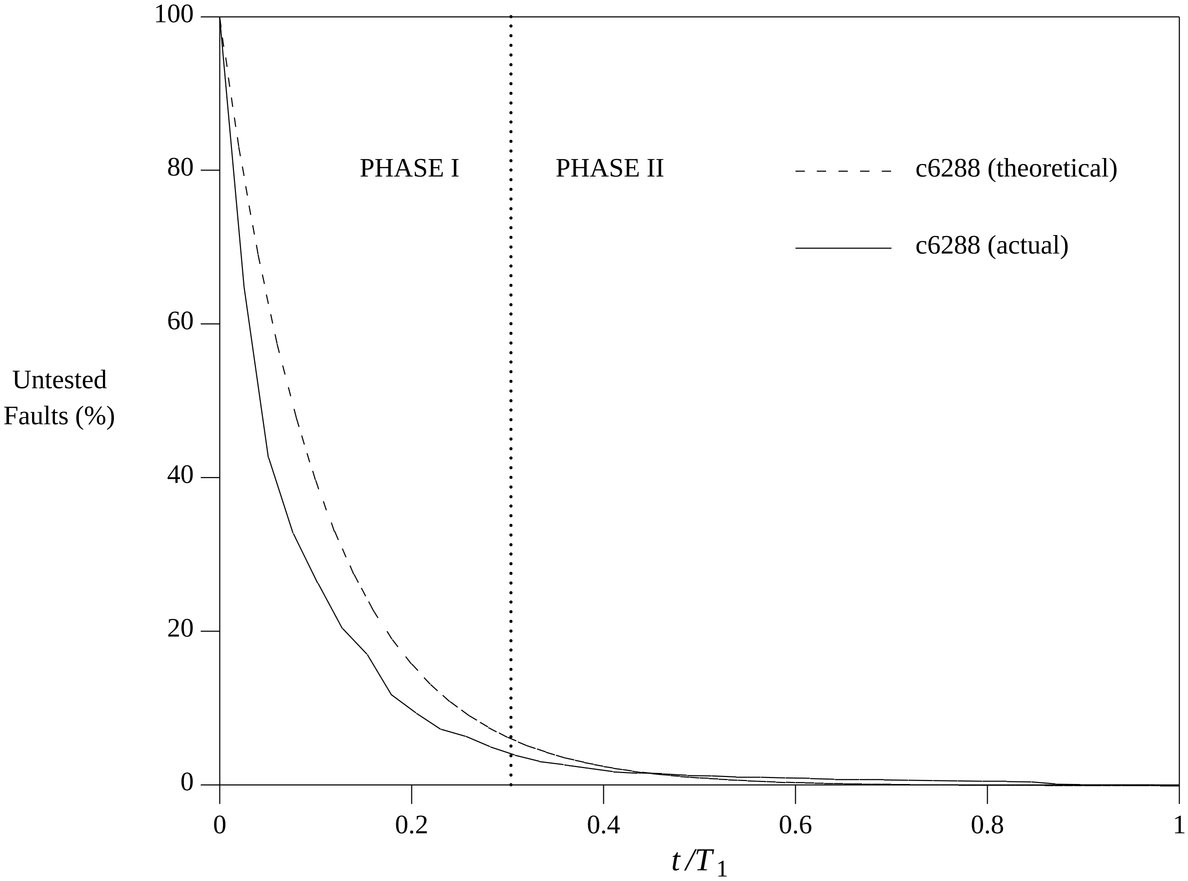


Figure 5.3. Actual and theoretical TG/FS profiles for c6288

$$C_{test}(1) = (\alpha n_I + \gamma b_{av})GT_1 = 0.4 \times 34 = 13.6 \; seconds$$

The uniprocessor fault simulation time is

$$C_{sim}(1) = F_0 \gamma G n_f \, \frac{1 - e^{-k}}{1 - e^{-k/T_1}} = 1 \times 0.0012 \times 7744 \frac{1 - e^{-9.21}}{1 - e^{-9.21/34}} = 39.16 \; seconds$$

Thus the total TG/FS time on the uniprocessor is 13.6 + 39.16 = 52.76 seconds. Let us now estimate the run time on 16 processors. Assume there is a ten-fold increase in test length on the multiprocessor, i.e., $T_{16} = 340$. Thus, $C_{test}(16) = 0.4 \times \frac{340}{16} = 8.5 \; seconds$. The fault simulation time is given by

$$C_{sim}(16) = 1 \times 0.0012 \times \frac{7744}{16} \times \frac{1 - e^{-9.21 \times 16}}{1 - e^{-16 \times \frac{9.21}{340}}} = 1.65\ seconds$$

To estimate the communication overhead, for the purpose of analysis, ignore the overhead in transferring the fault list ($\omega = 0$). On the Intel *i* PSC/2 hypercube, the value of C_{short} is 0.3 millisecond or 0.0003 second. Thus, communication overhead for load balancing is

$$C_{load}(16) = N(2N-1)C_{short} = 16 \times 31 \times 0.0003 = 0.1488\ second$$

Assume that there is no communication for vectors, i.e., $p_c = 0$. Thus, the total run time on 16 processors is $8.5{+}1.65{+}0.15 = 10.30\ seconds$ and the speedup is

$$S_{16} = \frac{C(1)}{C(16)} = \frac{52.76}{10.30} = 5.12$$

This is a rather surprising result since we would expect much more parallelism from fault partitioning. The speedup calculated above is rather on the optimistic side since we have ignored the communication overhead for communicating the fault list and the relative skew in test generation work loads because of unequal faults processed during the test generation phase. We have also not considered the effect of dropped or redundant faults. The highest actual speedup for the circuit c6288 for any partitioning heuristic (without communication for test vectors) was found to be 3.74 (Table 4.5). The partitioning heuristic which resulted in this speedup was found to be MACP. Also, the values of T_1 and T_N for MACP were found to be 39 and 367, respectively, which are very close to the assumptions made above. In the experimental results presented in Section 4.5, the communication overhead played a negligible part in determining the speedup. The term which determined the speedup was the test generation term $C_{test}(N)$ which in turn is dependent on T_N. The domination of the $C_{test}(N)$ with increase in N is also evident from the above calculations for $C(1)$ and $C(16)$. Thus, the test length is a dominant parameter in determining the speedup.

5.3.4. Memory overhead

The above analysis is based on examining the run time trade-offs. Another major component in the performance of any algorithm is the memory utilization. Since our implementation is based on a distributed-memory architecture, the circuit has to be replicated across processors to keep the communication cost low. Thus, it would seem as if the memory requirements of the parallel algorithm are N times the memory requirements of the uniprocessor algorithm, where N is the number of processors. However, on closer examination the memory requirements of the parallel algorithm turn out to be much lower. Only the circuit graph has to be replicated across processors and the dynamic lists generated during test generation normally consume very less memory due to the low memory requirements of an algorithm such as PODEM [13]. The memory consumed during fault simulation per processor is however *reduced* as the number of processors is increased since the fault list is now partitioned across several processors and the resultant dynamic lists generated during fault simulation will be shorter.

Most present day distributed memory machines lack virtual memory. Thus, memory utilization can become a bottleneck for very large circuits if the circuit graph cannot fit into the physical memory of each processor. It may not even be possible to run the parallel algorithm for very large circuits (larger than 50,000 gates) by simple replication. In such a case, the circuit graph may have to be partitioned on the basis of high-level modules or input/output cones so that test generation and fault simulation for different modules can proceed concurrently. In case the circuit is partitioned by input/output cones, only the regions belonging to more than one cone need be replicated.

However, virtual memory for distributed memory machines is a very active area of research [54] and we feel that the demands on physical memory will go down as new distributed-memory machines are introduced which incorporate virtual memory. In the case of shared-memory machines, where "messages" are basically read and write operations to the shared memory, the circuit graph need not be replicated and can be resident in the shared memory which can be accessed by all processors. Only the processor specific dynamic information need be replicated.

5.4. Summary

In this chapter we presented techniques to analyze parallel search and the effect of fault partitioning techniques on the performance of parallel test generation algorithms. If only redundant faults were considered, it was shown that the analysis of parallel search reduces to the analysis of a parallel algorithm with a simple divide-and-conquer approach. We also considered the case in which the search space explored by the parallel algorithm is different from that explored by the serial algorithm. A probabilistic approach was proposed to handle such a case. We proposed a performance model to evaluate the effect of fault partitioning and communication on the performance of a parallel test generation algorithm. The performance model takes into account the effect of increased test length, overhead for communication for test vectors and the overhead of the dynamic load balancing algorithm. Finally, we examined the memory overhead in a fault-parallel environment and suggested techniques to overcome memory bottlenecks for large circuits.

CHAPTER 6.

PARALLEL TEST GENERATION FOR SEQUENTIAL CIRCUITS

6.1. Introduction

Generating a test for a fault in a sequential circuit (a circuit with memory elements) requires searching the input space of a network for a vector sequence which distinguishes the faulty machine from a good machine. It has been shown that the problem of test generation is NP-hard [10, 11] even for combinational circuits. Most test generation algorithms use heuristics to guide the search for a vector sequence. Though many of the originally proposed test generation algorithms were for combinational circuits [12-14], they can be modified to generate tests for sequential circuits. However, even for moderately large sequential circuits, the test generation time can be prohibitive. Thus, speeding up test generation for sequential circuits using better uniprocessor heuristics [55-61] or by using multiprocessing hardware still remains an active area of research. This chapter concentrates on using general-purpose multiprocessing hardware to speed up sequential circuit test generation.

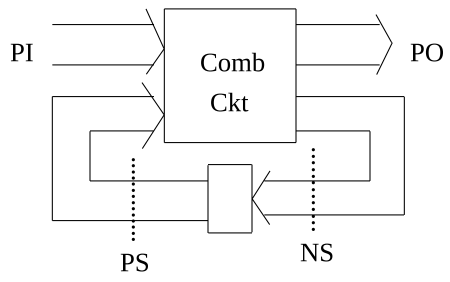
Due to similarities in the techniques used for test generation for combinational and sequential circuits, similar parallelization techniques may be applied to sequential circuits. However, as we discuss later, there are some features that are unique to sequential circuit test generation which render simple extensions of parallel combinational test generation techniques inefficient. One major contributing factor is the size of the search space for sequential circuit test generation. Also, due to the effect of feedback paths, fault partitioning techniques previously proposed for combinational circuits [7, 62] may not be applicable or may lead to unacceptable results in both the quality and efficiency of the parallel algorithm. In this chapter, we present parallel TG/FS algorithms for synchronous sequential circuits. The algorithms are evaluated using the ISCAS sequential benchmark circuits [63] on an 8 processor Encore Multimax shared-memory multiprocessor.

The rest of this chapter is organized as follows. In Section 6.2 we present a brief outline of algorithms for sequential circuit test generation. We will assume that the sequential circuits under consideration are synchronous in nature. In Section 6.3 we present algorithms for parallel fault simulation of sequential circuits. We discuss parallel test generation techniques for sequential circuits in Section 6.4. In Section 6.5 we present experimental results using the ISCAS sequential benchmark circuits on an Encore Multimax shared-memory multiprocessor.

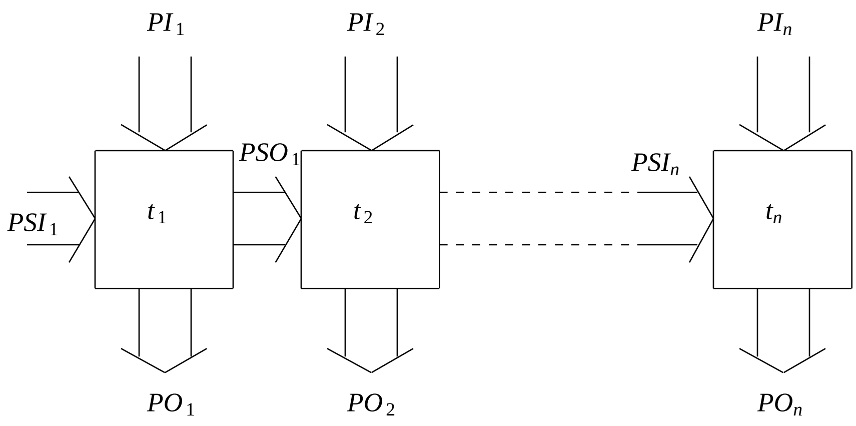
6.2. Test Generation for Synchronous Sequential Circuits

Most test generation algorithms use the iterative logic array (ILA) model for synchronous sequential circuits and use modified versions of the combinational circuit test generation algorithms such as PODEM and the D-algorithm. For example, the implementations reported in [56, 57, 59] are based on the D-algorithm and the implementations reported in [58, 61, 64] are based on PODEM. However, only the forward propagation part of the algorithm reported in [58] resembles PODEM while the fault excitation sequence is found by using a partially enumerated state transition graph. The enumeration of the state transition graph is also done by using an implicit enumeration technique similar to PODEM by assuming a reset state. The algorithm reported in [15] does not use an ILA model but uses a cost-function to guide the search for a test vector. Also, the algorithm reported in [15] can also be used for asynchronous circuits. The algorithm reported in [60] uses the Boolean cover of the sequential machine (representing the next state logic) for test generation. Our implementation of a test pattern generator for synchronous sequential circuits is based on PODEM and uses the conventional techniques of *forward-time processing (FTP)* and *reverse-time processing (RTP)* [56] for test generation. The serial test generation algorithm is discussed in more detail in the following paragraphs.

A conceptual model of a synchronous sequential circuit is shown in Figure 6.1 (a) and its ILA interpretation is shown in Figure 6.1 (b). The lines collectively labeled *PI* are the primary inputs and the lines labeled as *PO* are the primary outputs. The lines labeled *PS* are the present state lines and the lines labeled *NS* are the next state lines. The next state lines feed into a bank



(a)



(b)

Figure 6.1. (a) Sequential machine (b) ILA model

of memory elements (usually D-type flip-flops) and emerge as the present state lines. Thus the logic values on the present state lines assume the values of the next state lines at the next clock. We will be assuming an implicit single-phase clock. To represent the sequential machine as an ILA, the state lines at the output of the memory elements are "cut" and the machine is unrolled across time frames. The present state lines are referred to as pseudo-inputs (PSI) and the next state lines are referred to as pseudo-outputs (PSO) in the ILA model. Thus, the pseudo-outputs of one time frame form the pseudo-inputs of the next time frame. Using this ILA model, any conventional combinational test generation algorithm can be used with a few modifications. Thus, the time complexity of a sequential machine is transformed into space complexity by an ILA.

We now describe how test generation is performed using the ILA representation of a sequential circuit. First, a fault is injected into each time frame. The five-valued logic $(0,1,D,\overline{D},X)$ used by combinational test pattern generation algorithms is inadequate for sequential circuits since it does not take care of the repeated effect of faults across time frames. Also, the five-valued logic results in over-specification of signal values which may make justification of signal values more difficult. Thus, a nine-valued logic [65] is used which results in a minimal specification of signal values, which are represented as a pair of logic values denoting the good circuit and fault circuit values. Since each of the circuits (good and faulty) can assume three logic values (0, 1 and X) we have nine-logic values. Thus a value 1/0 denotes a value 1 in the good circuit and a value 0 in the faulty circuit (called a "D" in the five-valued logic). A value 1/X indicates that the good circuit value is 1 and the faulty circuit value is unspecified (or a "dont-care").

The first phase in the test generation process consists of *forward-time processing (FTP)* where the test vectors are generated in the order in which they are applied. In the FTP phase, the fault is propagated across time frames until it reaches a primary output. Testability measures are used to guide the propagation so that the FTP phase can be accomplished using as few time frames as possible. The FTP phase results in an assignment of values to the pseudo-inputs and primary inputs in the time frames necessary to propagate the fault. The vector specifying the pseudo-input values in the time frame in which the fault is first excited specifies the initial state S_0 in which the machine should be for the fault propagation and excitation to be successful.

The *reverse-time processing phase* or the *state justification phase* is triggered by the initial state vector S_0 generated by the FTP phase. The purpose of the RTP phase is to generate a sequence of vectors which result in the initial state S_0. Since these vectors are generated in an order opposite to the order they are applied, it is called the reverse-time processing phase. The RTP phase starts with the initial state S_0 and keeps going back in time until an all-unknown state or a reset state is reached. Some test pattern generators may also try to utilize the state generated by a test vector sequence generated for the previous fault [61]. Also, the algorithm reported in [58] tries to generate the vector sequence necessary to reach the initial state directly from the partial state transition graph. We use the PODEM algorithm to generate vectors during the RTP phase. Given a next state vector which has to be realized at the end of the current time frame, only one physical time frame has to be used to generate the present state vector and the input vector. The present state vector forms the next state vector for the next RTP step and the process stops when the present state vector is an all-unknown state or a reset state.

If all test vector sequences are derived independent of the state at the end of the test vector sequence for the previous fault, the sequences are said to be *self initializing*. However, most fault simulators assume the state at the end of the previous vector sequence (instead of starting from an all unknown state) to increase the fault coverage. Thus the test vector sequences have to be applied in the same order as that used by the test pattern generator/ fault simulator to guarantee the same fault coverage. Our test pattern generator is able to operate in three modes: (1) it can generate self initializing sequences, (2) it can make use of a reset state if one is available or (3) it can make use of the state at the end of the previous vector sequence generated for the previous fault. Also, during the process of FTP or RTP, the test pattern generator has a mechanism to handle repeated states.

Since the problem of test generation for sequential circuits can be expected to be harder than that for combinational circuits, the testability measure used can have a significant impact on the overall performance. The testability measure used should result in as few assignments on the primary inputs as possible and also should lead to shorter test sequences to reduce test generation time. The testability measure should bias the assignment of values so that very few pseudo inputs are assigned values. The combinational SCOAP [44] heuristic has been widely used for test generation for combinational circuits and has been shown to be very efficient for most combinational circuits [21, 22]. We use a slightly modified version of the sequential SCOAP heuristic [44]. The algorithm iterates through the SCOAP values along feedback paths until either the controllability/observability values stabilize or an iteration limit is exceeded. When the iterations are started, the controllability of pseudo-inputs and the observability of pseudo-outputs are set to infinity (some high integer value).

Another measure which is useful in pruning the search space and reducing the memory requirements for test generation is an estimate of the number of time frames necessary to generate a test for a fault. The *sequential depth* [66] of a node is the minimum number of flip-flops between any primary input and the node under consideration. Since this particular definition does not take into account the effect of intervening logic, we define the *logical sequential depth* as the minimum number of time frames necessary to realize a logic value on a particular line. We also define the *observation depth* as the minimum number of time frames necessary to observe the fault effect on a particular line at any primary output. The minimum number of time frames necessary to generate a logic value 0 will be called 0-sequential depth and the corresponding value for logic value 1 will be called 1-sequential depth. The algorithm used to compute the sequential depth values iterates through the network until the values stabilize or the iteration limit is exceeded. If the 0- and 1-sequential depths of a line n are indicated by $depth_0[n]$ and $depth_1[n]$, and the observation depth by $obs\_depth[n]$, the following expressions show how the values at each line are updated at each iteration step:

Initial values:

Primary inputs: $depth_0[n] = depth_1[n] = 1$.

Primary outputs: $obs\_depth[n] = 1$.

Pseudo-inputs: $depth_0[n] = depth_1[n] = \infty$.

Pseudo-outputs: $obs\_depth[n] = \infty$.

Sequential depth values:

AND gate: $\begin{cases} depth_0[n] = \min_i depth_0[i] \\ depth_1[n] = \max_i depth_1[i] \end{cases}$ where i is an input of n.

OR gate: $\begin{cases} depth_0[n] = \max_i depth_0[i] \\ depth_1[n] = \min_i depth_1[i] \end{cases}$.

NOT gate: $\begin{cases} depth_0[n] = depth_1[i] \\ depth_1[n] = depth_0[i] \end{cases}$.

Memory element (flip-flop): $\begin{cases} depth_0[n] = depth_0[i] + 1 \\ depth_1[n] = depth_1[i] + 1 \end{cases}$.

Observability depth values:

For a fanout stem i with a fanout branch n:

$$obs\_depth[i] = \min_n obs\_depth[n].$$

If i and m are inputs of an AND gate whose output is n:

$$obs\_depth[i] = obs\_depth[n] + \max_{m \neq i} depth_1[m].$$

Similarly, for an OR gate:

$$obs\_depth[i] = obs\_depth[n] + \max_{m \neq i} depth_0[m].$$

For a NOT gate with output n and input i: $obs\_depth[i] = obs\_depth[n]$.

For a memory element with output n and input i: $obs\_depth[i] = obs\_depth[n] + 1$.

The test length estimate is calculated as follows:

Test length for n s-a-0: $depth_1[n] + obs\_depth[n] - 1$.

Test length for n s-a-1: $depth_0[n] + obs\_depth[n] - 1$.

The above estimates of test length will be used to calculate the degree of difficulty of generating a test for a fault. It should be noted however, that the test length estimate is biased more towards the optimistic side. The estimates can be made pessimistic by replacing all *max* operations by "+" (addition) operations. Also, the test length estimate can be made more accurate by including the state of the faulty circuit. A test length with a value infinity indicates that the fault is either un-initializable or redundant. We mark all such faults *un-initializable* and delete them from the fault list.

After a test for a particular fault has been generated, fault simulation is carried out by using a highly optimized deductive [16] fault simulator. The fault simulator is very memory efficient and is able to handle even the largest ISCAS circuit without any need to use multiple passes.

6.3. Parallel Fault Simulation

The basic event-driven deductive fault simulator for synchronous sequential circuits uses a levelized circuit graph and processes the events in level order. This allows us to evaluate the events in the most efficient fashion since each gate will be evaluated exactly once. In conventional event-driven simulation algorithms, since events may be processed in any arbitrary order, a gate may be evaluated more than once (as many times as the number of inputs of the gate). In the case of deductive fault simulation, it is critical that a gate be re-evaluated as few times as possible since the cost of computing the fault list at a gate output can be very high (since each such evaluation will involve a series of list intersection and merge operations).

We use a circuit partitioned approach to parallel fault simulation since it allows us to exploit parallelism in fault simulation at the lowest granularity. Various techniques have been proposed in the past for parallel logic simulation using circuit partitioning [26]. The most obvious (and least efficient) technique is to maintain a single global event queue from which different processors obtain events for processing. The processors synchronize at each gate level before proceeding to evaluate the events scheduled for the next gate level. Though this technique would ensure that the processor loads are balanced at all times, the contention for the single event queue can limit the speedups severely. It may not be possible to obtain speedups of more than 2-3 [26]. To tackle this problem, a distributed event-queue approach was suggested in [26] which allows event updates to be done in parallel without the need to lock a global event queue. Our approach also uses distributed event-queues but is different from the approach suggested in [26] in the sense that the circuit partitioning is done statically. We will discuss our approach in more detail in the later part of the section.

The need for an event queue can be dispensed with by using a *compiled code* simulation approach where a gate is evaluated irrespective of whether the inputs of the gates have changed. However, even though the speedups obtained over the corresponding uniprocessor algorithm may be close to linear, the overall run times will still be much higher than that for the corresponding parallel event-driven simulation algorithm because of the extra computation involved for gates whose outputs have not changed.

Another approach is to evaluate the gates in an asynchronous fashion, i.e., process the events as they arrive instead of synchronizing at each gate level before proceeding to the next level. This method gets rid of the barrier synchronization needed at each gate level. However, this might entail re-evaluating a gate a number of times. In a logic simulation environment, where the process of re-evaluating a gate is basically a table look-up operation, the overheads are bound to be low. Also, since the number of logic values possible at the output of a logic gate is very low, the probability that a gate output is going to change and initiate further events is also low. Thus optimistic asynchronous approaches such as the Time Warp [67] and Chandy-Misra [68] algorithms may be well-suited for logic simulation. However, in deductive fault simulation, since the effect of all the faults is evaluated concurrently, the probability that a gate output is going to change is very high. Since re-evaluation of gate outputs is going to be more expensive than in the case of logic simulation, the performance gains will not be very high. Thus, our approach avoids re-evaluating the gates by constraining events to be evaluated in level order.

Our parallel fault simulation approach first assigns gates statically to each of the processors. Thus, normally, a processor which "owns" the gate will process an event associated with that gate. After the initial static assignment, a dynamic assignment of events takes place when processors become idle. Due to the efficiency of the dynamic allocation, we found that the static assignment of gates has very little effect on the performance of the parallel fault simulator as long as roughly an equal number of gates are assigned to each processor. Once the gates are assigned to each processor, each processor generates a list of gates assigned to itself. Also, the list is partitioned into L separate lists where L is the number of levels in the logic circuit. A list for level l for a processor p contains all gates at level l which are assigned to processor p. Barrier synchronization takes place at each level of the logic circuit so that all processors are done with processing gates at level l before proceeding with level $(l+1)$. Each gate has a shared flag associated with it which is set when that particular gate is scheduled for evaluation and reset when the gate has been evaluated.

Thus, if a gate g owned by processor p is to be scheduled for evaluation, the processor generating the event sets the flag associated with that gate instead of making an entry in the event list for processor p which would require locking of the event list. The gate evaluation flags can be set by different processors without the need to lock the data structure for that gate. Before a processor begins evaluating the gates scheduled for evaluation at level l, the processor goes through the list of gates owned at processor l and generates an event list for all gates whose evaluation flag is set. Thus, this event list generation can proceed concurrently. Once the event list is generated, the evaluation of gates at level l begins. The dynamic load balancing strategy (discussed next) however necessitates that the event queue be locked when an event is deleted from the queue for processing. However, the contention for the queue is very low since other processors will be accessing the queue only when they become idle. Also, a count is maintained which shows the number of events left unprocessed at level l by processor p.

Even though an equal number of gates are assigned to different processors, it is very likely that the processor workloads are skewed due to unequal activity in different portions of the logic circuit. This is taken care of by a dynamic load balancing strategy which supplements the static allocation. When a processor p_i exhausts all events owned by itself at level l, it starts looking into the event lists owned by other processors. Then p_i calculates the processor ID p_d of the processor which owns the largest unprocessed event list at level l. If the length of the event list owned by p_d is greater than a threshold MIN_EVENTS, processor p_i gains access (by locking) to the event queue owned by p_d, and divides the events scheduled for evaluation on p_d equally between itself and p_d. If the number of events left is less than MIN_EVENTS, processor p_i simply waits at the barrier for all processors to complete evaluation of gates scheduled at level l.

The uniprocessor version of the deductive fault simulator incorporates efficient memory retrieval techniques which are based on updating the reference counts associated with dynamic lists generated by different gates. However, in a parallel environment, the reference counts may be updated by different processors which would require mutual exclusion. Instead of using reference counts, the parallel algorithm contains a list of gates at each level whose fault lists can be retrieved. These memory retrieval lists at each level are also divided equally among different processors so that memory retrieval can proceed in parallel. Also, the deductive fault simulation is preceded by a phase where the state of the good circuit is evaluated. It has been shown in [53] that the fault simulation activity decreases almost exponentially with the number of test vectors since fewer faults per vector are detected as the number of vectors is increased. Thus, the good circuit simulation dominates the computation time as the number of vectors is increased and it becomes necessary to speed up good circuit simulation. The state of the good circuit is also evaluated in parallel using techniques very similar to those used for parallel fault simulation. Using the techniques discussed above, it was possible to get speedups of 6 on 8 processors during the fault simulation phase for the largest ISCAS circuit.

As an example, consider the synchronous sequential circuit shown in Figure 6.2. The circuit is first levelized starting from the pseudo-inputs and the primary inputs. Thus, the circuit has seven levels, denoted by L_1 through L_7. The vertical lines in Figure 6.2 are the points for barrier synchronization during the fault simulation process. Thus, all gates at a particular level can be evaluated in parallel since all of the inputs of any gate at that level would have been evaluated. For example, gates 14 and 15 can be evaluated in parallel when level L_3 is evaluated. Also, memory retrieval at each level can take place in parallel. For example, after evaluation of gate 20 at level L_5, memory associated with fault lists of gates 18 and 19 can be returned to the free

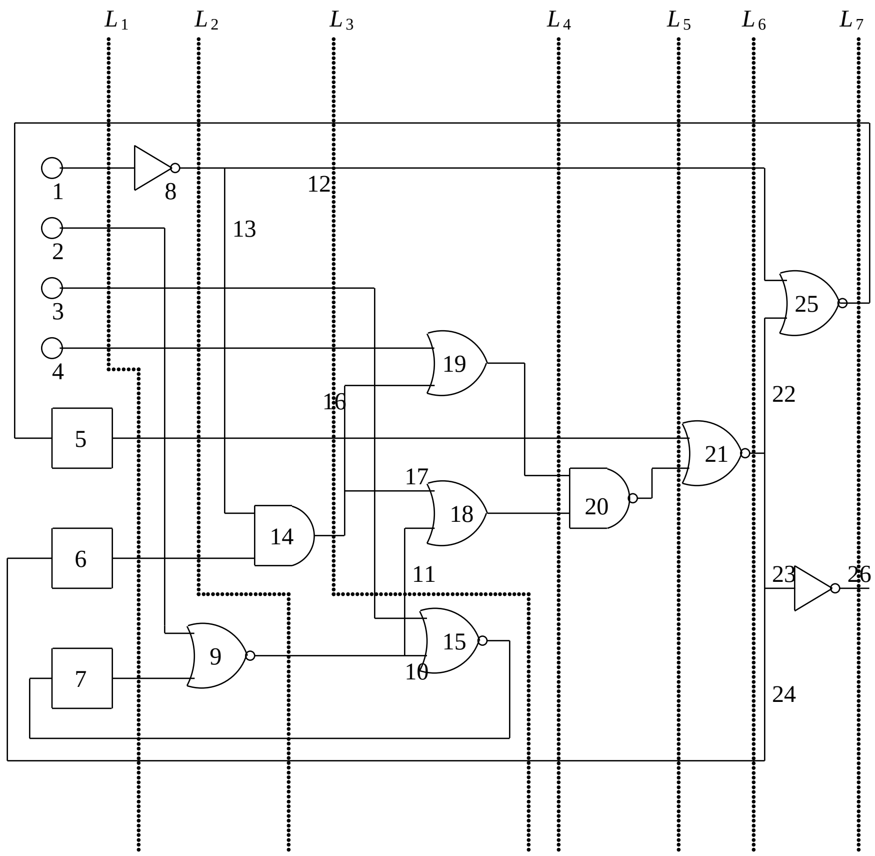


Figure 6.2. Parallel fault simulation

pool. The minimum time to evaluate a given circuit in parallel is dependent on the time required to evaluate the gates along the critical path in the logic circuit [69].

6.4. Parallel Sequential Circuit Test Generation

6.4.1. Distributed backtracking

One of the main bottlenecks in any sequential circuit test generation algorithm is memory usage. Due to the time-space transformation of the sequential circuit into an ILA, the memory usage increases with the test sequence length for a single fault. Multiprocessors offer increased computing power as well as more memory. Also, due to memory limitations on a uniprocessor, conventional sequential circuit test generation algorithms defer justification of values on pseudo-inputs until all requirements on the current time frame have been satisfied. This might impede early detection of conflicts since some of the assignments already made may be unjustifiable. This will not be discovered until the processing for the current time frame is completed and another time frame is expanded. To clarify the point further, consider a sequential circuit in the reverse-time processing phase. Given the required next state vector (values to be realized on the pseudo-outputs), the test generation algorithm tries to come up with a primary input assignment and a present state vector (assignments to the pseudo-inputs) which will give rise to the next state vector. However, the assignments on the pseudo-inputs are delayed until all the requirements for the current time frame have been satisfied.

This is an example of a space-time tradeoff where early detection of a conflict is sacrificed to save space. If instead of waiting for all assignments in the current time frame, the justification was performed all the way backward in time as soon as a value to a pseudo-input is assigned, it is very likely conflicts can be discovered early during the test generation process, resulting in an effective pruning of the search space. Since a uniprocessor implementation would require physically expanding all the time frames at once, this method is normally not used due to memory limitations. Let us now examine how this can be accomplished on a multiprocessor by distributing time frames *across* processors. This method can also be thought of as a circuit partitioned approach to parallel test generation where the circuit partitions are entire time frames.

Figure 6.3 shows each time frame in the ILA model of a sequential circuit and the decision tree associated with each time frame during test generation. Suppose we have p processors and we are trying to generate a test for a fault which requires n time frames. Thus we can assign $[n/p]$ time frames to each processor where a processor is assigned time frame i if $(i \ mod \ p)$ equals the processor ID (where the processor IDs are assumed to be from 0 to $(p-1)$). This results in a "wrap-around" assignment of time frames where consecutive time frames are assigned to different processors. Since the test generation algorithms proceed through time frames in sequence (i.e., two consecutive assignments as a result of a procedure such as backtrace either are made in the same time frame or in neighboring time frames), this assignment

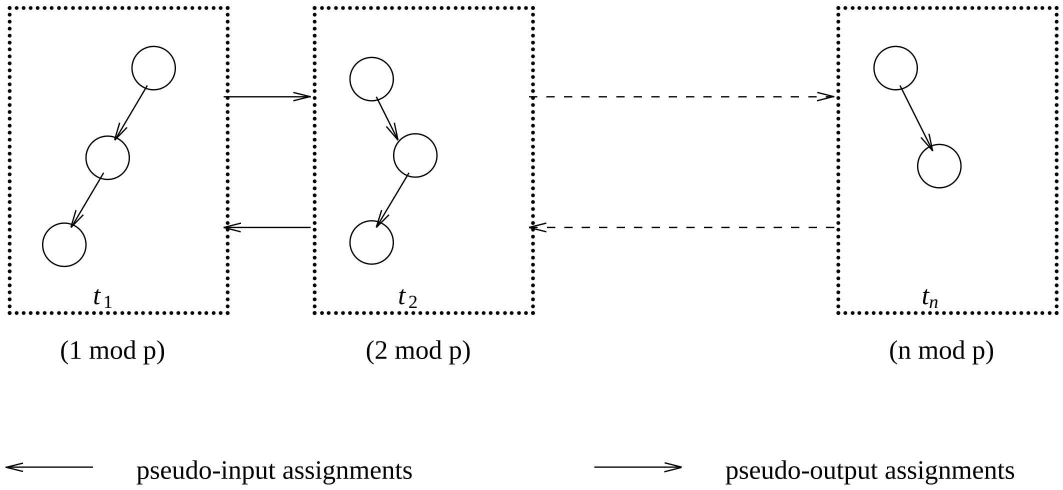


Figure 6.3. Distributed backtracking

should be able to initiate concurrent waves of activity as opposed to an assignment of time frames where a block of consecutive time frames are assigned to the same processor. Now given this particular assignment of time frames where time frame i is assigned to processor ($i \ mod \ p$), let us see how test generation can proceed in parallel. The pseudo-outputs and pseudo-inputs can be considered to be communication channels where signal requirements are exchanged. Now, during the test generation process, if an assignment is made on a pseudo-input in time frame i by a processor owning the time frame, the processor owning time frame $(i-1)$ tries to justify the required value at the corresponding pseudo-output in time frame $(i-1)$. Thus, activity in time frame i and $(i-1)$ can proceed concurrently. The activity in time frame i proceeds in an optimistic fashion, i.e., as if the values assigned at the pseudo-inputs have been realized. Similarly, justification objectives may be initiated due to assignments made on the pseudo-inputs in time frame $(i-1)$ giving rise to activity in time frame $(i-2)$. This is an example of concurrent waves of activity initiated backward in time. Concurrent activity can also occur forward in time during the FTP phase of test generation.

In cases where the required objectives cannot be met, a backtrack message has to be communicated which forces an alternative value to be assigned to a pseudo-input. Also, in the process of justifying the objective value at a particular pseudo-output in time frame i, additional values may be implied at other pseudo-outputs and this has to be communicated to the processor with time frame $(i+1)$, which can update the corresponding values on the pseudo-inputs in time frame $(i+1)$.

Consider Figure 6.3 again. The arrows between time frames in Figure 6.3 indicate the backward and forward flow of information. The forward flow of information (time frame t_i to t_{i+1}) is due to pseudo-output assignments occurring in time frame t_i and the backward flow of information (from t_i to t_{i-1}) is due to pseudo-input assignments occurring in time frame t_i. Thus, it can be seen that implementation of the distributed backtracking method is very complex and will need a very complicated distributed rollback and recovery schemes such as the one used in Time Warp [67]. An efficient termination detection algorithm will also be needed to evaluate whether all the assignments are consistent and whether there are no outstanding unjustified assignments. In most test generation algorithms, the testability measures are chosen such that assignments to pseudo-inputs are avoided as much as possible, unless the assignments are mandatory to realize some condition. This will result in large idle times across processors since the concurrent activity is initiated by assignments at the pseudo-inputs. The concurrency will also be limited by the inherently sequential waves of activity across time frames and high synchronization overheads. A complex distributed backtracking mechanism may be needed to make sure that the entire search space is explored [23]. The deficiencies of this approach are very similar to those encountered during the implementation of AND parallelism. Due to the above reasons, it was decided not to explore the method any further.

6.4.2. Parallel search

The effectiveness of the parallel search method for combinational circuits was demonstrated in [6]. We propose a similar method for sequential circuits. It was shown in [6] that only a small percentage of faults (less than 10%) required parallel search (these were referred to as hard-to-detect or HTD faults). The remaining faults were detected using very few backtracks. The reason for this behavior can be traced back to the use of inaccurate testability measures. It was shown in [21] that if a testability measure is able to find a test for a particular fault, it is able to do so with very few backtracks, otherwise, even though a test for a fault may exist, a large number of backtracks will be needed to steer out of the non-solution space into the solution space. Though the observations in [21] were based on combinational circuits, it can be shown that similar observations are valid for sequential circuits. Thus, the parallel branch and bound algorithm proposed in [6] can be extended to sequential circuits. However, since the size of the search space for sequential circuits is orders of magnitude higher than that for combinational circuits, more careful splitting of search space is required as outlined below.

Conceptually, the parallel branch and bound technique is shown in Figure 6.4. All processors cooperate with each other in generating a test (if one exists) for the *same* fault. The decision trees before and after search space splitting are shown. A node labeled ($x_n:i$) indicates that primary input (or pseudo-input during the RTP phase) x_n in time frame i has been assigned a value shown along the solid edge. The dashed edges indicate alternative values which have yet

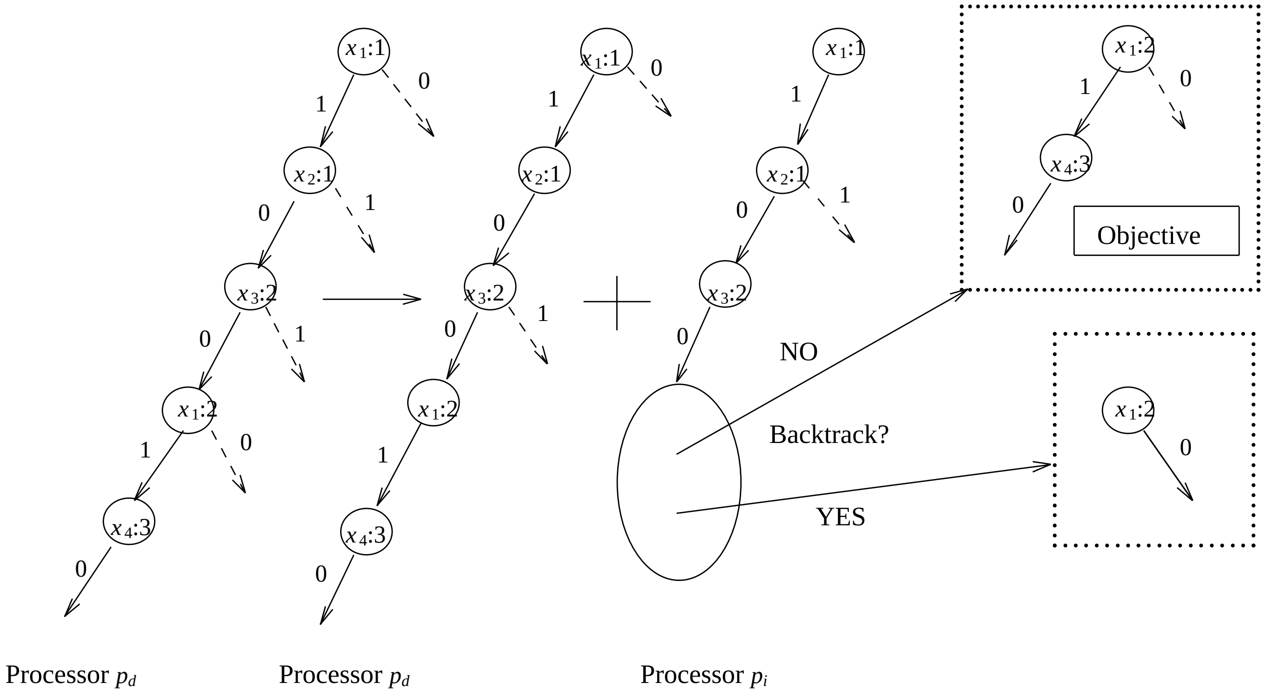


Figure 6.4. Search space allocation

to be assigned. An idle processor p_i requests work from a busy processor p_d. The processor p_d splits the search space between itself and p_i such that the number of choice points (or backtrack points) assigned to each processor is roughly equal. The treatment of the node at the leaf of the decision tree varies, depending on the test generation state of processor p_d. The splitting shown in Figure 6.4 may result in disjoint or overlapping search spaces depending on the test generation state of processor p_d. In the case of disjoint splitting, processor p_d is made to take a path disjoint from that of processor p_i by taking the alternative choice at the terminal node. The test generation state is indicated by a flag which indicates whether the last step executed by a processor was a backtrace or a backtrack.

If the last step executed was a backtrace, it means that the processor has reached a consistent set of assignments (locally) which may lead to a test. In such a case, forcing the processor p_i to take a disjoint path may lead to fruitless search since the assignments made by processor p_d may be mandatory. In such a case, processor p_d stores the last choice point encountered during backtrace. When the search space is split between p_i and p_d, the terminal node is not split. Instead, only the choice points are split equally between p_i and p_d. Processor p_i uses the choice point stored by p_d as an initial objective to direct the search in a different direction.

If the last step executed by p_d was a backtrack, then the search space is split in a disjoint fashion. The conditional splitting of the terminal node is shown in Figure 6.4. If the last step executed by processor p_d was a backtrack, then processor p_i takes an alternative path at node x_1:2. If the last step executed by p_d was not a backtrack, i.e., it was a backtrace, p_i takes the same path as p_d but with a temporary backtrace objective stored by p_d which helps it in searching in a different solution area. This strategy helps in reducing useless search in non-solution areas. However, it should be noted that choice points are split in both cases and the resultant search spaces allocated to both processors are in general smaller.

Unlike the implementation in [6] , the search space allocation takes place in a distributed fashion. This results in a better utilization of the available processors and prevents a single processor from becoming a bottleneck. Also, an idle processor p_i requests work from a processor p_d whose stack (or search space) has been split the least number of times. This ensures that p_i gets the largest chunk of work possible. Otherwise, p_i might exhaust the allocated search space and may need to request work repeatedly to remain busy which may increase the synchronization overhead and increase the idle time. During the parallel search step, only the dynamic information (circuit state etc.) need be replicated across all processors while the static circuit graph is shared by all processors thus reducing the memory overhead. The parallel search can take place under either a backtrack limit or a time limit. If a backtrack limit is used, each processor maintains its own backtrack limit counter in the shared memory. When the sum of backtracks across all processors exceeds the specified backtrack limit, test generation is aborted. Similarly, if a time limit is used, a shared *timeout* flag indicates whether the time limit is exceeded. All processors check the timeout flag periodically and also check their local clock. If a processor finds that the timeout flag has been set, it aborts test generation for the current fault. The timeout flag is set by the first processor which finds that its local clock has exceeded the time limit.

6.4.3. Fault parallelism

As mentioned earlier, a majority of the detectable faults in a logic circuit are easy-to-detect (i.e., can be detected with a very few backtracks). For such faults, the use of the parallel search method will yield very low speedups. To speed up test generation for easy-to-detect faults, a simple fault parallel approach would suffice. We use fault parallelism to filter out easy-to-detect faults so that the aborted hard-to-detect faults can serve as inputs to the parallel search phase. In the fault parallel phase, each processor picks up a fault for test generation from the fault list and tries to generate a test for the fault in parallel. The processors then synchronize at a barrier and all the test vectors generated for all the p faults are simulated using parallel fault simulation using the circuit partitioned approach given in Section 6.3. Thus, all the test vectors generated at the end of the test generation step are simulated by all processors in the same sequence. To ensure that the time taken for test generation for faults being processed in parallel are the same, the fault list is sorted according to the test length estimate given in Section 6.2.

The fault partitioning strategies detailed in [7, 62] and Chapter 4 were not used since the partitioning strategies were based on tracing out the region of influence of a fault effect. Due to the presence of feedback paths in a sequential circuit, it is very likely that a majority of lines are reachable by the fault effect. This would require inclusion of a large portion of the circuit in a partition and cause a skew in processor work loads. Also, it was shown in [7, 62] that the increase in test length is significant even for combinational circuits resulting in lower speedups. This increase in test length would be more marked for sequential circuits resulting in very low speedups.

6.4.4. Overall strategy

The overall parallel test generation strategy is as follows. First the network description is read in and converted to the internal data structures. The circuit is levelized and the SCOAP testability measures and test length estimates are calculated. At this stage, un-initializable faults are deleted from the fault list and marked as un-initializable. This is followed by a fault parallel test generation phase interleaved with parallel fault simulation using circuit partitioning. Since this phase is for easy-to-detect faults, a very low backtrack limit or a low time limit per fault is used. At the end of this phase (referred to as Phase 1 in the results), the aborted faults are collected which serve as inputs to Phase 2, which is the parallel search phase outlined in Section 6.4.2. Each processor cooperates in test generation for the same fault and if test generation is successful, parallel fault simulation is carried out using the circuit partitioned approach. We present results based on this parallel test generation strategy in the next section.

6.5. Experimental Results

The parallel test generation system for sequential circuits was implemented using the C programming language on an Encore Multimax shared-memory multiprocessing system. The SCOAP heuristic was used for test generation and a limit of 255 time frames per fault was used. Table 6.1 summarizes the TG/FS statistics using a single processor (which is a National Semiconductor NS 32532 processor rated at 8.5 MIPS). It can be seen that even the uniprocessor versions of the test pattern generator and fault simulator are comparable to implementations reported in the past [59, 61, 66]. For all the circuits, a dynamic (incremental) backtrack limit was used. However, it is still possible to improve the performance of the serial algorithm by using better uniprocessor heuristics, as demonstrated in [64]. The time required to calculate the testability measure (SCOAP) is not shown in Table 6.1. In most cases, it was found to be negligible compared to the overall TG/FS time.

The following is a brief description of the column headings in Table 6.1:

Circuit: The ISCAS benchmark circuit

Tgen: Test generation time in seconds

Fsim: Fault simulation time in seconds

Vec: Number of test vectors

Table 6.1. Uniprocessor results

Circuit	Tgen	Fsim	Vec	Len	Flts	Det	Red	UnInit	Abort	Cov	Eff
s208	42.6	9.5	160	18	215	136	44	0	35	63.26	83.72
s298	1996.8	17.7	208	33	308	262	18	0	28	85.06	90.91
s344	123.0	6.5	64	8	342	314	4	0	24	91.81	92.98
s349	41.2	6.3	61	8	350	310	6	0	34	88.57	90.29
s382	840.8	31.8	133	86	399	229	0	0	170	57.39	57.39
s386	1641.9	11.1	297	17	384	314	70	0	0	81.77	100.00
s400	848.7	35.1	137	86	421	230	6	0	185	54.63	56.06
s420	3787.2	18.3	138	22	430	177	163	0	90	41.16	79.07
s444	5659.3	150.0	864	114	474	412	16	0	46	86.91	90.30
s510	0.0	0.0	0	0	564	0	0	564	0	0.00	100.00
s526	3409.0	96.1	196	126	555	274	1	0	280	49.37	49.55
s526n	3342.0	98.3	196	126	553	274	0	0	279	49.55	49.55
s641	12.9	18.2	229	6	467	404	0	59	4	86.51	99.14
s713	57.4	20.3	209	8	581	476	16	67	22	81.93	96.21
s820	1811.6	20.0	93	5	850	321	0	0	529	37.77	37.77
s832	2010.0	21.1	91	5	870	312	14	0	544	35.86	37.47
s838	3291.3	47.5	145	22	857	253	52	0	552	29.52	35.59
s953	15.3	0.8	12	2	1079	87	0	987	5	8.06	99.53
s1196	28.9	44.6	352	4	1242	1239	3	0	0	99.76	100.00
s1238	58.5	54.0	388	3	1355	1283	71	0	1	94.69	99.93
s1423	2150.1	59.3	127	13	1515	609	6	0	900	40.20	40.59
s1488	5921.6	23.1	57	5	1486	597	0	0	889	40.18	40.18
s1494	6359.7	24.4	58	5	1506	573	10	0	923	38.05	38.71
s5378	4091.5	782.6	856	30	4603	3243	66	0	1294	70.45	71.89
s9234	0.3	1.2	3	1	6927	18	0	6909	0	0.26	100.00
s13207	6652.4	123.1	111	37	9815	579	8	8596	632	5.90	93.56
s15850	41.7	5.4	5	1	11725	85	0	11636	4	0.73	99.97
s35932	41547.0	4634.5	485	34	39094	34941	3856	0	297	89.38	99.24

Len: Maximum test sequence length

Flts: Size of the fault list

Det: Faults detected

Red: Faults proved redundant

UnInit: Number of un-initializable faults

Abort: Number of faults aborted

Cov: Fault coverage

Eff: Fault efficiency

The fault coverage and efficiency figures are calculated as follows:

$$Fault\ coverage = \frac{Faults\ detected}{Total\ faults}$$

$$Fault\ efficiency = \frac{Faults\ detected + redundant + un-initializable}{Total\ faults}$$

Table 6.2 shows the parallel fault simulation statistics using 1, 2, 4, 6 and 8 processors, respectively. It can be seen that speedups range from about 2 for the smaller circuits to more than 6 for the largest circuit (s35932) on 8 processors. The test vectors for this experiment were obtained from prior TG/FS runs on the uniprocessor. The fault coverage for the vector set is also shown.

Table 6.3 and 6.4 show sample results for parallel search for the hard-to-detect faults using some of the ISCAS circuits *without the use of fault simulation*. The faults used to evaluate the search method were the faults aborted for the corresponding circuit on the uniprocessor (i.e., after using a low time limit of 2 seconds). Table 6.3 shows the parallel search results when a fixed total backtrack limit (1000) was used. It was shown in [6] that speedup anomalies are possible leading to efficiencies greater than the number of processors (measured as the ratio of

Table 6.2. Parallel fault simulation on 8 processors

Circuit	Vec	Cov	Processors				
			1	2	4	6	8
s208	160	63.26	9.6	6.4	4.7	4.4	4.4
s298	197	83.77	16.0	10.0	6.4	5.6	5.4
s344	64	91.81	6.2	4.0	3.0	2.7	2.7
s349	61	88.57	6.1	4.1	2.9	2.7	2.7
s400	137	54.63	34.7	21.1	14.2	12.4	14.1
s420	138	41.16	18.0	12.0	9.4	8.9	10.9
s444	713	71.94	120.6	76.0	54.4	43.6	51.9
s526	196	49.40	92.4	58.6	36.7	30.9	27.9
s526n	196	49.55	91.4	58.0	36.7	30.9	28.6
s641	229	86.51	18.7	14.2	12.4	12.3	14.2
S713	209	81.93	21.3	15.8	13.3	13.1	15.0
s820	93	37.77	20.0	11.4	6.4	5.0	4.5
s832	91	35.86	19.5	11.4	6.5	5.0	4.3
s838	145	29.52	45.5	30.7	24.7	24.7	25.3
s1196	352	99.76	43.7	26.6	17.4	15.1	13.8
s1238	388	94.69	56.4	33.9	21.4	18.4	17.1
s1423	127	40.20	59.4	37.9	25.3	23.0	22.0
s1488	57	40.18	23.4	13.3	7.6	5.7	5.1
s1494	58	38.05	23.6	13.5	7.7	5.7	5.7
s5378	856	70.45	771.8	431.1	244.1	180.7	160.0
s13207	111	5.90	194.9	108.8	61.4	46.1	42.4
s15850	5	0.73	16.1	9.3	5.6	4.6	4.2
s35932	86	87.99	1118.4	593.2	311.3	219.6	177.1

Table 6.3. Parallel search with backtrack limit

Circuit	Faults	Processors									
		1		2		4		6		8	
		Eff	Time	Eff	Time	Eff	Time	Eff	Time	Eff	Time
s208	35	34.29	225.6	31.43	111.0	31.43	68.5	31.43	44.5	31.43	47.8
s298	42	40.48	1938.9	40.48	737.0	40.48	295.1	40.48	189.0	42.86	136.1
s344	24	4.17	218.0	8.33	102.0	4.17	56.0	20.83	33.8	16.67	29.0
s349	34	5.88	310.9	20.59	130.8	26.47	65.4	32.35	48.8	35.29	32.2
s641	4	0.00	49.5	0.00	28.6	0.00	15.9	0.00	11.0	0.00	9.3
s713	22	81.88	280.9	81.88	140.9	81.88	81.4	81.88	61.5	81.88	56.5
s953	5	0.00	105.3	0.00	53.2	0.00	28.5	0.00	19.8	0.00	16.0
s35932	297	43.10	11342.5	43.10	10281.1	43.10	9462.1	43.10	9193.9	43.10	9206.0

Table 6.4. Parallel search with time limit

Circuit	Faults	Processors									
		1		2		4		6		8	
		Eff	Time	Eff	Time	Eff	Time	Eff	Time	Eff	Time
s208	35	14.29	65.7	25.71	60.6	28.57	57.7	28.57	57.3	28.57	56.0
s298	42	26.19	74.8	30.95	69.0	35.71	66.5	38.10	65.5	40.48	63.4
s344	24	0.00	48.3	4.17	46.6	4.17	46.6	20.83	40.9	16.67	42.3
s349	34	0.00	68.4	20.59	57.78	26.47	53.9	35.29	47.8	38.23	46.3
s641	4	0.00	8.1	0.00	8.1	0.00	8.1	0.00	8.1	0.00	8.3
s713	22	13.64	43.1	27.27	39.4	31.82	37.3	40.91	36.8	40.91	34.9
s953	5	0.00	10.1	0.00	10.1	0.00	10.2	0.00	10.2	0.00	9.9
s35932	297	43.10	10059.2	43.10	9096.4	43.10	8594.3	43.10	8319.0	43.10	8235.5

speedup to number of processors used). In the case of sequential circuits, since the search space is very large, the probability of noticing such anomalies would be much lower than that for combinational circuits. However, it is possible to observe superlinear speedups in the case of circuit s298 in Table 6.3. Search anomalies can also lead to decrease in efficiency due to the non-deterministic nature of the parallel search algorithm. This can occur if the parallel algorithm fails to find a solution which was found by the uniprocessor algorithm. Restricting the parallel search to only hard-to-detect faults reduces the likelihood of this problem. However, it is possible to notice a slight deterioration in the quality of the solution in the case of circuit s208 as the number of processors is increased. However, such instances are rare and in most cases the fault efficiency is a non-decreasing function of the number of processors. Also, due to the size of the search space, much larger processing power than that provided by 8 processors may be needed. The speedup obtained for the largest ISCAS circuit considered (s35932) is 1.2 on 8 processors because of an unusually large number of flip-flops and primary inputs. The performance of the search algorithm does not depend on the size of the circuit but on the number of primary inputs and flip-flops in the circuit which determine the size of the search space. Table 6.4 shows the parallel search results when a time limit was used. The time limit was varied according to the size of the circuit (from 2 seconds per fault for s208 to 20 seconds per fault for s35932). The overall behavior is similar to that in Table 6.3 except that the apparent speedups are lower. Here, the speedups should not be calculated with respect to the uniprocessor since the actual number of backtracks and the total time are roughly p times those of the uniprocessor.

Table 6.5 shows the TG/FS statistics for the fault parallel phase (Phase 1) on 8 processors. The faults were sorted in a non-increasing order of the estimated test lengths. Due to the difference in fault ordering which occurs when tests for faults are generated in parallel, there is a marginal variation in fault coverages from those in Table 6.1. The maximum speedup obtained for test generation and fault simulation was more than 6 (for circuit s35932) during Phase 1. The faults dropped during Phase 1 were then used as inputs to the parallel search phase (Phase 2) of the test generation process.

Table 6.6 shows the results for Phase 2. In most cases, there is a significant increase in the overall coverage for hard-to-detect faults. For example, for circuit s444 more than 60 percent of the hard-to-detect faults were either detected or declared redundant.

Table 6.5. Phase 1 on 8 processors

Circuit	**Tgen**	**Fsim**	**Vec**	**Len**	**Flts**	**Det**	**Red**	**UnInit**	**Abort**	**Cov**	**Eff**
s208	10.1	6.6	204	26	215	137	44	0	34	63.72	84.19
s298	201.4	8.0	302	34	308	255	8	0	45	82.79	85.39
s344	33.6	2.4	54	8	342	312	4	0	26	91.22	92.40
s349	8.3	3.3	70	8	350	319	6	0	25	91.14	92.86
s382	495.1	21.4	340	86	399	236	0	0	163	59.15	59.15
s386	71.8	0.9	9	3	384	102	51	0	231	26.56	39.84
s400	489.8	21.5	340	86	421	245	6	0	170	58.20	59.62
s420	55.0	17.4	258	34	430	179	89	0	162	41.63	62.33
s444	602.8	41.8	654	114	474	341	13	0	120	71.94	74.68
s526	2044.0	52.7	461	126	555	275	1	0	279	49.55	49.73
s526n	2020.5	49.6	428	126	553	277	0	0	276	50.09	50.09
s641	2.6	8.9	167	9	467	404	0	59	4	86.51	99.14
s713	14.2	9.7	154	8	581	476	16	67	22	81.93	96.21
s820	270.0	3.5	83	5	850	309	0	0	541	36.35	36.35
s832	316.9	3.5	80	5	870	302	14	0	554	34.71	36.32
s838	170.2	31.2	190	26	857	254	65	0	538	29.64	37.22
s953	3.2	0.5	20	2	1079	87	0	987	5	8.06	99.54
s1196	6.5	15.7	408	4	1242	1239	3	0	0	99.76	100.00
s1238	22.7	19.4	429	4	1355	1283	71	0	1	94.69	99.93
s1423	261.7	16.9	100	10	1515	590	6	0	919	38.94	39.34
s1488	887.0	4.3	44	5	1486	646	0	0	840	43.47	43.47
s1494	915.7	4.3	43	5	1506	661	10	0	835	43.89	44.56
s5378	1976.4	215.9	1013	30	4603	3284	66	0	1253	71.35	72.78
s13207	853.2	55.9	222	37	9815	594	8	8596	617	6.05	93.71
s15850	11.0	4.3	24	3	11725	85	0	11636	4	0.73	99.97
s35932	6136.5	952.9	616	35	39094	34929	3822	0	343	89.35	99.12

Table 6.6. Phase 2 on 8 processors

Circuit	Tgen	Fsim	Vec	Len	Flts	Det	Red	Abort	Cov	Eff
s208	53.9	0.0	0	0	34	0	10	24	0.00	29.41
s298	64.9	2.2	33	12	45	9	12	24	20.00	46.67
s344	770.0	1.5	41	8	26	17	3	6	65.39	76.92
s349	196.2	1.8	50	8	25	15	1	9	60.00	64.00
s382	327.9	0.0	0	0	163	0	0	163	0.00	0.00
s386	541.4	4.6	323	17	231	212	19	0	91.78	100.00
s400	342.1	0.0	0	0	170	0	0	170	0.00	0.00
s420	2143.3	0.0	0	0	162	0	86	76	0.00	53.09
s444	835.4	10.0	230	53	120	72	3	45	60.00	62.50
s526	457.6	13.5	126	45	279	60	0	219	21.51	21.51
s526n	448.1	10.2	90	45	276	60	0	216	21.74	21.74
s641	7.8	0.0	0	0	4	0	0	4	0.00	0.00
s713	35.2	0.0	0	0	22	0	9	13	0.00	40.91
s820	898.2	15.7	98	8	541	129	0	412	23.85	23.85
s832	910.4	13.3	89	8	554	131	0	423	23.65	23.65
s838	1080.3	0.0	0	0	538	0	16	522	3.00	3.00
s953	10.0	0.0	0	0	5	0	0	5	0.00	0.00
s1238	0.9	0.0	0	0	1	0	1	0	100.00	100.00
s1423	1851.7	17.9	49	13	919	47	0	872	5.11	5.11
s1488	1475.4	12.6	44	6	840	149	0	691	17.74	17.74
s1494	1629.2	7.2	27	5	835	57	0	778	6.83	6.83
s5378	12037.7	22.0	42	14	1253	61	19	1173	6.39	6.39
s13207	7312.4	15.7	19	19	617	14	0	603	2.30	2.30
s15850	39.4	0.0	0	0	4	0	0	4	0.00	0.00
s35932	5601.9	0.0	0	0	343	0	162	181	47.23	47.23

Table 6.7 shows the combined results of Tables 6.5 and 6.6. It can be seen that the overall results on 8 processors are significantly better both in terms of the overall run time as well as the efficiency. For example consider circuit s298. It took 2014.5 seconds on the uniprocessor to obtain an efficiency of 90.91 percent. However, on the multiprocessor, it took only 276.5

Table 6.7. Combined results of Phases 1 and 2 on 8 processors

Circuit	Tgen	Fsim	Vec	Len	Flts	Det	Red	UnInit	Abort	Cov	Eff
s208	64.0	6.6	204	26	215	137	54	0	24	63.72	88.84
s298	266.3	10.2	335	34	308	264	20	0	24	85.71	92.21
s344	803.6	3.9	95	8	342	329	7	0	6	96.20	98.24
s349	204.5	5.1	120	8	350	334	7	0	9	95.43	97.42
s382	823.0	21.4	340	86	399	236	0	0	163	59.15	59.15
s386	613.2	5.5	332	17	384	314	70	0	0	81.77	100.00
s400	831.9	21.5	340	86	421	245	6	0	170	58.19	59.62
s420	2198.3	17.4	258	34	430	179	175	0	76	41.63	82.32
s444	1438.2	51.8	884	114	474	413	16	0	45	87.13	90.51
s526	2501.6	66.2	587	126	555	335	1	0	219	60.36	60.54
s526n	2468.6	59.8	518	126	553	337	0	0	216	60.94	60.94
s641	10.4	8.9	167	9	467	404	0	59	4	86.51	99.14
s713	49.4	9.7	154	8	581	476	25	67	13	81.93	97.76
s820	1168.2	19.2	181	8	850	438	0	0	412	51.53	51.53
s832	1227.3	16.8	169	8	870	433	14	0	423	49.77	51.38
s838	1250.5	31.2	190	26	857	254	81	0	522	29.64	39.09
s953	13.2	0.5	20	2	1079	87	0	987	5	8.06	99.54
s1196	6.5	15.7	408	4	1242	1239	3	0	0	99.76	100.00
s1238	23.6	19.4	429	4	1355	1283	72	0	0	94.69	100.00
s1423	2113.4	34.8	149	13	1515	637	6	0	872	42.05	42.44
s1488	2362.4	16.9	88	6	1486	795	0	0	691	53.50	53.50
s1494	2544.9	11.5	70	5	1506	718	10	0	778	47.68	48.34
s5378	14014.1	237.9	1055	30	4603	3345	85	0	1173	72.67	74.52
s13207	8165.6	71.6	241	37	9815	608	8	8596	603	6.19	93.86
s15850	50.4	4.3	24	3	11725	85	0	11636	4	0.72	99.97
s35932	11738.4	952.9	616	35	39094	34929	3984	0	181	89.35	99.54

seconds to obtain an efficiency of 92.21 percent. Thus, from the results presented in this section, it is clear that parallel processing is a very cost-effective method to improve the efficiency as well as speed of sequential circuit test generation algorithms.

6.6. Summary

We presented parallel algorithms for test generation and fault simulation to utilize the increased computing power provided by general purpose multiprocessors. A circuit partitioned approach to parallel fault simulation was presented which results in high concurrency at the cost of negligible memory overhead. Also, a fault parallel approach to accelerate test generation for easy-to-detect faults was proposed. A simple test length estimate was used to estimate the time required to generate a test for a fault. To accelerate test generation for faults left after filtering out easy-to-detect faults, a parallel search method for sequential circuits was proposed and evaluated. Finally, a highly efficient parallel test generation system for synchronous sequential circuits incorporating the above techniques was proposed and evaluated using the ISCAS benchmark circuits on an Encore Multimax shared-memory multiprocessor. It was demonstrated that parallel processing can yield significant improvements in the speed and efficiency of sequential circuit test generation algorithms.

CHAPTER 7.

CONCLUSIONS

7.1. Summary

With the increase in complexity of digital circuits, it has become extremely important to verify correct operation of a digital circuit. Test generation algorithms have exponential worst-case time complexity while the time complexity of fault simulation algorithms is quadratic. Since test generation and fault simulation for circuits of VLSI complexity can take a prohibitive amount of time, speeding up test generation and fault simulation algorithms by either using better uniprocessor heuristics or by using the tremendous computing power available on multiprocessors becomes important. With general-purpose multiprocessors becoming more common and affordable, accelerating test generation and fault simulation using parallel processing becomes a very attractive and cost-effective alternative. The design and analysis of parallel algorithms for test generation and fault simulation were the focus of this thesis research. This dissertation is a part of a much larger project called HIPERCAD, whose objective is to design efficient parallel algorithms for a variety of Computer-Aided Design (CAD) problems.

We categorized various parallel processing techniques available for test generation and fault simulation. We discussed the advantages and disadvantages of each of the proposed techniques. We proposed a parallel search method to overcome the deficiencies of inaccurate search heuristics. It was shown that the parallel search method scales very easily with the number of processors and can be implemented easily on configurations with a large number of processors.

We demonstrated that this method not only results in faster execution of the test generation algorithm but also results in a better quality of the solution.

We proposed fault partitioning techniques to speed up test generation for faults which are relatively easy-to-detect. The objective of the fault partitioning techniques is to maximize concurrency without affecting the quality of the overall solution. We proposed load balancing techniques which try to minimize the processor idle time with very low communication overhead.

We proposed a performance model which takes into account the various trade-offs in exploiting parallelism in a test generation/ fault simulation environment. Using the performance model, it is possible to predict the speedup on a multiprocessor based on statistics collected on the uniprocessor.

Finally, we presented a parallel test generation system for sequential circuits. A parallel search technique was used to accelerate test generation for hard-to-detect faults and a circuit partitioned approach was used to accelerate fault simulation. It was shown that the parallel test generation system is comparable in efficiency as well as performance to most existing test pattern generators.

7.2. Future Research

As a part of future research, it would be interesting to see whether any of the techniques presented in this thesis can be applied to other CAD problems. For example, we can transform the logic verification problem into the problem of test generation and use the parallel search techniques proposed in this thesis. Since the fault partitioning techniques presented in this thesis are not exhaustive, it may be possible to devise more efficient partitioning techniques. We considered fault partitioning based on fault compatibility. It may be possible to partition faults based on fault independence. Also, parallel algorithms for other CAD problems such as VLSI layout compaction, Boolean verification using binary decsion diagrams and logic synthesis need to be investigated.

The parallel algorithms presented in this thesis are independent of the MIMD multiprocessor architecture. It is necessary to evaluate the performance of the parallel algorithms on other architectures, specially in a loosely-coupled environment such as a distributed network of workstations. It is our belief that as long as a proper communication to computation ratio is maintained, it should be possible to achieve performance close to or better than that documented in this thesis. We also gave an intuitive reasoning as to why the distributed backtracking and AND parallel approaches will lead to low speedups. This has to be evaluated experimentally.

The purpose of this thesis is also not to detract from the use of better uniprocessor heuristics. The performance of the parallel algorithm may be limited by the efficiency of the serial algorithm. Thus a two-pronged approach has to be used to accelerate CAD algorithms: research in the areas of better uniprocessor algorithms as well as parallel algorithms has to proceed concurrently. It may be possible to incorporate advances in uniprocessor algorithms into existing parallel algorithms to get even better speedups. Also, it might be possible to apply the insight gained through investigation of parallel algorithms to accelerate existing serial algorithms. Thus, parallel and serial algorithm research go hand in hand and should not be considered as two separate disciplines.

APPENDIX A.

HIPERTEST

NAME

HIPERTEST - a parallel test pattern generation system for combinational circuits.

SYNOPSIS

tgen [-option] [< *circuit_file*] [> *output_file*]

DESCRIPTION

The **tgen** program is the parallel test pattern generator for combinational circuits referred to as **HIPERTEST** in this thesis. The program accepts input from the standard input and directs all its output to the standard output. Optionally, the input or output may be redirected to read or write to a file. The program runs on an Intel *i*PSC/2 hypercube multiprocessor and the node programs are loaded from within the host program (invoked as **tgen**). The number of processors to be used is specified by using the *getcube* command on the *i*PSC/2 host. The input to the program is the netlist description with an embedded fault list. Details about the netlist description may be found in Appendix C.

The basic test generation algorithm used is PODEM and the fault simulation algorithm used is deductive fault simulation. The program processes the fault list in two phases. In the first phase (Phase 1), faults are partitioned across processors and each processor carries out test generation and fault simulation independently. The static fault partitioning strategy can be specified as a command line option. There is also a dynamic load balancing mechanism which redistributes the fault list as the processors become idle. The degree of communication among processors for test vectors can be varied by changing the *communication cutoff factor*.

The faults dropped after executing Phase 1 are used as inputs to drive Phase 2 of the program which employs parallel search to generate a test for each fault. If the test generation for a fault is successful, fault simulation is again used. The backtrack limit used in Phase 2 is the same as that used in Phase 1 but on a per processor basis. Thus, if b is the backtrack limit during Phase 1 and n is the number of processors, the total backtrack limit per fault is $n \times b$ in Phase 2.

OPTIONS

-s Use the SCOAP heuristic for guidance (default: none).

-c Use COP.

-m Use CAMELOT.

-d Use FANOUT/DISTANCE.

-r Use RANDOM.

-b*n* Set backtrack limit to n (default: 25).

-f Use random fill-in for primary inputs which are unspecified (useful during fault simulation).

-p*n* Set the type of static partitioning specified by n. If n is 0 use random partitioning, if 3 use input partitioning, if 4 use output partitioning and if 7 use compatible fault sets (MACP) (default: none).

-u Use redundancy identification before entering Phase 2.

-v*n* Use a communication cutoff factor of n. Legal values of n are between 0 and 100. Thus, n specifies the fault coverage in percent after which communication for test vectors is stopped (default: 0).

EXAMPLE

To run the test pattern generator on circuit *c432* using the SCOAP heuristic, random fill-in, input partitioning and a communication cutoff of 65:

tgen *-s -f -p3 -v65 < c432 > c432.out*

APPENDIX B.

STGEN

NAME

STGEN - a parallel test pattern generation system for sequential and combinational circuits.

SYNOPSIS

stgen *circuit_file* [-options]

fsim *circuit_file* [-options]

DESCRIPTION

stgen is a parallel sequential circuit test pattern generator which uses the PODEM type approach and can also handle the combinational circuits. The program has been designed to run on a shared-memory Encore Multimax multiprocessor. The netlist format is the same as that used by HIPERTEST with the addition of an extra primitive to handle memory elements (D-type flip-flop). Refer to Appendix C for details about the circuit format. Muth's nine-valued logic is used to avoid overspecification of logic values which may make state justification difficult. A forward time processing algorithm (FTP) is used for fault propagation and a reverse time processing (RTP) algorithm is used for state justification. The FTP phase is responsible for consuming most of the memory during test generation. However, the memory used by the RTP phase does not vary much with the number of time frames used since only one time frame at a time is used for state justification. In the event of a backtrack to a previous time frame during the RTP phase, the state is restored from the primary input and pseudo input lines stored on the stack.

A modified form of the SCOAP testability measure is used and the test pattern generator can also give an estimate of the maximum test sequence length. Since the SCOAP measure is calculated iteratively, it may be necessary to adjust the MAXITER constant in the file "heur.c" if a circuit takes too much time for testability measure calculation. The test pattern generator has the capability of using the previous state (resulting from the test sequence for the previous fault) to generate a test for the next fault. This can produce a significant decrease in test lengths and run times for some of the ISCAS circuits. The test pattern generator is also able to generate a test length estimate which is useful in determining the memory requirements and the difficulty in generating a test for a particular fault.

The test pattern generator also uses the deductive fault simulation technique for fault simulation. Due to the use of efficient memory retrieval techniques, it is possible to simulate even the largest ISCAS benchmark circuit (about 40,000 gates) without the need to partition the fault list. Though the fault simulator is designed to be used in conjunction with the test pattern generator, it is possible to use the fault simulator alone (use **fsim**).

The program processes the fault list in two phases. In Phase 1, a fault partitioned approach is used where each processor picks up a fault to be processed from a shared fault list. The faults in the fault list are ordered in non-increasing order of the estimated test lengths. The processors synchronize after generating a test and use a circuit partitioned approach for parallel fault simulation. In Phase 2, parallel search is performed over faults dropped by Phase 1. Phase 2 also incorporates parallel fault simulation using circuit partitioning.

INPUT AND OUTPUT FILES

Given a netlist description in the file *circuit_file,* the program expects other inputs in the files given below. Also, the output is written to files which are created by adding simple extensions to the file name for the netlist (denoted here as *circuit_file*).

circuit_file

Input file containing the netlist description.

circuit_file.fsim

Output file for fault simulation trace.

circuit_file.atpg

Output file for test generation trace.

circuit_file.res

Output file with result summary.

circuit_file.vec

Output file with test vectors. Also, input file for test vectors if only fault simulation is used (by invoking **fsim**).

circuit_file.flts

An optional output file generated if **stgen** is asked to dump the aborted faults. Also, an optional input file in case **stgen** is asked to ignore the fault list embedded in *circuit_file* and use the faults specified in *circuit_file.flts* instead.

OPTIONS

Invoking **stgen** without any other command line arguments gives the list of options given below.

-s Use the SCOAP heuristic for test generation (default: none).

-d Use the FANOUT/DISTANCE heuristic.

-r Use heuristic RANDOM.

-f Use fault simulation with random fill-in for unspecified primary inputs.

-p Try to use previous state information. Works only if **-f** options is used.

-b*n* Set backtrack limit to *n* (default: 25).

-t*n* Set time limit to *n* seconds (default: if the backtrack limit is set, the backtrack limit is used else a backtrack limit of 25 is used). If both time limit and backtrack limit are set, then the time limit overrides the backtrack limit.

-w Dump all aborted faults to *circuit_file.flts*.

-a Do not use fault list embedded in the circuit description. Instead, read fault list from *circuit_file.flts*.

-g use random pattern generation before starting deterministic test pattern generation.

-n*n* Use *n* processes for parallel test pattern generation and fault simulation (default: 1).

EXAMPLES

To run the test pattern generator using 8 processes on the circuit *s208* using fault simulation, SCOAP heuristic, the previous state information and a time limit of 2 seconds:

stgen *s208 -sfp -n8 -t2*

The results will be written to the file *s208.res*, the fault simulation and test generation traces are written to the files *s208.fsim* and *s208.atpg*, respectively, and the test vectors will be written to the file *s208.vec*.

To run the fault simulator alone on circuit *s208* for vectors stored in *s208.vec* using 8 processors:

fsim *s208 -n8*

The results will be written to the file *s208.res* and a fault simulation trace will be written to *s208.fsim*. If *s208.vec* is non-existent, the fault simulator will prompt for the number of the random vectors to be used.

APPENDIX C.

TRANS

NAME

TRANS - a translator for the *ISCAS89* circuits.

SYNOPSIS

trans *circuit_file*

DESCRIPTION

Trans transforms the circuits in the *ISCAS89* format to one which is compatible with the input for **stgen**. The circuit format for **stgen** also includes an embedded fault list which is a simplistically reduced equivalent fault set. The original *ISCAS89* files contain two types of files: the **.bench* files which contain the netlist description and the **.flts* files which contain the simplistically collapsed fault list for each of the *ISCAS* circuits. The program **trans**, however, does not require the **.flts* files and generates the fault list using **.bench* files. The algorithm used to generate the fault list is exactly the same as that used to generate the **.flts* files so the resultant fault lists are identical.

CIRCUIT FORMAT

The following is a grammar for the circuit description. The symbols indicated in bold are terminal symbols.

```
network   :  comment
          |  network   comment
          |  network   gate

gate      :  gtype     fanout    fanin   fltlist   inlist

fltlist   :  empty
          |  fltlist   fault

inlist    :  empty
          |  inlist    input

gtype     :  inpt
          |  dff
          |  fan
          |  and
          |  or
          |  xor
          |  buff
          |  nand
          |  nor
          |  xnor
          |  buff

fault     :  >sa0
          |  >sa1
```

Each gate is assigned an implicit gate number which is equal to the order in which the gate appears in the circuit description. Each gate is referred to by a number and no alphanumeric symbols are used. Everything beginning with an asterisk ('*') and ending in a newline is taken as a comment and is ignored. Optionally, each gate description may be followed by a comment giving the alphanumeric label of the gate in the original **.bench* files in the original *ISCAS89* format. A primary output node is one with a fanout 0 (zero). The program **stgen** expects at least one comment at the beginning of the circuit file. The symbol **`>sa0`** (**`>sa1`**) in the gate description indicates that a test for the gate output stuck-at-0 (stuck-at-1) has to be generated. Thus, the fault list is embedded in the circuit description.

EXAMPLE

To convert the circuit *s27.bench* to the format for **stgen** just type:

trans *s27*

The circuit is shown in Figure C.1 and the circuit description before and after translation by **trans** are shown in Tables C.1 and C.2, respectively. The line numbers indicate the gate numbers in the new circuit format.

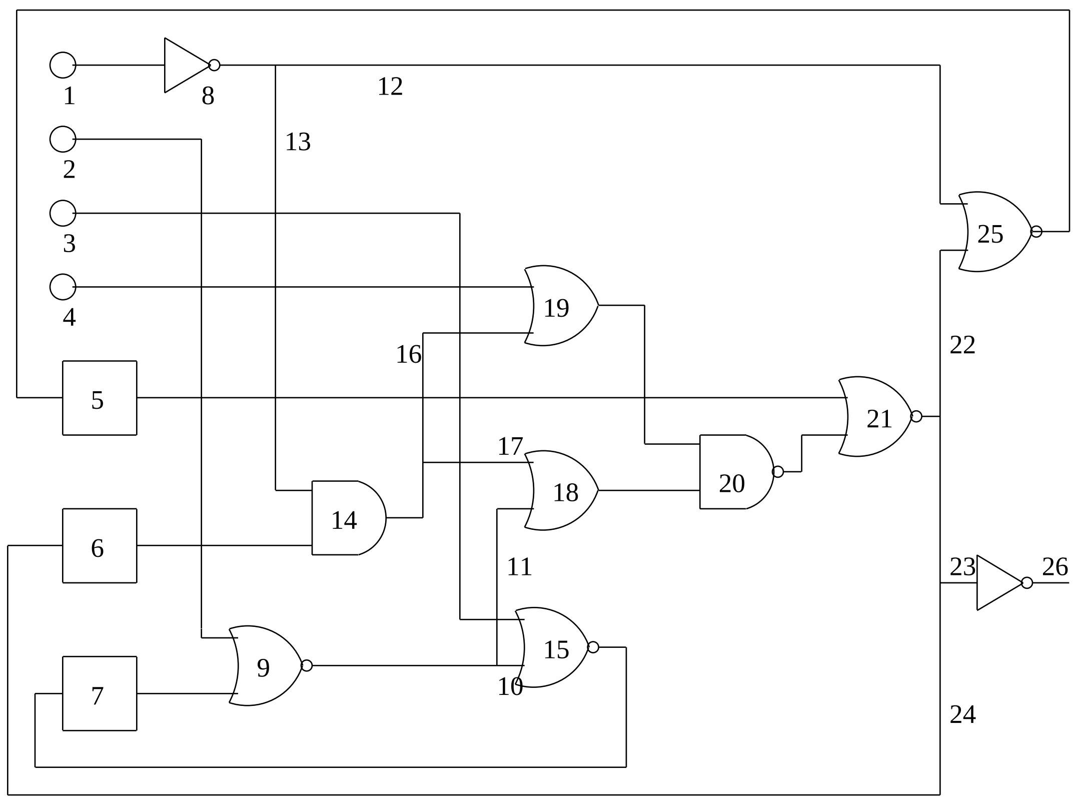


Figure C.1. Circuit s27

Table C.1. ISCAS89 description for s27

```
# 4 inputs
# 1 outputs
# 3 D-type flipflops
# 2 inverters
# 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)

INPUT(G0)
INPUT(G1)
INPUT(G2)
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)

G15 = OR(G12, G8)
G16 = OR(G3, G8)

G9 = NAND(G16, G15)

G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

Table C.2. Description for s27 after translation

```
* 4 inputs
* 1 outputs
* 3 D-type flipflops
* 2 inverters
* 8 gates (1 ANDs + 1 NANDs + 2 ORs + 4 NORs)
* Maximum fanout: 3
* Maximum fanin: 2
* Number of faults: 32

inpt 1 0 * G0
inpt 1 0 >sa0 * G1
inpt 1 0 >sa0 * G2
inpt 1 0 >sa0 * G3

dff 1 1 >sa0 25 * G5
dff 1 1 >sa1 24 * G6
dff 1 1 >sa0 15 * G7

not 2 1 >sa0 >sa1 1 * G14
nor 2 2 >sa0 >sa1 7 2 * G12
fan 1 1 >sa0 9 *
fan 1 1 >sa0 9 *
fan 1 1 >sa0 8 *
fan 1 1 >sa1 8 *
and 2 2 >sa0 >sa1 6 13 * G8
nor 1 2 >sa0 >sa1 10 3 * G13
fan 1 1 >sa0 14 *
fan 1 1 >sa0 14 *
or 1 2 >sa1 17 11 * G15
or 1 2 >sa1 16 4 * G16
nand 1 2 >sa0 18 19 * G9
nor 3 2 >sa0 >sa1 20 5 * G11
fan 1 1 >sa0 21 *
fan 1 1 21 *
fan 1 1 >sa0 >sa1 21 *
nor 1 2 >sa0 >sa1 22 12 * G10
not 0 1 >sa0 >sa1 23 * G17
```

REFERENCES

[1] K. T. Tam, ‘‘Parallel processing for CAD applications,’’ *IEEE Design and Test*, pp. 13-17, October 1987.

[2] P. Banerjee and M. Jones, ‘‘A parallel simulated annealing algorithm for standard cell placement on a hypercube computer,’’ *Proc. Int. Conf. Computer-Aided Design*, pp. 34-37, November 1986.

[3] J. Sargent and P. Banerjee, ‘‘A parallel row-based algorithm for standard cell placement with integrated error control,’’ in *Proc. 26th Design Automation Conf.*, Las Vegas, NV, June 1989.

[4] R. Brouwer and P. Banerjee, ‘‘A parallel simulated annealing algorithm for channel routing on a hypercube multiprocessor,’’ *Proc. Int. Conf. Computer Design*, pp. 4-7, October 1988.

[5] K. P. Belkhale and P. Banerjee, ‘‘PACE2: An improved parallel VLSI extractor with parametric extraction,’’ *Proc. Int. Conf. Computer-Aided Design*, November 1989.

[6] S. Patil and P. Banerjee, ‘‘A parallel branch and bound algorithm for test generation,’’ *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 313-322, March 1990.

[7] S. Patil and P. Banerjee, ‘‘Fault partitioning issues in an integrated parallel test generation/fault simulation environment,’’ in *Proc. Int. Test Conf.*, Washington D.C., pp. 718-726, August 1989.

[8] W. D. Hillis, in *The Connection Machine*. Cambridge, MA: MIT Press, 1986.

[9] G. F. Pfister, ‘‘The Yorktown Simulation Engine: Introduction,’’ *Proc. 19th Design Automation Conf.*, pp. 51-54, 1982.

[10] O. H. Ibarra and S. K. Sahni, ‘‘Polynomially complete fault detection problems,’’ *IEEE Trans. Computers*, vol. C-31, pp. 555-560, June 1982.

[11] 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.

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

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

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

[15] V. D. Agrawal, K. T. Cheng, and P. Agrawal, ‘‘CONTEST: A concurrent test generator for sequential circuits,’’ *Proc. 25th Design Automation Conf.*, 1988.

[16] D. B. Armstrong, ‘‘A deductive method for simulating faults in logic circuits,’’ *IEEE Trans. Computers*, vol. C-21, pp. 462-471, May 1972.

[17] E. G. Ulrich and T. Baker, ‘‘Concurrent simulation of nearly identical digital networks,’’ *Computer*, vol. 7, pp. 39-44, April 1974.

[18] S. Seshu, ‘‘On an improved diagnosis program,’’ *IEEE Trans. Electronic Computing*, vol. EC-14, pp. 76-79, 1965.

[19] D. Harel and B. Krishnamurthy, ‘‘On the complexity of fault simulation,’’ Technical Report No. CR-86-61, Tektronix Laboratories, January 1987.

[20] 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.

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

[22] 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.

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

[24] 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.

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

[26] L. Soule and T. Blank, ‘‘Parallel logic simulation on general purpose machines,’’ *Proc. 25th Design Automation Conf.*, pp. 166-171, 1988.

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

[28] 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.

[29] H. Fujiwara and T. Inoue, ‘‘Optimal granularity of test generation in a distributed system,’’ *IEEE Trans. Computer-Aided Design*, vol. 9, pp. 885-892, August 1990.

[30] 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.

[31] S. Arvindam, V. Kumar, V. N. Rao, and V. Singh, ‘‘Automatic test pattern generation on multiprocessors,’’ *Proc. Int. Conf. Knowledge-Based Computer Systems*, December 1989.

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

[33] D. L. Ostapko and Z. Barzilai, ‘‘Fast fault simulation in a parallel processing environment,’’ *Proc. Int. Test Conf.*, pp. 293-298, 1987.

[34] P. A. Duba, R. K. Roy, J. A. Abraham, and W. A. Rogers, ‘‘Fault simulation in a distributed environment,’’ *Proc. 25th Design Automation Conf.*, pp. 686-691, June 1988.

[35] F. Ozguner, C. Aykanat, and O. Khalid, ‘‘Logic fault simulation on a vector hypercube multiprocessor,’’ *Proc. 3rd Int. Conf. Hypercube and Concurrent Computers and Applications*, vol. II, pp. 1108-1116, January 1988.

[36] V. Narayanan and V. Pitchumani, ‘‘A parallel algorithm for fault simulation on the connection machine,’’ *Proc. Int. Test Conf.*, pp. 89-93, September 1988.

[37] R. B. Mueller-Thuns, D. G. Saab, R. F. Damiano, and J. A. Abraham, ‘‘Portable parallel logic and fault simulation,’’ in *Proc. Int. Conf. Computer-Aided Design*, Santa Clara, CA, pp. 506-509, November 1989.

[38] 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.

[39] V. K. Janakiram, D. P. Agrawal, and R. Mehrotra, ‘‘A randomized parallel branch-and-bound algorithm,’’ *Proc. Int. Conf. Parallel Processing*, pp. 69-75, 1988.

[40] 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.

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

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

[43] 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.

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

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

[46] 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.

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

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

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

[50] 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.

[51] J. A. Bondy and U. S. R. Murty, in *Graph Theory With Applications*. New York: North-Holland, 1976.

[52] D. Harel, ‘‘A linear time algorithm for finding dominators in flow graphs and related problems,’’ *Proc. 17th ACM Symp. Computing*, pp. 183-184, May 1985.

[53] P. Goel, ‘‘Test generation cost analysis and projections,’’ *Proc. 17th Design Automation Conf.*, pp. 77-84, June 1980.

[54] K. Li and R. Schaefer, ‘‘A hypercube shared virtual memory system,’’ *Proc. Int. Conf. Parallel Processing*, vol. I, pp. 125-132, August 1989.

[55] R. Marlett, ‘‘EBT: A comprehensive test generation technique for highly sequential circuits,’’ *Proc. 15th Design Automation Conf.*, pp. 332-338, June 1978.

[56] S. Mallela and S. Wu, ‘‘A sequential circuit test generation system,’’ *Proc. Int. Test Conf.*, pp. 57-61, October 1985.

[57] R. Marlett, ‘‘An effective test generation system for sequential circuits,’’ *Proc. 23rd Design Automation Conf.*, pp. 250-256, 1986.

[58] H. K. T. Ma, S. Devadas, A. R. Newton, and A. Sangiovanni-Vincentelli, ‘‘Test generation for sequential circuits,’’ *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 1081-1093, October 1988.

[59] W. T. Cheng, ‘‘The Back algorithm for sequential test generation,’’ *Proc. Int. Conf. Computer Design*, pp. 66-69, 1988.

[60] A. Ghosh, S. Devadas, and A. R. Newton, ‘‘Test generation for highly sequential circuits,’’ *Proc. Int. Conf. on Computer-Aided Design*, pp. 362-365, November 1989.

[61] T. P. Kelsey and K. K. Saluja, ‘‘Fast test generation for sequential circuits,’’ *Proc. Int. Conf. Computer-Aided Design*, pp. 354-357, November 1989.

[62] S. Patil and P. Banerjee, ‘‘Performance trade-offs in a parallel test generation/ fault simulation environment,’’ *IEEE Trans. Computer-Aided Design*, submitted for publication.

[63] F. Brglez, D. Bryan, and K. Kozminski, ‘‘Combinational profiles of sequential benchmark circuits,’’ *Proc. Int. Symp. Circuits and Systems*, pp. 1929-1934, May 1989.

[64] T. Niermann and J. H. Patel, ‘‘HITEC/PROOFS: A test generation and fault simulation system,’’ in *Proc. TECHCON 90*, San Jose, CA, October 1990.

[65] P. Muth, ‘‘A nine-valued circuit model for test generation,’’ *IEEE Trans. Computers*, vol. C-25, pp. 630-636, June 1976.

[66] M. H. Schulz and E. Auth, ‘‘ESSENTIAL: An efficient self-learning test pattern generation algorithm for sequential circuits,’’ in *Proc. Int. Test Conf.*, Washington D.C., pp. 28-37, August 1989.

[67] D. Jefferson, ‘‘Virtual time,’’ *ACM Transactions of Programming Languages*, vol. 7, pp. 404-425, July 1985.

[68] K. M. Chandy and J. Misra, ‘‘Asynchronous distributed simulation via a sequence of parallel computations,’’ *Communications of the ACM*, pp. 198-206, April 1981.

[69] S. Patil, P. Banerjee, and C. D. Polychronopoulos, ‘‘Efficient circuit partitioning algorithms for parallel logic simulation,’’ in *Proc. Supercomputing Conf.*, Reno, NV, pp. 361-370, November 1989.

VITA

Srinivas Patil was born in Hubli, India, on December 22, 1963. He received his B.Tech. degree in Electrical Engineering from the Indian Institute of Technology, New Delhi, India, in 1985. He obtained his M.S. degree in Electrical Engineering from the University of Iowa, Iowa City, Iowa, in 1987. He was a research assistant at the University of Iowa with the Department of Electrical and Computer Engineering from 1985 to 1986. At the University of Illinois, he was employed as a research assistant with the Center for Reliable and High-Performance Computing in the Coordinated Science Laboratory from 1987 to 1990. For the summer of 1989, he was employed by the IBM T. J. Watson Research Center, Yorktown Heights, New York.

After completing his doctoral dissertation, he will be joining IBM Corporation, Poughkeepsie, New York. His research interests include Testing and Simulation, Computer-Aided Design and Parallel Processing.

PARALLEL ALGORITHMS FOR TEST GENERATION AND FAULT SIMULATION

Srinivas Patil, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois, Urbana-Champaign, 1991

With increase in complexity of digital circuits, it has become extremely important to detect faults to ensure correct operation of a digital circuit. Since test generation and fault simulation for circuits of VLSI complexity can take a prohibitive amount of time, speeding up test generation and fault simulation algorithms by either using better uniprocessor heuristics or by using the tremendous computing power available on multiprocessors thus becomes important. With general-purpose multiprocessors becoming more common and affordable, accelerating test generation and fault simulation using parallel processing becomes a very attractive and cost-effective alternative. The design and analysis of parallel algorithms for test generation and fault simulation are the focus of this thesis research. This dissertation is a part of a much larger project called the HIPERCAD project whose objective is to design efficient parallel algorithms for a variety of Computer-Aided Design (CAD) problems.

We first categorize various parallel processing techniques available for test generation and fault simulation. We discuss the advantages and disadvantages of each of the proposed techniques. We then propose a parallel search method to overcome the deficiencies of inaccurate search heuristics. It will be shown that this method scales very easily with the number of processors and can be implemented easily on configurations with a large number of processors. We show that this method not only results in faster execution of the test generation algorithm but

also results in a better quality of the solution. We also propose a performance model to evaluate the parallel search technique.

We then propose fault partitioning techniques to speed up test generation for faults which are relatively easy to detect. The objective of the fault partitioning techniques is to maximize concurrency without affecting the quality of the overall solution. We propose load balancing techniques which try to minimize the processor idle time with very low communication overhead. We propose a performance model which takes into account the various trade-offs in exploiting parallelism in a test generation/ fault simulation environment.

Finally, we present a parallel test generation system for sequential circuits. A parallel search technique is used to accelerate test generation for hard to detect faults, and a circuit partitioned approach is used to accelerate fault simulation. It is shown that the parallel test generation system is comparable in efficiency as well as performance to most existing test pattern generators.

DEDICATION

TO MY PARENTS

ACKNOWLEDGMENTS

I would like to express my gratitude to the following people:

My thesis advisor Professor Prithviraj Banerjee for his guidance, encouragement and patience. I consider myself fortunate to be able to walk into his room at any time to seek his advice on both technical and non-technical matters.

Professor Janak Patel for his valuable comments regarding this thesis research. I would also like to thank him for organizing the Test Group Meeting which served as an excellent forum for exchange of ideas.

Professor Sudhakar Reddy for introducing me to the area of testing.

Professors David Liu, Constantine Polychronopoulos and Resve Saleh for serving on my thesis committee.

All my friends and colleagues at the Center for Reliable and High-Performance Computing (formerly the Computer Systems Group) and the Coordinated Science Laboratory for their friendship and support.

Finally, my family for their love, support and encouragement from across the oceans.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES