Parallel Processing to Enhance Performance of ATPGs

Awais M. Kamboh (amkamboh@umich.edu)
Ramashis Das (ramashis@umich.edu)
Department of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor

Submitted as the Project Report for EECS-570
December 19, 2005.

Abstract

Automatic Test Pattern Generation (ATPG) is known to be an NP hard problem. To solve such problems, heuristics are used that often require a huge amount of processing. Parallel processing has, thus, been applied to speed up the test generation. There are a few possible solutions to this problem. This report surveys the major techniques used to allow parallel generation of test vectors. We discuss fault parallelism, heuristic parallelism, search space parallelism, functional/algorithmic parallelism and the circuit parallelism. We discuss the pros and cons of these techniques and compare their performances. A few variants of these approaches are also discussed. We use fault partitioning in conjunction with FAN algorithm and report the achieved speedups for ISCAS-85 circuits.

1. Introduction

Generation of test patterns for combinational logic is a search through the set of all input values to find one that causes the output of a good circuit to differ from that of one containing a fault [1].

Much research has gone into increasing the efficiency of algorithms for ATPG. However, the overall gains achieved through these improvements have not kept pace with increasing circuit size, and computation times are still excessive. This report surveys techniques now being explored to map the ATPG to parallel processing machines.

As the size and complexity of IC's continue to grow, the need for fast and effective testing methods for these devices becomes even more important. A significant portion of design time for IC's and digital systems in general, is spent in generating test patterns that distinguish a faulty IC from a fault free one [2]. In order to keep defective products from reaching the market, manufacturers must be able to test their product in an efficient and cost effective manner.

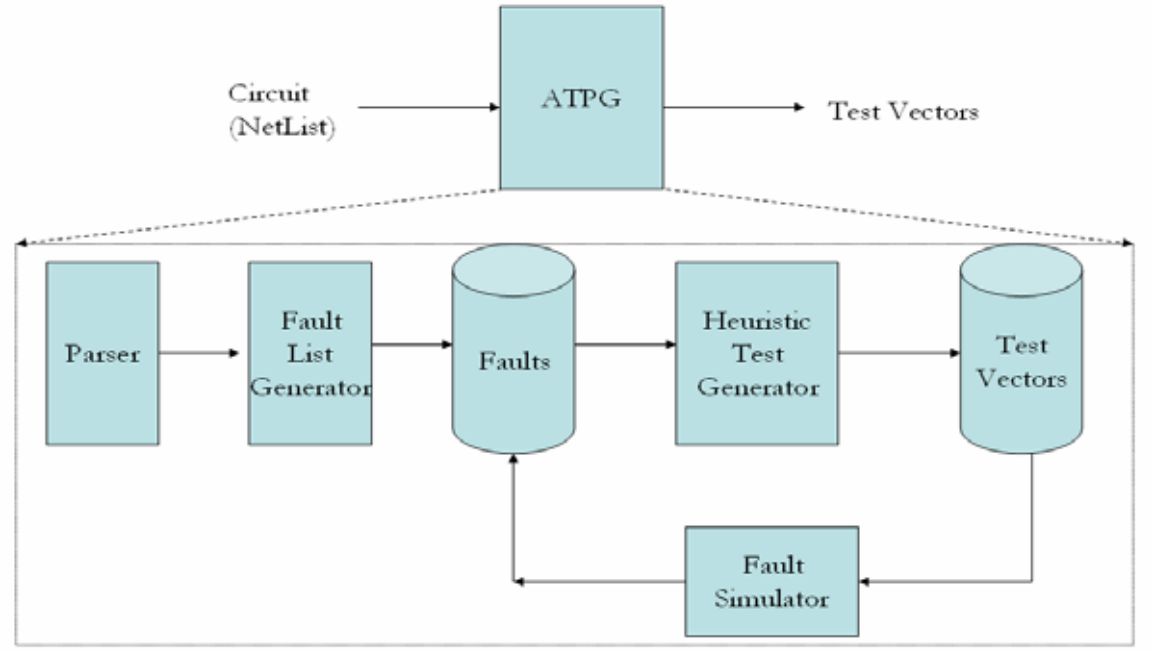

Figure 1: Components of ATPG.

Testing digital circuits must include the two classes of digital circuits: combinational and sequential. For combinational logic circuits, only one test vector sequence is required for stuck-at fault detection. Sequential circuits inherently require the application of a series of test vector sequences for the detection of a fault. Hence, combinational testing is a subset of the sequential test problem. Most sequential test algorithms map the generation of test sequences to iterative combinational test methods. Some techniques allow for the conversion of sequential circuits to combinational circuits for the purpose of testing. This conversion reduces the complexity of test generation for a sequential circuit to that of combinational logic. Therefore, efficient combinational test algorithms are needed to reduce the time spent in test.

Test generation can be achieved either by deterministic test pattern generation or by statistical test pattern generation. Deterministic test pattern generation uses a specific algorithm to generate a test for every fault in a circuit, if a test exists. Statistical test pattern generation randomly selects test vectors, and using fault simulation, determines which faults are detected.

This statistical method can quickly find tests for the easy-to-detect faults, but becomes significantly less efficient when only the hard-to-detect faults remain. Deterministic test pattern generation uses one of numerous Automatic Test Pattern Generation (ATPG) algorithms. ATPG algorithms provide a mechanism to generate a test vector for a specific fault, and fault simulation algorithms are available which can determine if any additional faults are covered by a given vector. As a result, it is now possible to test large circuits within a reasonable period of time.

In addition to using algorithmic techniques to improve the efficiency of ATPG, parallel processing environments can be utilized to reduce computation time. There are several methods available to parallelize ATPG [1,3]. These methods include fault partitioning [4,5,6,7], heuristic parallelization [1], search space partitioning [1,3,6], algorithmic partitioning [6], and topological partitioning [8]. Of these methods, the simplest to

implement is fault partitioning, which divides the fault list across various processors. It is this method of parallelization that is the basis of this investigation.

2. Background

There are two types of parallel processing architectures.

- Shared Memory Architecture
- Message Passing Architecture.

These two differ in their memory organization, resulting in different speed and communication. Programs written for one type of architecture might not perform well when executed on the other architecture.

Shared Memory Systems:
Shared memory systems have single global memory which can be accessed by all processors. Processors have their own caches but the address space is the same. A major characteristic of most shared memory systems is that access to data is independent of the processor making the request and is relatively fast, almost as fast as typical memory access times in a uniprocessor system. However, when many processors are making simultaneous requests to a single memory location or bank, and memory access becomes a bottleneck, access times can increase greatly. For this reason, physical memory layout and data organization within the memory are critical to ensure that the memory system can handle as many simultaneous requests as possible.

In the ATPG problem, the circuit topology information can be kept in a global data structure. The disadvantage of using global variables is that maintaining data consistency is more difficult. Synchronization between processes must be maintained to ensure this data consistency. In general, algorithms for shared memory are easier to design because of their flexibility, but shared-memory machines are more difficult to program and debug because of the need to maintain data consistency.

Message Passing Systems:

Message Passing systems have local memory for each processor but no globally accessible memory. Processors must send messages across some interconnection medium to share data, called a message fabric. It may take hundreds or even thousands of instructions to package a message for transmission, so communication costs are much higher than for shared memory.

Synchronization between processors also depends on messages and is therefore more time consuming than in shared-memory systems. Setup time is much longer on message passing systems because all of the program code and data, such as the circuit topology information, must be loaded across the message fabric. In general, algorithms for message-passing systems are harder to design well, but the programs themselves are easier to implement and debug because data consistency is more easily maintained.

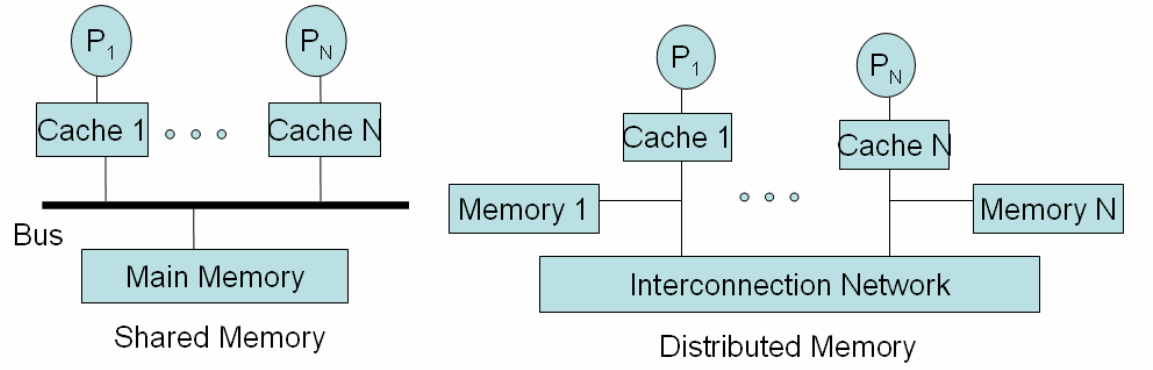

Figure 2: Shared Memory and Distributed Memory system architectures.

The whole idea of moving from uni-processor to multiprocessor based parallel implementation is to reduce the time consumed in test generation, in other words, to speed up the test generation process. Speed up attained by a parallel system is given by

$$Speedup = \frac{T_1}{T_N}$$

Where T_1 is the time taken by 1 processor and T_N is the time taken by N processors.

$$T_N = \left[\frac{M}{N}\right] T_{calc} + T_{comm}$$

Where M is the total number of faults, T_{comm} is the time consumed in communication and T_{calc} is the time taken in test generation.

The goal is to have speedups increase linearly with increasing number of processors. According to Amdahl's law,

$$speedup \leq \frac{1}{s + \frac{(1-s)}{N}}$$

where *s* is the fraction of code that cannot be parallelized and must remain sequential. N is the number of processors. According to this law, if N approaches infinity, the speed up can approach a maximum of 1/s. For example if 5% of the code must be executed sequentially then the maximum achievable speed up is 20.

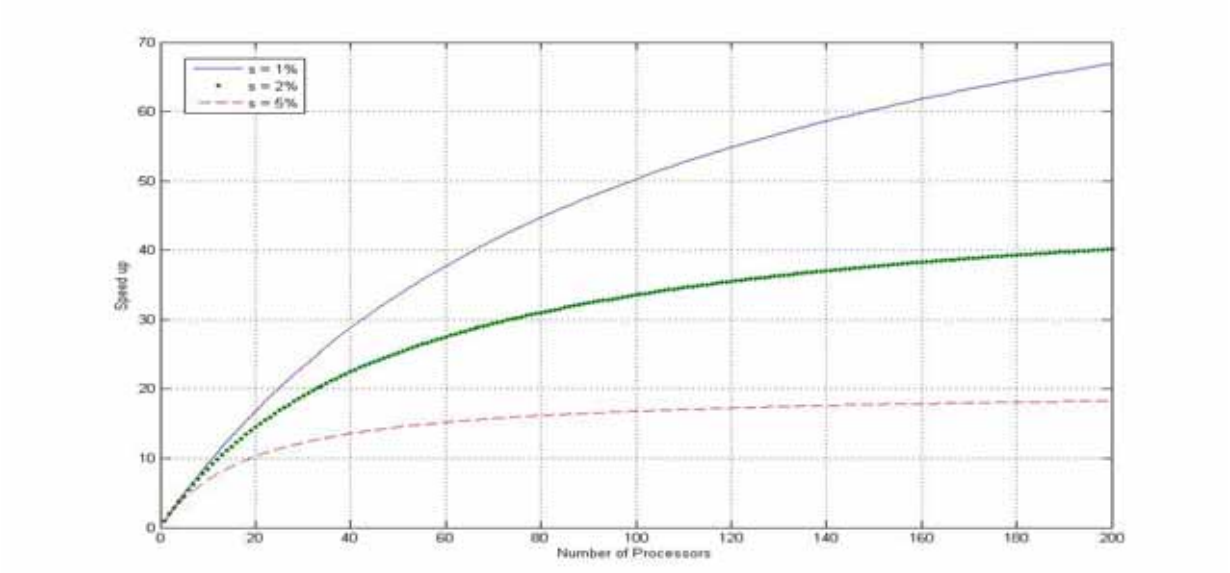

Figure 3: Amdahl's Law, speed ups verses the number of processors.

3. Prior work

There are five main approaches used to adapt serial test generation algorithms for parallel implementation.

- Fault Parallelism
- Heuristic Parallelism
- Search Space Partitioning
- Functional Parallelism
- Circuit Partitioning

Fault partitioning is the simplest to implement, and has many variants. Circuit partitioning is considered to be the toughest with respect to implementation issues. Search space partitioning gives the most promising results but is only good for hard to detect faults. We will see that no partitioning scheme is free from flaws and thus the best result is obtained by combining two or more schemes in some appropriate way so as to reduce the time consumed.

4. Goals

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

- Avoid potential increase in test length by assigning all faults in the same compatible fault [5] set to the same processor.
- Control the degree of communication.
- Use dynamic load balancing if necessary to keep the load balanced on all processes.
- The time for static partitioning should be very small as compared to the actual test generation and fault simulation time.

One criterion for evaluating the quality of a parallel solution to a problem is how well it scales. An algorithm scales well if the computation time decreases linearly, or nearly so, with an increase in the number of processors in the system.

In the following section we discuss various approaches that aim at achieving these goals.

5. Methods

5.1 Fault Partitioning:

Let there be N processors, the main idea is to divide all the faults among these processors and each processor generates tests for its portion of faults.

There are two basic ways of partitioning faults among the processors.

- Dynamic fault partitioning schemes.
- Static fault partitioning schemes.

5.1.1 Dynamic Fault Partitioning: to get maximum speedup on a multiprocessor system, we need uniform load balancing across all processors at all times. In dynamic fault partitioning a fault list is maintained, every processor picks one fault at a time, generates a test for it and removes the fault from the list. This process is repeated until tests for all the faults have been generated and the fault list gets empty. This approach gives almost perfect load balancing. In this case we can predict linear speedups as reported by [11]. However here we have to use test generation algorithm once for every fault, which can amount to a lot of time.

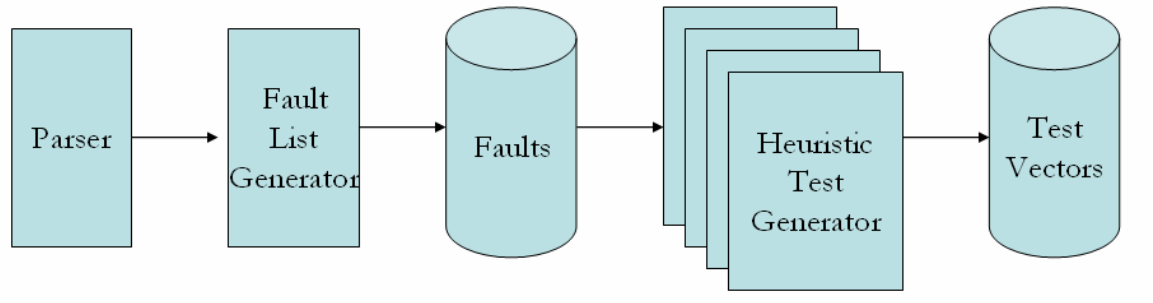

Figure 4: Dynamic fault partitioning model, using only TG.

This problem can be mitigated with the use of fault simulation. In case fault simulation is employed, each time a test is generated, fault simulation is performed to find all other faults this test can detect. The results are communicated to other processors. If any of the processors is generating test for a fault that has been identified by fault simulation step, it aborts the process and picks a new undetected fault [5]. This approach results in a smaller number of tests required for the circuit and since the complexity of fault simulation is less than test generation, this takes smaller amount of time. However these schemes involve a large amount of communication. In Shared memory systems this can lead to increased contention in mutually exclusive accesses to the same memory resources resulting in lost time which adds up significantly for large circuits. In message passing systems this communication overhead is very large to begin with, and extra message overhead is added which is inherent to such systems.

In the above dynamic fault partitioning we see that a lot of time and processing is wasted when a processor has to abort the test generation in cases when some other processor detects the fault during fault simulation step. The number of tests generated for the whole circuit is comparable to the number of tests generated by a uniprocessor. The small test length is owing to the fact that there is a large amount of communication involved between processors when every processor updates all other processors of the faults it has found and this update is repeated for every test generated. Communication overhead is increased when a processor finishes a job and requests a new fault to process. This time can be saved if static partitioning is employed.

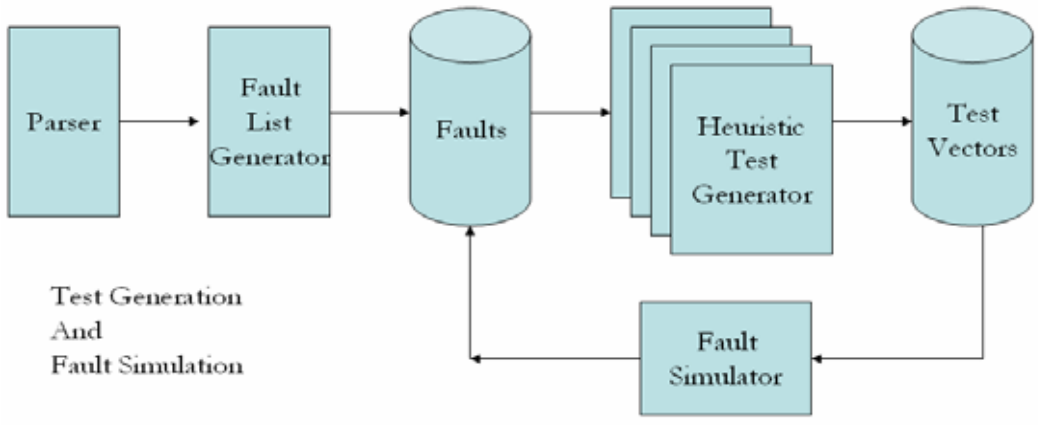

Figure 5: Dynamic fault partitioning using TG & FS.

5.1.2 Static Fault Partitioning: When the faults are partitioned among N processors, such that each processor has its own fault list before the test generation begins, this setup is called static fault partitioning.

Generally in static fault partitioning every processor works independently of other processors, which means once the faults have been partitioned and allotted to each processor, the processors do not communicate any more until all the tests are generated. If load balancing is required then when any of the processors finishes its jobs it requests a portion of jobs from one of the other processors. This communication amounts very little as compared to communication when dynamic partitioning is used. However every processor uses fault simulation, but only checks for its own faults and does not communicate the detected faults to other processors.

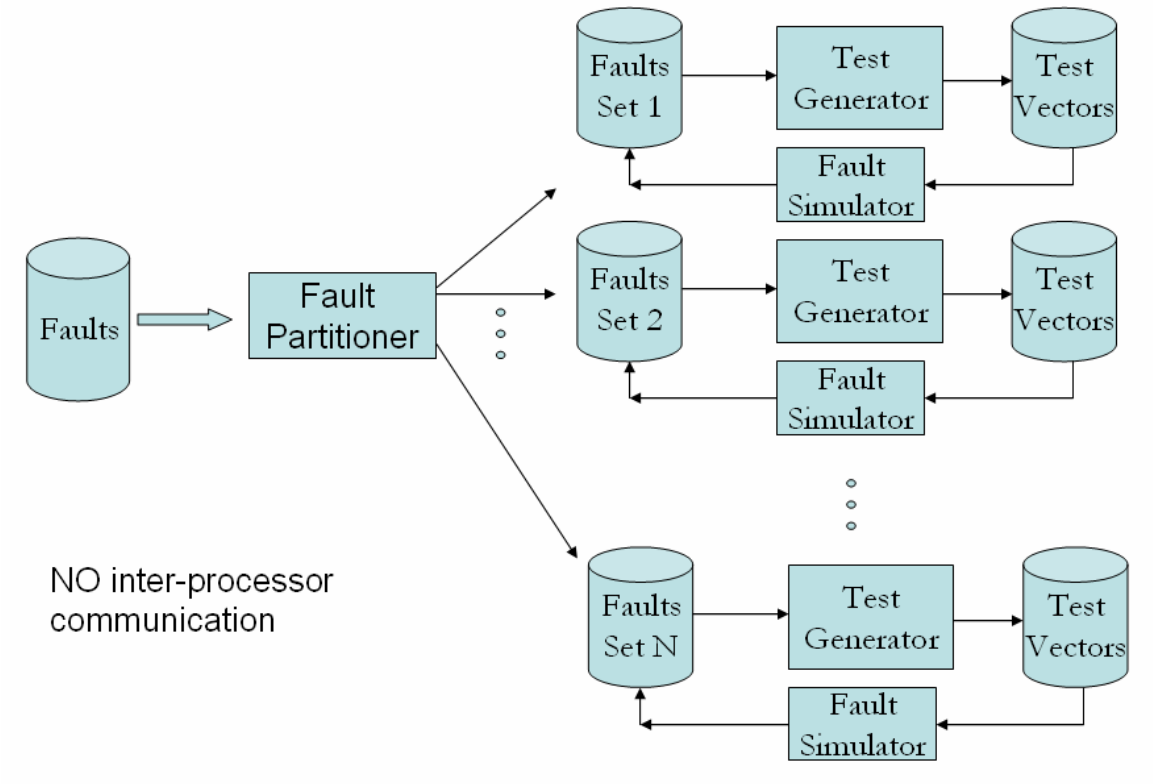

Figure 6: Static Fault Partitioning with minimum inter-processor communication.

This lack of communication reduces the time for overall test generation, because there is no time spent processing messages. However the price is that since each processor is working almost independently, the test length is very large as compared to dynamic fault partitioning or the uniprocessor test generation.

If the fault list is partitioned carefully, each processor will end up doing the same amount of work and thus will consume almost the same amount of time. Such partitioning however is difficult to do a priori.

There are different methods that can be used to partition fault list such that a processor gets faults that are in the same class or are likely to be tested by same tests.

Patil and Banerjee [5] have presented a few methods for static partitioning which include random partitioning, input and output cones partitioning, and mandatory constraint propagation. They define a set of compatible faults as “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.”

Hence our objective is to find out compatible fault sets without using test generation and fault simulation. We have to use heuristic methods to partition in this manner. Since heuristic methods are used, it cannot be guaranteed that two faults in same partition will have a common test. Such sets are called pseudo-compatible fault sets. Assigning east pseudo-compatible fault set to a different processor results in a lower run time and test length owing to more faults being detected during the fault simulation phase.

As the number of processors is increased, 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 expressed that the test length will also be increased.

a. **Random Partitioning**: faults are assigned randomly to the processors. This method is primarily used for comparison with other approaches.

b. **Input cones**: This method assumes that all compatible faults lie in the same fan-out cone of an input. A depth first search is conducted starting from the 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 fan-out cones of that node have already been assigned. Thus faults belonging to more than one input cone will be assigned to only one processor.

c. **Output cones**: this method is same as the input cones method with the difference that search begins from output and proceeds towards the input.

d. **Mandatory Constraint Propagation (MACP)**: A set of constraints is derived based on the testing requirements imposed by each fault [4]. Two faults are considered compatible if their testing constraints match, i.e. the testing requirements do not conflict. A combination of techniques is used [9,10]. The method involves the following key steps.

   a. A pre-processing phase in which the flow dominators of all the gates are generated [9]. Gate *G* is a dominator of a gate *g* if all paths from *g* to any output in the logic circuit pass through *G*.

   b. Starting at the fault site and by using backward and forward implication procedures, a set of uniquely implied logic values is generated. This procedure also gives a set of faults in the vicinity of a fault *f* which may be detected by a test for *f*. All such faults are put in the same pseudo-compatible fault set as *f* (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 [10].

c. 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 *f* 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. As a result, a test for a fault at the output of *g* must detect at least one fault at the output of *G* (either s-a-1 or s-a-0). 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 *f* generated in step *b*.

If two faults are in the same pseudo-compatible fault set, it is very likely that they have a common test vector. All faults belonging to the same pseudo-compatible fault set are assigned to the same processor. It can be seen that finding pseudo-compatible faults sets by mandatory constraint propagation requires much more computation time that any of the methods discussed above. All partitioning methods presented in this section are O(n), where n is the number of lines in the logic network, except partitioning by MACP. Partitioning by MACP takes O(n^2) time which is equivalent to one pass for fault simulation.

One of the greatest disadvantages of fault partitioning is the long setup time for message passing system. Moreover the method also performs poorly if only a few hard-to-detect faults account for most of the processing time. Processors cannot cooperate in generating a test for the same fault. Clearly this method of parallelization is less than optimum, although it is the simplest to implement.

5.2 Heuristic Parallelization: In heuristic parallelization, each processor uses a different heuristic to generate the test for the same fault. It has been found that many heuristics take lesser time than others for a particular fault and generate a test within some time limit when other heuristics fail to do so. Some of the possible heuristics are D-algorithm, FAN, and Podem with different secondary heuristics e.g. every backtrace

should assign values alphabetically and that each alphabet should be assigned a 1 before being assigned a 0. Chandra and Patel [11] have reported results for such a scheme.

There are two basic schemes discusses in literature.

- Concurrent parallel heuristic
- Uniform partitioning

5.2.1 Uniform partitioning: Like the static fault partitioning, fault list is divided among processors and each processor generates tests for its own portion. To generate the tests, however, multiple heuristics are used in sequential order. If a heuristic fails to generate a test within a time limit, it is aborted and next one starts the test generation. This scheme shares the same disadvantages as the fault partitioning scheme, i.e. a lot of time is wasted if the first heuristic doesn't find the test. However, the better part is that multiple heuristics perform well for hard-to-detect faults. Chandra and Patel [11] report almost linear speedup for up to 5 processors with distributed memory.

5.2.2 Concurrent Parallel Heuristics: In concurrent heuristics, the number of processors should be equal to or an integer multiple of number of heuristics available. If they are equal then each processor generates a test for the same fault using one of the heuristics.

If the number of processors is an integer multiple of number of heuristics, then processors are grouped into clusters and each cluster works on a different fault. This case is actually a combination of fault partitioning and heuristic partitioning. Whenever a test is found, the processor informs other processors in the cluster. Other processors abort their procedures and a new fault is selected for the whole cluster.

The concurrent heuristics method can achieve greater speedups as compared to uniform partitioning because of possible anomalies in the ordering of the heuristics for different faults [1]. However in the concurrent heuristic method, we cannot ensure that each processor has a disjoint search space, i.e. every heuristic may follow the same path and a

test may not be found within the available time. One drawback of concurrent heuristic method is that for each fault, the work of all the other processors is wasted. Also there is the communication overhead involved which is not present in uniform partitioning.

As an example, suppose there are 3 heuristics. In uniform partitioning the 'timeout' for each heuristic is set to 5 seconds. If there is a fault whose test can be generated only by the last heuristic, say in 3 seconds, then the total time taken by uniform partitioning will be 13 seconds, whereas concurrent parallel heuristics will take only 3 seconds. On the other hand, if there is a fault whose test can be generated by the very first heuristic in uniform partitioning, then both the methods will take 3 seconds, however, the work of 2 processors in concurrent parallel heuristics will be wasted, and there will be communication overhead as well.

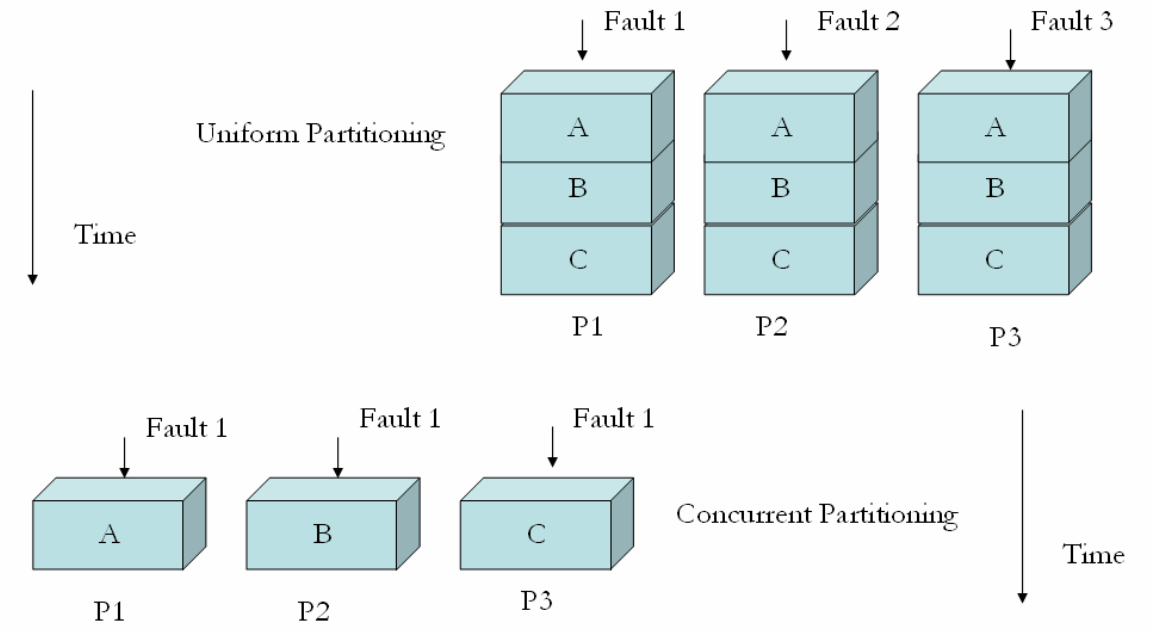

Figure 7: Heurisitic partitioning example.

Chandra and Patel [11] report that due to these reasons, for most benchmark circuits, the concurrent heuristic method generally doesn't perform as good as uniform partitioning method. In circuits where there are very few or no hard-to-detect faults, the concurrent partitioning method doesn't show any speedups at all over uniprocessor systems.

Whereas the uniform partitioning method can give linear speedups for such circuits, this is because most of the tests can be generated using first heuristic.

Uniform Heuristic Detector	**Uniform heuristic**	**Concurrent heuristic**	**Proc. work wasted in concurrent heuristic**
Heuristic C	5+5+3=13 sec	3 sec	0
Heuristic A	3 sec	3 sec	2 proc. For Concurrent

Table 1: Result of heuristic partitioning example.

The main disadvantage of the heuristic partitioning and fault parallelization techniques is that the processors cannot work together to find a test for a single hard-to-detect fault, especially when there are a few hard-to-detect faults that take up very large amount of time.

5.3 Search Space Partitioning: Search space partitioning is a method that allows the processors to work collectively and cooperate in finding a test for a single fault.

One way to cooperate is by dividing the algorithm into smaller tasks that can be completed in parallel. Such a division is called functional, algorithmic or AND parallelism.

Another way is to divide the search space into disjoint spaces and evaluate them simultaneously, called search space partitioning or OR parallelism. Patil and Banerjee [3] adapted this method based on Podem because Podem orders the search space and allows it to be divided easily.

The heuristic used may not lead the algorithm to the shortest path to the solution, or may not lead to the solution at all. According to Patil and Banerjee [3], previous research shows that increasing the number of backtracks in an algorithm with a particular heuristic does not necessarily result in better performance. They believe this is because the assignment that caused the conflict, and hence the backtrack, is not necessarily the

assignment that was just made. In other words, heuristics can order the search space, such that the cause and effects of backtracks are not close to each other in the search tree [1].

One way to solve this problem is to divide the search space such that sub-problems skipped by one processor are evaluated by another. The search spaces for the processors are therefore disjoint. This setup helps finding the solution much quicker.

Consider for example, Figure 8, a representation of the binary search space for a J s-a-0 fault in the circuit under test. This search space was constructed using the simple heuristic of always trying the logic 1 value on a primary input first. Since assignment of the value 1 for node B in the left-hand subtree is inconsistent, all solutions that live below B in that part of the solution space can be pruned from the search tree. The ordering of the search space also lets it be divided into disjoint sections so that work on the different sections can proceed simultaneously. Note that the processor must have access to the entire circuit description.

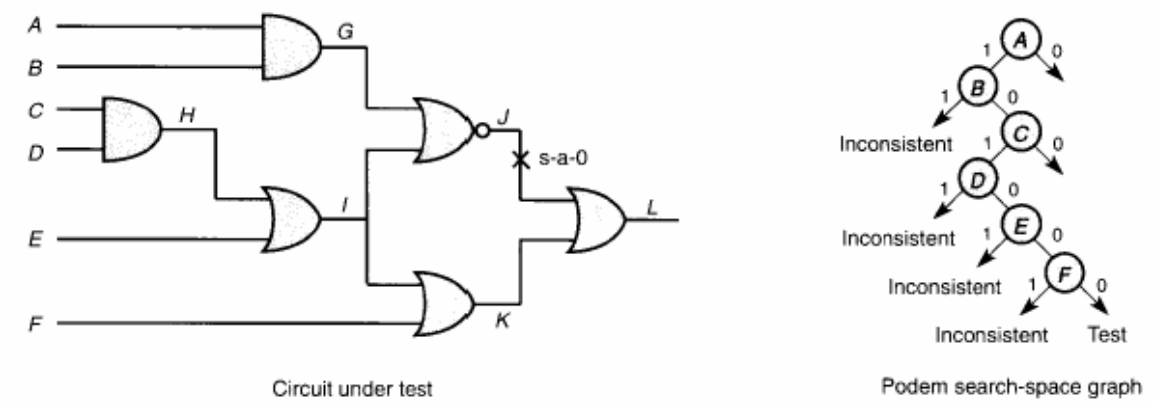

Figure 8: Circuit under test and its corresponding Podem search-space graph [1]

Figure 9 shows the division of a search tree. The search space of processor X is divided into two spaces, processor X and Y. Note that the processors are always working on different problems i.e. disjoint search spaces and that each processor will backtrack to a different edge. If processor X finds a conflict, it backtracks and tries alternate input value of A, if processor Y finds a conflict, it backtracks and tries input C. This approach tends to make the search more efficient.

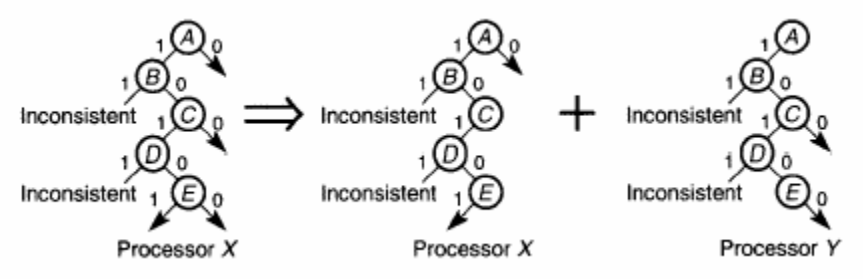

Figure 9: Example of search space partitioning [1]

This partitioning is implemented such that a processor generates the search tree and divides it among all processors as shown. If a processor searches its entire search tree and doesn't find a test, it stops and requests a portion of the search space from a busy processor. If a processor succeeds in generating a test, it informs others and they stop work, a new fault is picked and the process begins again.

Patil and Banerjee [3] published results for several benchmark circuits and made several observations. First is that increasing the number of allowed backtracks does not yield better results in a uniprocessor implementation, a parallel algorithm produces better results for the same number of backtracks. The reason is that the parallel algorithm searches a larger portion of solution space. The parallel algorithm exhibits almost linear speedups with respect to the number of processors. In one case it produced super-linear speedups, which are sometimes possible in branch-and-bound algorithms because of anomalies in search tree.

This approach requires long setup time and the entire circuit must be loaded onto each processor. On the other hand, each processor works independently of others and there is very small communication overhead involved.

Researchers [1,6] have mentioned another method for dividing the search space among the processors. Instead of representing the search space as a binary tree, they represent it as a general m-ary tree in which each node can have an arbitrary number of children.

5.4 Functional Partitioning: Also known as Algorithmic parallelism or AND parallelism, this method also allows processors to cooperate in finding the test for a fault. This scheme is based on the idea of dividing the algorithm into smaller tasks, such that each task can be completed simultaneously independent of others.

Most of the ATPG algorithms are difficult to parallelize with respect to sub-tasks. Subtasks such as fault sensitization and path sensitization are not independent and thus cannot be parallelized. Action taken to perform one of these processes may change the circuit state such that it has a side effect or causes an inconsistency in another process.

One way to allow parallelism in justification is to perform justification for goals in different faults simultaneously. This technique resembles the fault partitioning described earlier. Another method proposed in [6] by Motohara et al. divides the fault list into sub-lists of compatible faults, as described in Static Fault Partitioning section. These compatible faults include those along the same path between fault site and a primary output. The sub-lists are sent to pairs of processors which include a test generator and a fault simulator. The test generator generates a test for the first fault using Podem, if a test is not generated within a small number of backtracks, the fault is considered hard-to-detect. If a test is found, it is sent to a fault simulator; other faults covered are detected and removed from the list.

This technique reduced the size of both the remaining fault list to be processed by the algorithm as well as the length of test set. However, in some cases performing fault simulation after test generation might not be required. For Example, if the ATPG algorithm sensitizes a path for a stuck-at fault that leads to a primary output, every node in that path is also tested for a stuck-at fault. If the ATPG algorithm keeps a list of these incidentally tested faults and removes them from the fault list, fault simulation may not be required and additional computation time could be saved [1].

In another setting, suppose we have divided the fault list and assigned a section to a cluster of processors. The test generator processors in the cluster take faults from the list

and perform test generation on them. When a test is found, it is passed to a fault simulator processor, which performs fault simulation using that test vector. If the faults are easy to detect, number of processors doing fault simulation will be approximately equal to number of processors doing test generation. If a test generator encounters hard-to-detect faults and backtracks a lot, then the number tests being generated will decrease, and the fault simulator processors will begin to get idle. If the system stops the work at this point, the work done so far will get wasted.

A better solution is to take an idle fault simulator processor and put it to work generating a test for the hard-to-detect fault using search-space division. So all the processors will remain busy and lesser processing would be wasted. The disadvantage of this method is that a processor has to know how to do both the fault simulation and parallel test generation. In shared memory systems and message passing systems with adequate memory to store both programs, this technique could be effective.

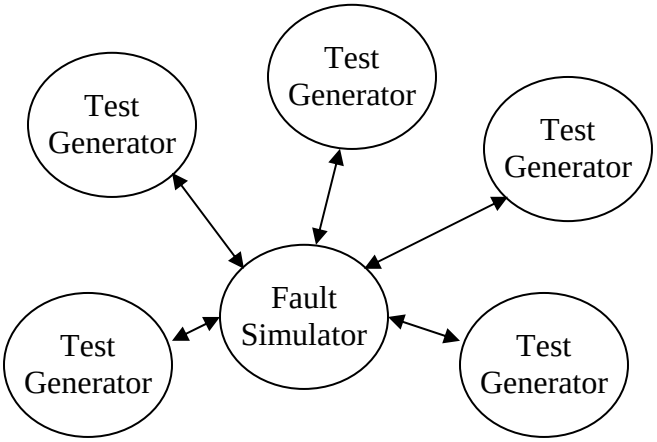

Figure 10: An example functional partitioning scheme.

So the most common known technique is to separate test generation and fault simulation. It is observed that the time spent for test generation is significantly greater than time spent on fault simulation so several test generators are coupled with one fault simulator to keep it from getting idle for longer periods.

5.5 Circuit Partitioning: For large circuits, it may not be possible for every processor to maintain the entire database because of the memory constraints. In this approach, the circuit is partitioned; each processor keeps a partition of the circuit and performs various operations such as backtracing, justification and implication on its own subcircuit.

Forward implication is a major component of test generation which can be parallelized by circuit partitioning using parallel logic simulation methods on each subcircuit.

There are many circuit partitioning schemes available for logic simulation namely natural, random, level-wise, input cones, output cones and string-wise. [12,1].

Natural partitioning consists of dividing gates into groups according to their position in the gate list. Partitioning by gate-level refers to assigning gates to groups according to their level in the circuit, i.e. their distance from primary input. Partitioning elements by strings refers to grouping sets of connected gates that include at most one fan-in or fan-out connection. In case of fan-out cones, when gates are connected to more than one fan-out cone they are assigned to the one they are more heavily connected to.

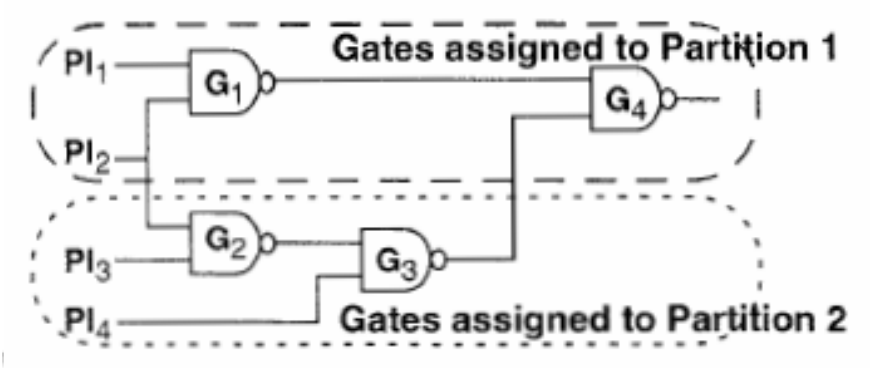

Figure 11: Circuit partitioning by input cones [16].

The optimum partitioning depends on algorithm used for test generation. Podem uses a simulation like process for justification, hence input / output cones partitioning works

better. D-algorithm tends to be alone circuit levels, as during advance of the D-frontier so gate level partitioning produces better results.

Research shows that input / output cones give the best balance between communication overhead and concurrency [13].

An issue in circuit partitioning for test generation is the number of gates in each partition, also called block-size. As the number of gates in a block decrease, the amount of work that can be done between communications decreases. For message passing systems this could prove highly inefficient, or in other words the circuit size would be significantly large.

5.6 Reported Results

Authors	**Methods Used**	**Scalability**
Patil & Banerjee [4]	Fault & Algorithmic	TG with FS: Nearly linear upto 8 Processor
Chandra & Patel [11]	Fault & Heuristic	Uniform: Nearly linear upto 5 proc. Concurrent: Less than linear for upto 5 processors
Patil & Banerjee [3]	Search Space	Linear for upto 16 processors, Super linear in some cases
Motohara et al. [6]	Algorithmic & Search Space	Algorithmic: Linear for upto 10 processor Search-Space: Linear for upto 50 processors
Kramer [14]	Circuit	Linear for circuits with 15-18 inputs

Table 2: Results reported by different authors.

6. Our Implementation

Though we initially aimed at writing our own parallel-ATPG, we ended up parallelizing a standard ATPG ATALANTA [17]. ATALANTA is an ATPG program developed at the Virginia Polytechnic Institute. Based on FAN algorithm, ATLANTA used both random and deterministic test pattern generation. We identified the functions involved and used Pthreads and mutex locks to parallelize the algorithm. We disabled the random test generation to find the speedups gained on FAN algorithm.

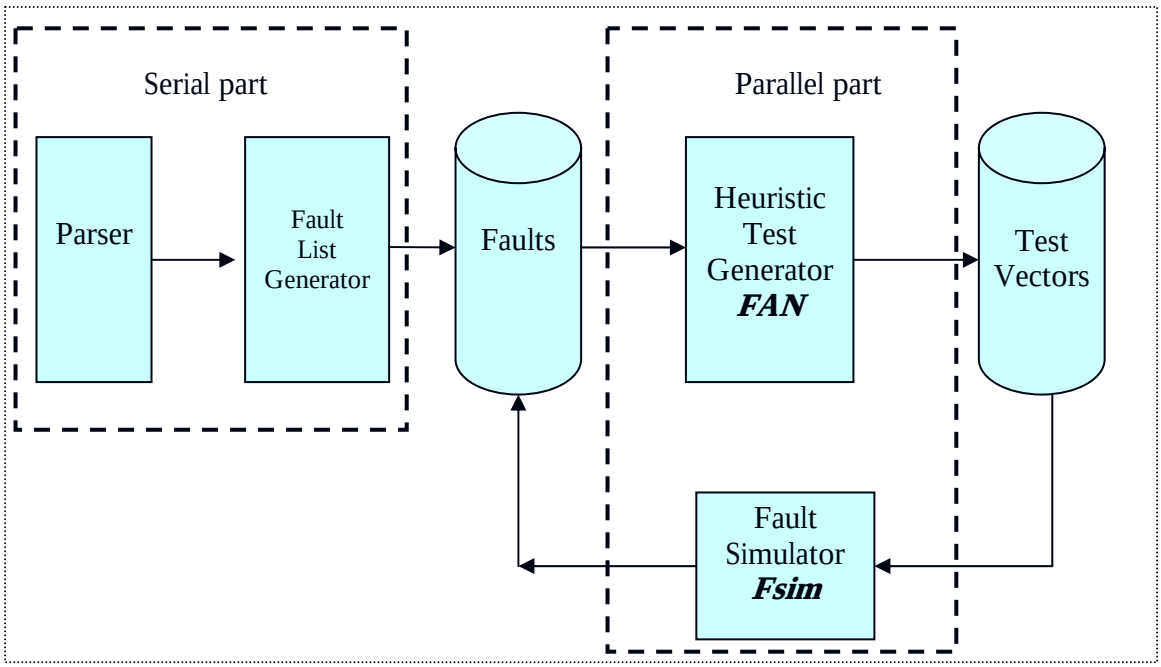

Figure 12: Our approach of parallelizing the ATPG ATALANTA.

The basic blocks of ATALANTA are shown in figure 12. The *parser* converts a net-list written in *.bench* ISCAS89 format to an internal data-structure that is used by the other blocks. *Fault-list generator* generates all possible stuck-at faults for the circuit. These two blocks consume a negligible fraction of the total time (refer to table 3) and are difficult to parallelize. Hence, in our implementation, these two blocks comprise of the serial part. The most processing-intensive block is the *test-generator* that picks a fault from the fault-list and generates a test-vector for that fault. ATALANTA uses FAN algorithm to find a test vector for a fault. The *fault simulator* finds other faults detected

by the test vector and removes them from the list of faults. In our implementation we parallelize the *FAN-Fsim* blocks. We used POSIX threads and targeted our implementation for shared memory architecture.

Benchmark circuits	Serial part		Parallel part		
	Parser	*Fault generator*	*FAN*	*Fsim*	*Total*
c7552	0.0013	0	2.3742	0.0093	2.3848
c6288	0.0006	0	0.5022	0.0063	0.5091
c5315	0	0	0.3727	0.0041	0.3768
c3540	0	0	0.3286	0.0039	0.3325

Table 3: Time taken by various blocks of ATALANTA for a single processor run.

Table 3 suggests that the serial part of the code takes negligible time and hence is dropped for time calculations in further experiments. Another implication of the time distribution is that we should be able to achieve almost *k* speed-up for a *k*-processor machine using *k* threads. Also, the structure of original ATALANTA code and using threads makes it difficult (almost impossible!) to calculate the times for *FAN* and *Fsim* blocks separately. Hence we only use the total time as the performance measure for further experiments.

In our implementation, we used the serial part as-it is, though had to change a lot of the code to support the modified parallel part. In other words, the functionality of the serial part was not changed. The serial part generates a list of faults in an array *fault-list*. We distribute this array into various threads and let them run *FAN* and *Fsim* on them separately, in turn modifying the global array *fault-list*. Mutex locks are used to maintain consistency of the global array. We implemented two approaches of distributing the *fault-list* to the threads:

1. Dynamic fault partitioning: In this, a *free* thread takes up the *next* available fault from the *fault-list*. This is implemented by maintaining a global *next_index* variable that keeps track of the *next* fault to be considered. Though this leads to very good load balancing due to its inherent dynamic nature, it doesn't perform as good as the second method. The reason can be attributed to a high contention for

acquiring the lock on *next_index*. A typical circuit consists of thousands of faults which implies that there is a chance of contention thousands of times!

2. Static fault partitioning (row-block): The array *fault-list* is statically divided into *k* parts in a row-block fashion, where *k* is the total number of threads. This shows much better speed-up as shown by their run on dual-core Intel Xeon. But when we ran it on 4-processor Simpool machines, there was an interesting trend of speedup with increasing number of threads. There was speed-up even beyond 5 threads that clearly implied bad load balancing. Hence, we went into another fault partitioning scheme.
3. Fault partitioning (row-cyclic): The increase in speed-up for row-block distribution scheme for $k > 4$, encouraged us to look into *fault-list*. It is observed that the *fault-list* contains faults in increasing order of a fault's distance from input. It takes larger time to generate test pattern for a fault farther from input. This clearly implies that row-block was distributing faults such that the quickly detectable faults went into the first thread, and so on. So, the easiest solution is to distribute the faults in row-cyclic fashion. And as expected we see better speed-up with this scheme.

8. Results and analysis

Since ATALANTA is a combinational ATPG, we used the ISCAS-85 benchmark suite circuits to measure the performance of our implementation. First, to compare the performance of dynamic and static partitioning, we ran our code on dual-code Xeon machine. Table 4 lists the run-time and speedups for various configurations. Figure 13 plots the speed-ups of the two partitioning scheme. It clearly indicates that dynamic fault partitioning, though balances load evenly, causes excessive contention on the lock on *next_index.*

Benchmark		Single thread	Static portioning (Row block)		Dynamic partitioning	
Circuit name	Input size (n)	Runtime (in sec)	Runtime (in sec)	Speedup	Runtime (in sec)	Speedup
		1 thread	2 threads			
c7552	207	2.371	2.009	1.18	2.368	1.00
c6288	32	0.536	0.316	1.70	0.342	1.57
c5315	178	0.377	0.3168	1.19	0.339	1.11
c3540	50	0.328	0.23	1.43	0.232	1.41

Table 4: Run-times and speed-ups obtained on Xeon dual core machine.

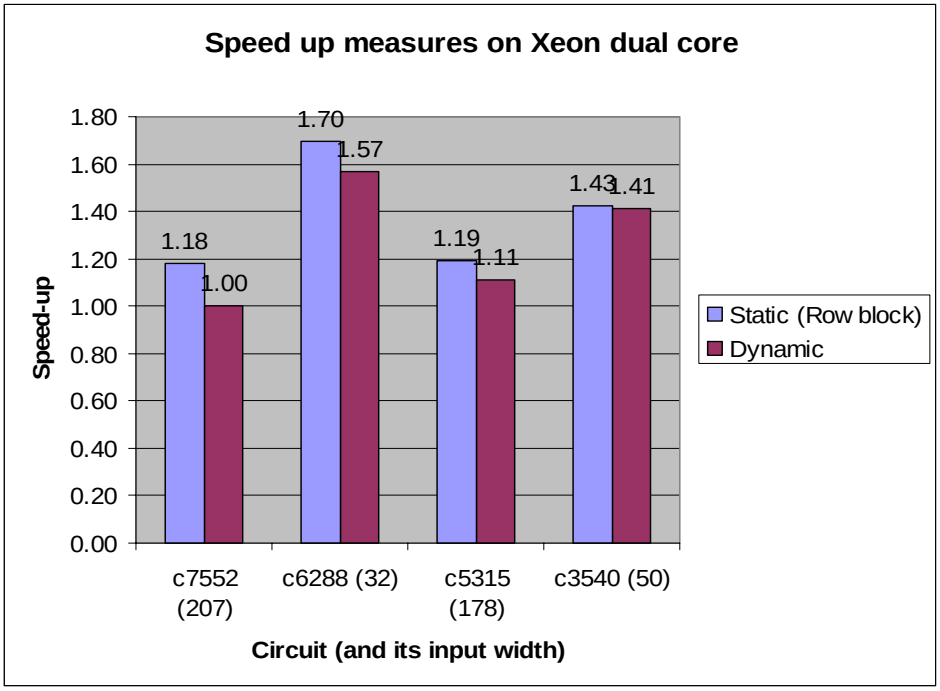

Figure 13: Speed ups obtained on Xeon dual core processor for dynamic and static partitioning.

An interesting observation, which repeats even for the remaining experiments, is that the circuits with smaller input width show higher speed-up. The reason for this is that it is *easier* to find a test-vector for a fault if the input-width is large, because *FAN* starts from the fault location and traces back to the inputs to assign a possible value that will sensitize the fault. More inputs mean lesser backtracking since there are more available values to sensitize the fault. This implies that *FAN* takes more time to find a test-vector for a fault in c6288 than it takes for c7552. As we are parallelizing the *test-pattern finding* part, it leads to higher speedup for c6288 than c7552. But again, if the load

balancing is good, the speed-ups should be similar for all the circuits. This is what we observe later.

Since Xeon processor takes fraction of seconds for most of the benchmark circuits, run-times are prone to noise. So we decided to conduct rest of the experiments on the slower Simpool machines. Note that it also provides the platform for using more than just 2 threads. Also, we dropped dynamic partitioning method as it doesn't look very promising due to high contention on the global array.

Bench-mark circuit	Number of threads										
	1	2		3		4		5	7	10	32
	Time	Time	Speedup	Time	Speedup	Time	Speedup	Speedup	Speedup	Speedup	Speedup
C7552	38.281	22.478	1.703	15.130	2.530	12.316	3.108	3.340	3.070	3.010	3.377
C6288	50.423	26.884	1.876	18.108	2.785	13.392	3.765	3.233	3.546	3.663	3.905
C5315	10.562	5.950	1.775	4.003	2.638	3.118	3.387	2.925	3.007	2.996	3.151
C3540	5.400	3.128	1.726	2.310	2.338	1.551	3.483	2.803	2.800	2.818	2.909
C2670	3.801	2.203	1.725	1.574	2.415	1.125	3.378	2.993	3.410	3.166	2.850
S5378	7.419	4.410	1.682	3.254	2.280	2.496	2.973	2.546	2.926	2.662	2.600

Table 5: Runtimes and speed-ups for row-block fault distribution on simpool.

First we executed the static portioning based on row-block scheme on simpool machines with various numbers of threads. Run-times and speedups obtained are shown in table 5. For an extra data-point, we converted a sequential ISCAS-89 benchmark circuit s5378 to combinational by breaking the D-F/Fs into inputs and outputs. Figure 14 plots the speed-ups against the number of threads used. An interesting observation is that the speed-up increases even after increasing the number of threads beyond 4. This clearly implies that the load distribution between processors is not even, which allows a 5 threaded program to run faster than 4 threaded program on a 4-processor machines

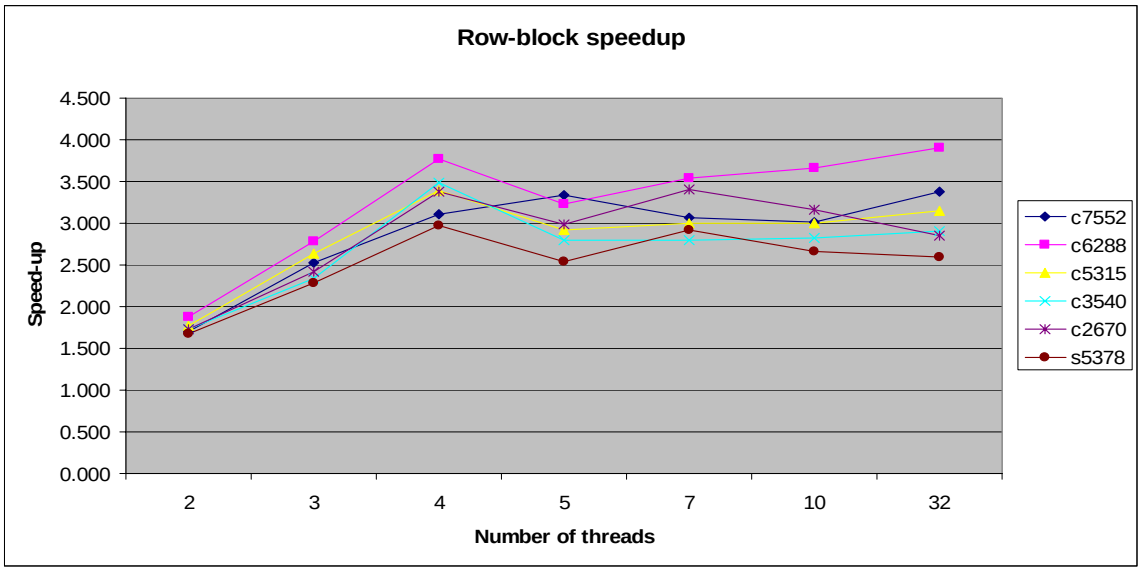

Figure 14: Plot of speed-ups against number of threads.

Bench-mark Circuit	1	2		3		4		5	7	10	32
	Time	Time	Speedup	Time	Speedup	Time	Speedup	Speedup	Speedup	Speedup	Speedup
C7552	38.281	19.834	1.930	14.140	2.707	11.409	3.355	2.881	3.421	3.099	3.268
C6288	50.423	27.074	1.862	17.387	2.900	14.445	3.491	3.182	3.636	3.691	3.932
C5315	10.562	5.672	1.862	3.861	2.736	3.015	3.503	2.991	3.196	2.915	3.410
C3540	5.400	3.021	1.788	1.946	2.775	1.514	3.567	3.058	3.234	3.051	3.221
C2670	3.801	2.105	1.806	1.455	2.613	1.162	3.272	2.915	3.264	2.790	2.999
S5378	7.419	4.127	1.798	2.968	2.500	2.323	3.193	2.664	2.908	2.701	2.617

Table 6: Runtimes and speed-ups for row-cyclic fault distribution on simpool.

This result encourages us to think of partitioning the faults in a more intelligent fashion. A simple way is to distribute the faults from *fault_list* to the threads in a cyclic fashion as the faults are roughly in increasing order of difficulty of finding test-vector. Cyclic distribution of faults gives better results, as expected. Table 6 lists the speed-ups of this new distribution scheme. Figure 15 plots the speed-ups corresponding to the thread numbers.

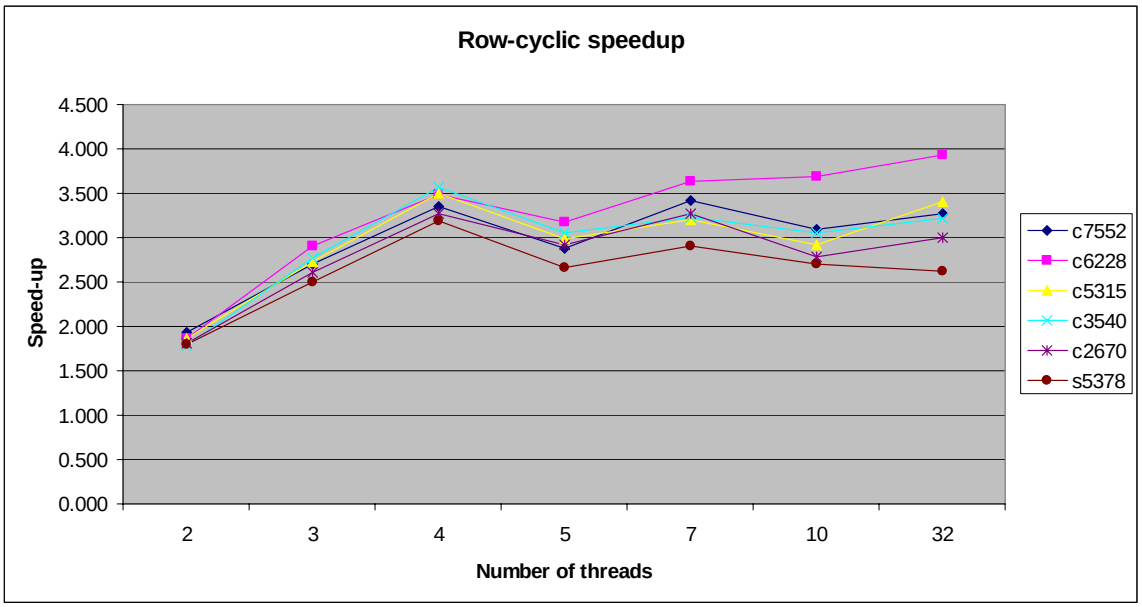

Figure 15: Plot of speed-ups against number of threads.

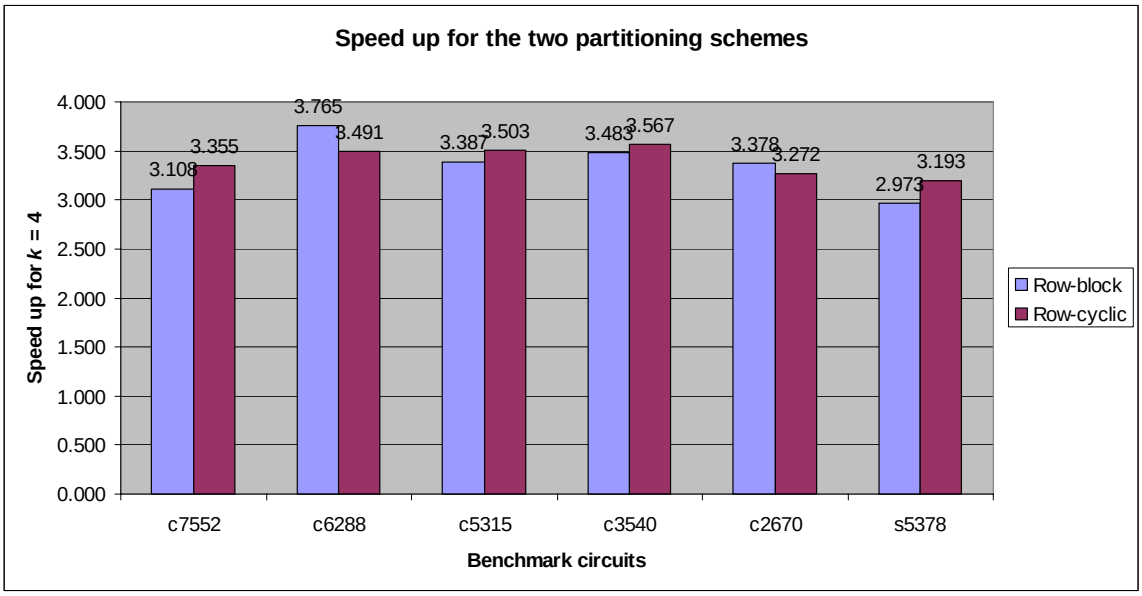

Figure 16: Comparison of speed-ups for various benchmark circuits.

We observe a linear speed-up (Fig. 14 & 15) till $k = 4$, which is as expected. Also, the speedups are very high which supports the fact that most of the processing time of the ATPG is parallelized. An interesting observation is that the speed-up of c6288 increases with increasing number of threads even beyond 4! In-fact, for $k = 32$, the speedup almost

touches 4 which is unexpected and difficult to explain! There seems to be some sort of artificial speed-up along with better load-balancing in this particular case. Figure 16 plots the speed-up of the two partitioning schemes for $k = 4$ for the benchmark circuits. Except for c6288, we see that row-cyclic performs better than row-block, as expected.

9. Conclusions

Our experiments show linear speedup which justifies a correct implementation. This study and various previous studies agree that none of the existing schemes contains all the good qualities desired for efficient parallelization of test generation. Therefore, a combination of schemes can yield better results than all of its constituent schemes. There is no scheme that gives best results for every possible circuit under test. Similarly almost every scheme performs better than all others for some specific group of circuits. The communication overhead becomes the dominant bottleneck for the scalability of any algorithm. Therefore most of the algorithms produce linear speedups up to a few processors, then their performance starts decreasing with increasing number of processors.

10. Future work

We need a scheme that can distribute work to processors such that they can co-operate in test generation for hard-to-detect faults and is also efficient for easy-to-detect faults. Secondly the initial set up time for existing schemes is very significant, as is the communication overhead, so a better scheme should have a balance between the initial setup time and the run-time communication overhead.

We also need to maintain a balance between the time taken for test generation, fault coverage and the corresponding test length. The ‘time out’ technique used in many of the above mentioned schemes to save time may stop the search for a test prematurely even when there exists a test for that specific fault which will inversely effect the fault coverage.

Many algorithms try to minimize the communication involved between processors, in doing so they compromise the length of test set. The resulting test set is much bigger than the one generated by a uniprocessor for the same circuit and can be far from minimal. There is no algorithm known as yet, that can guarantee a minimal test set without having generated the complete fault table.

For shared memory machines, the memory access time creates the bottleneck, advances in fast access memories will surely contribute towards faster test generation.

The communication overhead is more critical in Message passing systems as compared to shared memory systems, but message passing systems are more scalable and most of the modern systems with tens of processors are based on message passing architecture.

10. References:

[1] Klenke, R.H.; Williams, R.D.; Aylor, J.H., "Parallel-processing techniques for automatic test pattern generation", IEEE Computer, Volume 25, Issue 1, Jan. 1992 Page(s):71 – 84

[2] Wolf, J.M.; Kaufman, L.M.; Klenke, R.H.; Aylor, J.H.; Waxman, R.; "An analysis of fault partitioned parallel test generation" Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, Volume 15, Issue 5, May 1996 Page(s):517 - 534

[3] S. Patil and P. Banerjee, "A Parallel Branch-and-Bound Algorithm for Test Generation," Proc. 26th ACMI/IEEE Design Automation Conf., CS Press, Los Alamitos, Calif., June 1989, pp. 339-334.

[4] S. Patil and P. Banerjee, “Fault Partitioning Issues in an Integrated Parallel Test Generation Fault Simulation Environment,” Proc. I989 Int’l Test Conf., CS Press, Los Alamitos, Calif. 1989, pp. 718-726.

[5] S. Patil, P. Banerjee, Performance Trade-Offs in a Parallel Test Generation/Fault Simulation Environment, IEEE Transactions on Computer-Aided Design, Vol. 10, No. 12, December, 1991, pp. 1542-1558.

[6] A. Motohara, K. Nishimura, H. Fujiwara, I. Shirakawa, A Parallel Scheme for Test-Pattern Generation, IEEE International Conference on Computer-Aided Design, 1986, pp. 156-159.

[7] T. Inoue, T. Fuji, H. Fujiwara, On the Performance Analysis of Parallel Processing for Test Generation, IEEE Asian Test Symposium, 1994, pp. 69-74.

[8] R. H. Klenke, R. D. Williams, J. H. Aylor, “Parallelization Methods for Circuit Partitioning Based Parallel Automatic Test Pattern Generation”, Proceedings of the IEEE VLSI Test Symposium, April, 1993, pp. 71-78.

[9] D. Harel, “A Linear Time Algorithm for Finding Dominators in Flow Graphs and Related Problems.” Proc. 17th ACM Symp. on Comp., pp. 183-184. May 1985.

[10] S . B. Akers, C. Joseph, and B. Krishnamurthy, “On the Role of Independent Fault Sets in the Generation of Minimal Test Sets,” Proc. IEEE Int’l Test CO$. 1987, pp. 1100-1107.

[11] S. Chandra, J.H. Patel, “Test generation in a parallel processing environment,” Proc. Int. Conf. Computer Design, Oct. 1988.

[12] S.P. Smith, B. Underwood, and M.R. Mercer, “An Analysis of Several Approaches to Circuit Partitioning for Parallel Logic Simulation,” Proc. IEEE

Int'l Conf Computer Design: VLSI in Computers and Processors, CS Press, Los Alamitos, Calif., 1987. pp. 664-667.

[13] P. Banerjee, "Parallel algorithms for VLSI computer Aided Design", PTR Prentice Hall, 1994.

[14] G.A. Kramer, "Employing Massive Parallelism in Digital ATPG Algorithms," Proc. 1983 Int'I Test Conf., CS Press, Los Alamitos, Calif., 1983, pp. 108-114.

[15] D. E. Culler, J. P. Singh, and A. Gupta, Parallel Computer Architecture, A Hardware/ Software Approach, Morgan Kaufmann, 1999.

[16] Klenke, R.H.; Aylor, J.H.; Wolf, J.M.; "An analysis of fault partitioning algorithms for fault partitioned ATPG" Proceedings of 14th VLSI Test Symposium, 1996., 28 April-1 May 1996 Page(s):231 - 239

[17] H.K. Lee and D.S. Ha, "Atalanta: an Efficient ATPG for Combinational Circuits". Technical Report, 93-12, Dep't of Electrical Eng., Virginia Polytechnic Institute and State University, Blacksburg, Virginia, 1993.