An Analysis of Fault Partitioned Parallel Test Generation[1]

Joseph M. Wolf
Lori M. Kaufman
Robert H. Klenke
James H. Aylor
Ron Waxman
Department of Electrical Engineering
University of Virginia

Center For Semicustom Integrated Systems
Department of Electrical Engineering
University of Virginia
Charlottesville, Virginia 22903-2442

1. Support for this work was provided in part by the Advanced Research Projects Agency under contract number J-FBI-89-094 and the National Science Foundation under grant numbers CDA-9211097 and ASC-9201822.

An Analysis of Fault Partitioned Parallel Test Generation

Abstract

Generation of test vectors for the VLSI devices used in contemporary digital systems is becoming much more difficult as these devices increase in size and complexity. Automatic Test Pattern Generation (ATPG) techniques are commonly used to generate these tests. Since ATPG is an NP complete problem with complexity exponential to circuit size, the application of parallel processing techniques to accelerate the process of generating test vectors is an active area of research.

The simplest approach to parallelization of the test generation process is to simply divide the processing of the fault list across multiple processors. Each individual processor then performs the normal test generation process on its own portion of the fault list, typically without interaction with the other processors. The major drawback of this technique, called fault partitioning, is that the processors perform redundant work generating test vectors for faults covered by vectors generated on another processor. An earlier approach to reducing this redundant work involved transmitting generated test vectors among the processors and re-fault simulating them on each processor.

This paper presents a comparison of the vector broadcasting approach with the simpler and more effective approach of fault broadcasting. In fault broadcasting, fault simulation is performed on the entire fault list on each processor. The resulting list of detected faults is then transmitted to all the other processors. The results show that this technique produces greater speedups and smaller test sets than the test vector broadcasting technique. Analytical models are developed which can be used to determine the cost of the various parts of the parallel ATPG algorithm. These models are validated using data from benchmark circuits.

I. Introduction

As the size and complexity of integrated circuits (ICs) continues to grow, the need for fast and effective testing methods for these devices becomes even more important. A significant portion of design time for ICs, and digital systems in general, is spent in generating test patterns that distinguish a faulty IC from a fault free one. 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. Currently, up to one third of the design time for ASICs is spent generating test vectors [1].

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. Further, *Design for Test* (DFT) 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 which normally utilize the gate level description of a system to generate a condensed set of tests. 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 [2,3]. These methods include fault partitioning [4,5,6,7,8,9,10],

heuristic parallelization [8,11], search space partitioning [2,7,12,13], algorithmic partitioning [7], and topological partitioning [14,15,16]. 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.

The techniques presented in this paper improve the performance of an existing fault partitioned parallel ATPG system for combinational circuits by introducing efficient interprocessor communications. Dynamic load balancing is used to reduce processor idle time. Detected fault broadcasting [18,19] is used to reduce the amount of redundant work performed in generating excess test vectors. This technique will be shown to be an improvement on the vector broadcasting technique proposed in [17].

II. Background

This section presents the background for the paper. Included are discussions of PP-TGS, the fault partitioned parallel test generation system developed for this effort, and Mentat, the parallel programming environment upon which PP-TGS is based.

2.1 The Mentat Parallel Processing Environment

The Mentat parallel programming environment [20] was designed to provide an efficient easy-to-use, machine independent environment for developing and executing parallel programs. Mentat includes a set of language abstractions based upon C++ and a set of portable run-time support facilities that provide separation between the programmer and the physical system. Mentat supports object-oriented design, location transparency, encapsulation of parallelism, automatic detection and management of communications and synchronization, and scheduling transparency. There are two primary components of the Mentat environment, the Mentat Programming Language (MPL) [21], and the Mentat run-time system [22].

MPL is an object-oriented programming language based upon C++. C++ objects are used to encapsulate computation and indicate parallelism to the compiler. These objects are placed on remote processors for execution. Communication is realized through method invocations and returns on these remote objects. MPL programs are compiled using the Mentat compiler, MPLC. MPLC translates MPL programs into standard C++ while introducing calls to the Mentat run-time library as necessary.

The Mentat run-time system provides for management of remote object instantiation, initialization, scheduling, and destruction. It also provides for communication between objects

using the native communication protocol for the machine upon which it is running. In the case of a network of Unix workstations, the TCP/IP communication protocol is used. Mentat also provides fault tolerant capabilities in case one or more of the network processors fails.

Mentat presents a complete methodology for utilization of both homogenous and heterogeneous networks of workstations as distributed memory parallel processors. In addition to Mentat, there are other parallel programming environments that can make a network of workstations perform as a parallel processor [23, 24].

2.2 PP-TGS Software

The Parallel fault Partitioning Test Generation System (PP-TGS) program consists of seven independent objects. Four of these, *main, tgs, reader,* and *writer,* exist only on the master processor. The other three, *mini-master, test generator,* and *fault simulator,* are distributed to every processor. The architecture of the system is shown in Figure 1.

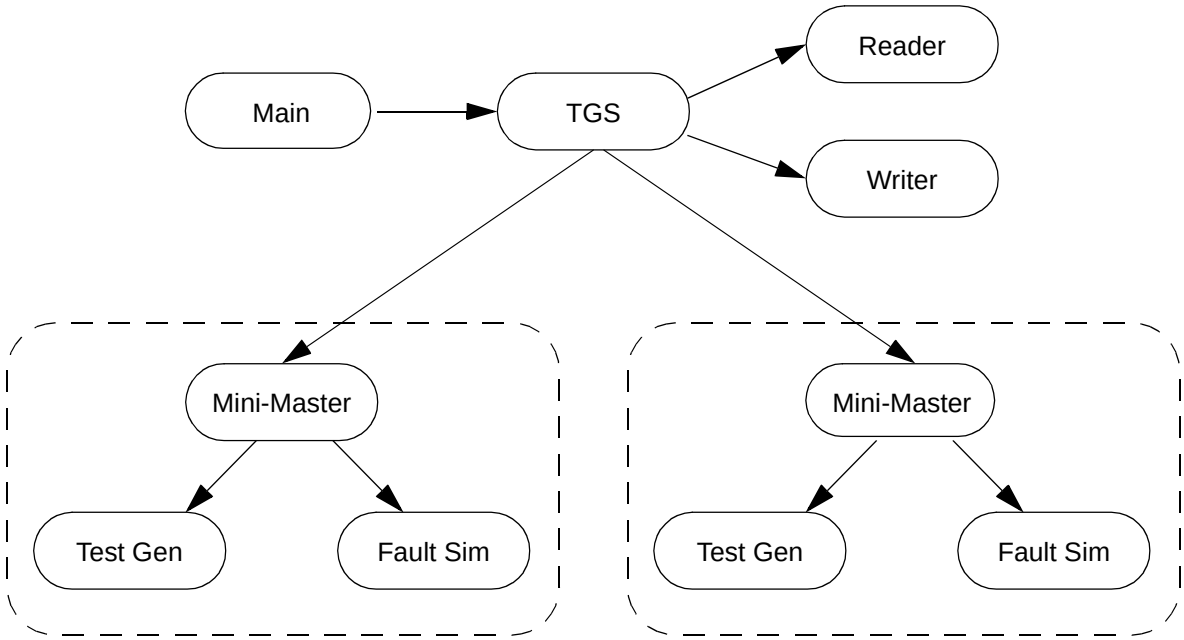

Figure 1. PP-TGS Object Hierarchy

The program begins when *main* is called. *Main* is responsible for initiating the PP-TGS system. *Main* invokes *tgs,* the global master, which creates the *reader* and *writer* objects. The *reader* reads the PODEM file from disk and maintains a circuit database. The *reader* is no longer needed after the system is initialized and, at that time, is deleted. The *writer* is responsible for collecting the test vectors as they are generated by the *mini-masters*. The *tgs* analyzes the circuit

database and stores the circuit topology while generating a list of possible stuck-at faults. The fault list is partitioned according to the number of processors in the parallel network. The *tgs* then invokes the *mini-masters* on the remote processors, initializing each with a copy of the circuit structure and its own fault partition.

The actual partitioning of the fault list is performed in a preprocessing step. For the benchmark circuits considered, either input or output cone partitioning was used depending on the circuit. Input and output cone partitioning have been shown to produce the best trade-off of partitioning time vs. partition quality when used for fault partitioned ATPG [5]. In this context, partition quality refers to the degree of processing time balance among the partitions and the amount of "independence" in the fault sets of each partition.

Each *mini-master* creates a *test generator* and a *fault simulator* on its processor. The local *mini-master* selects a fault from its fault list and sends it to the *test generator.* The *test generator* attempts to develop an input vector that will detect the fault if present in the circuit. If such a vector exists, then the *test generator* reports the resulting vector back to the *mini-master.* If no test can be found, the *mini-master* marks this fault as uncovered.

After a new test vector is generated, the *mini-master* passes it to the *fault simulator.* The *fault simulator* returns a list of all faults in the circuit that are detected by this particular vector. The *mini-master* marks all of the detected faults in its partition since separate vectors will not need to be generated for these faults. In the initial system, covered faults outside of the *mini-master's* partition are disregarded. This approach will result in redundant work as the other *mini-masters* generate tests for the already detected faults. Therefore, the resulting test set is larger than necessary, a major limitation to speedup in this version of the system. The runtimes in this system are also increased by the fact that some *mini-masters* finish their fault list ahead of the others, i.e. the load across the processors are unbalanced.

As new vectors are generated by the various *mini-masters*, they are reported to the *writer* object which eliminates any duplicates. When all faults in a particular *mini-master's* fault list have either been marked as detected or as uncovered, that processor has finished. When all processors have completed their work, the *writer* reports the total number of test vectors generated, the number of vectors in the test set after duplicates have been eliminated, the percentage of faults covered by the test set, and the time it took for the PP-TGS software to run. The *writer* can also write the list of generated test vectors to a file. A simple flow chart of the test generation process without any interprocessor communication is shown in Figure 2.

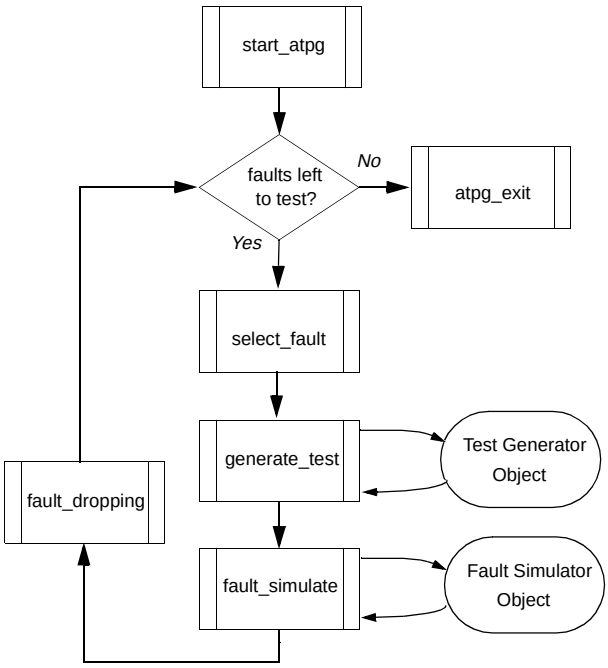

Figure 2. Flow chart of the PP-TGS ATPG process without interprocessor communications.

The *test generator* uses the PODEM [25] test generation algorithm. The *test generator* receives a fault from the *mini-master* and, if possible, generates a test for the fault. The resulting test is then returned to the *mini-master.* A backtrack limit is included to limit time spent processing redundant and hard-to-detect faults.

The fault simulator used is FSIM [26], one of the fastest combinational fault simulators available. FSIM uses parallel pattern single fault propagation to achieve its improved performance. Because FSIM simulates one fault at a time, only the fault's propagation zone need be evaluated. The propagation zone of a fault for a given test vector is the set of gates whose inputs are effected by the presence of that fault in the circuit. Hard-to-detect and redundant faults have inherently short propagation zones and, therefore, do not take long to simulate for most individual vectors. This is an important benefit because hard-to-detect and redundant faults must be simulated many times throughout the ATPG process. Easily detected faults tend to have long propagation zones, but because they are easily detected, these faults are normally only fault simulated for a small number of test vectors. Therefore, for combinational circuits, single fault propagation is a faster technique than concurrent (parallel) fault propagation [27]. The PP-TGS system does not take advantage of FSIM's parallel test vector processing because, during ATPG, test vectors are simulated one at a time.

The *fault simulator* receives the test vector produced by the *test generator* from the *mini-master* and performs fault simulation. The *fault simulator* maintains a copy of the global fault list and performs fault simulation on all faults in the circuit. The *fault simulator* returns the list of detected faults to the *mini-master.*

Other, more efficient, test generation algorithms such as those used in [28] and [29] could be implemented in the PP-TGS system simply by incorporating them into the *test generator* object. The remainder of the objects in the system would remain unchanged. The performance implications of this change are discussed in Section 4.6.3.

2.3 Modifications to PP-TGS

The major limitation of the original PP-TGS software was the lack of interprocessor communications during ATPG. First, since there was no load balancing among the processors, there was a problem of processors becoming idle before the test generation process was complete. Also, the lack of communication of detected fault information caused a considerable amount of redundant work to be performed.

2.3.1 Dynamic Load Balancing

The PP-TGS system statically partitions the fault list prior to fault processing. As stated previously, it is impossible to effectively partition the fault list *apriori* such that the workload for all processors is equal. Dynamic load balancing allows the fault list to be repartitioned at runtime when workload imbalances are detected. As demonstrated in [4,5], dynamic load balancing can significantly improve the performance of fault partitioned ATPG by minimizing processor idle time.

The basic test generation process, as described in the previous section, was modified to include remote method invocations to provide a means of splitting the fault list between a busy and an idle processor. The additional code performed the following tasks:

work_status:

This remote method is invoked by an idle mini-master to poll other mini-masters to determine the number of faults that mini-master has left to test.

rpt_work_status:

This remote method is invoked in an idle mini-master in response to a work_status call. It selects the first mini-master to respond with a list of

untested faults greater than a minimum threshold value. This method terminates the overall ATPG process when all processors are idle.

split_list:

This remote method is invoked by the idle mini-master to split the fault list of a busy mini-master as determined by rpt_work_status. The fault list of the busy mini-master is simply split in half.

split_store_faults:

This remote method is invoked by the mini-master that is dividing its fault list. It stores the fault list in the idle mini-master.

set_value:

This remote method invocation is invoked by the mini-master that is dividing its fault list. It resets the variables in the idle mini-master to allow for a continuation of the ATPG process.

The flow chart for the test generation process with the addition of dynamic load balancing is shown in Figure 3. Once a *mini-master* has finished test generation on its fault list, it broadcasts a message to all of the other *mini-masters* that it is idle and looking for work. It performs this broadcast by invoking the *work_status* method on every other *mini-master.* The remote *mini-masters* reply to this method invocation by sending the number of faults they have left unprocessed to the idle *mini-master.* This reply is performed by invoking the r*pt_work_status* method on the idle *mini-master.* This asynchronous communication method of invoking a remote method and having the reply sent back through another remote method invocation, instead of simply waiting for a reply was used to avoid deadlock. Deadlock could occur if, when using the call-and-wait method, two processors happened to call each other simultaneously looking for work.

When the idle *mini-master* finds a busy *mini-master*, it splits the busy *mini-master's* list through the *split_list* and *split_store_faults* methods. After the new faults are stored, the test generation process is continued. Termination is detected when every *mini-master* responds with a *rpt_work_status* call with no faults left unprocessed, at which point the idle *mini-master* will exit.

2.3.2 Detected Fault and Generated Vector Broadcasting

As discussed in Section 2.2, the major limitation to the performance of the PP-TGS system is the generation of redundant test vectors for faults already covered by test vectors generated on remote processors. This research compares two methods for eliminating this problem: detected fault broadcasting [18] and generated vector broadcasting [17]. Both methods were added as options to the base PP-TGS system.

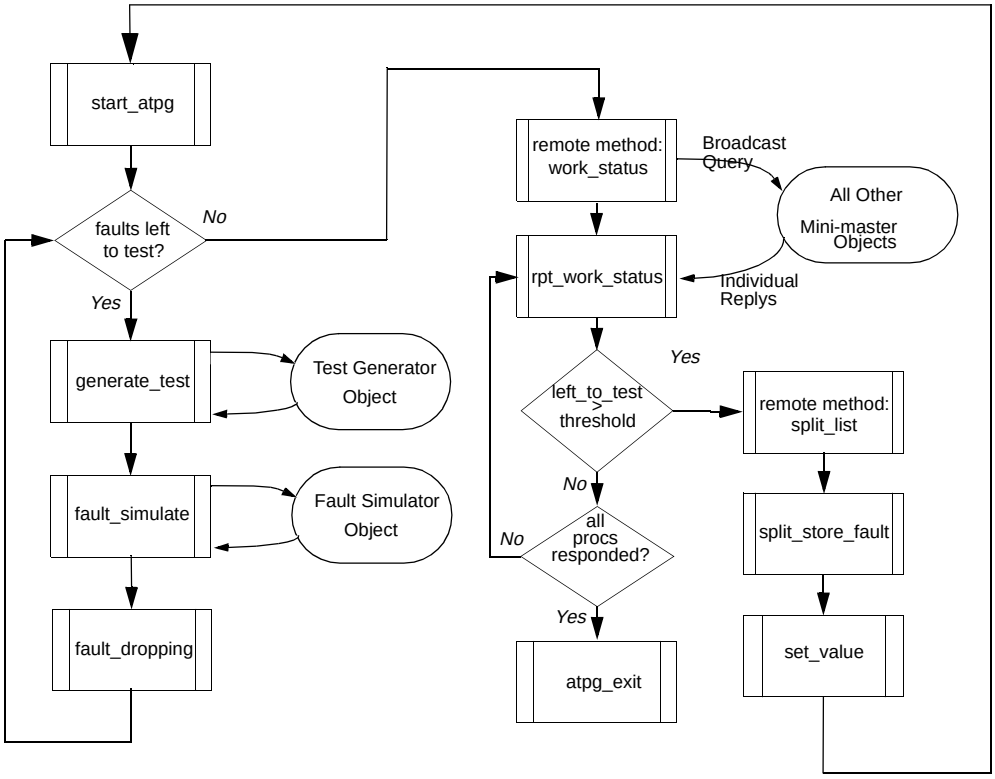

Figure 3. Flow chart of the PP-TGS ATPG process with dynamic load balancing.

Fault broadcasting occurs during the fault simulation phase of ATPG. In the fault broadcasting system, the *mini-masters* invoke fault simulation on local copies of the global fault list. When a processor generates a vector to cover an assigned fault, the *mini-master* often discovers that extra faults are covered that are not in its local partition. In response to this event, it sends a message to all the other processors informing them of the additional faults. This activity enables the appropriate *mini-masters* to eliminate those faults from their partitions, as they have already been detected. Because fault broadcasting decreases the number of test vectors generated, it decreases program execution time and generates shorter fault detection experiments.

Fault broadcasting during parallel ATPG for sequential circuits was presented in [19]. The system presented in [19] synchronizes the processors after every test generation pass and requires that the entire global fault list be broadcast during fault broadcasting. The system presented herein is asynchronous, in that no processor has to wait for faults to be broadcast from other processors. Also in this system, only newly detected faults are broadcast.

Vector broadcasting occurs after the test generation phase. Prior to fault simulation, the *mini-master* sends the newly generated vector to each of the other *mini-masters*. The remote *mini-masters* perform fault simulation on the vector and eliminate any detected faults from their own copy of the global fault list. Experiments revealed that no significant benefit could be gained by simulating only those faults in the local partition, because the fault simulator spends the majority of its time in good circuit simulation.

Since fault simulation is deterministic, vector broadcasting and fault broadcasting communicate the same information. The distributed vectors can be considered encoded versions of the fault broadcasting information and the remote fault simulators can be considered decoders. Because distributing test vectors involves communicating smaller packets of information than distributing detected faults, there is less communications overhead associated with vector broadcasting. The price of lower communications cost is that every test vector must be simulated by every fault simulator. This approach introduces redundant processing directly proportional to the number of processors in the system. The use of generated vector broadcasting rather than detected fault broadcasting trades redundant computation for improved communications.

To implement detected fault broadcasting, the ATPG system was modified to include remote method invocations to provide a means of disseminating the information regarding the remotely detected faults among the processors. Figure 4 shows the flow chart of the test generation process with dynamic load balancing and detected fault broadcasting. During fault broadcasting, the local *mini-master* sends its list of detected faults to the other *mini-masters* through an invocation of the *found_faults* method. The *found_faults* method eliminates the detected faults from the remote *mini-masters*' fault lists.

In addition, a version of the PP-TGS system was developed to implement vector broadcasting. Figure 5 shows the flow chart of the test generation process with dynamic load balancing and generated vector broadcasting. Vector broadcasting is accomplished by passing generated vectors to the remote *mini-masters* through the *vector_broadcast* method. *Vector_broadcast* invokes fault simulation and fault dropping on the processor on which it resides, thereby eliminating covered faults from further consideration.

III. Results

All versions of the PP-TGS software were executed on a network of eight Sun SPARC2 workstations running Mentat connected by Ethernet. The results presented were obtained during the early morning hours when the processor workloads were minimal. Several of the ISCAS '85

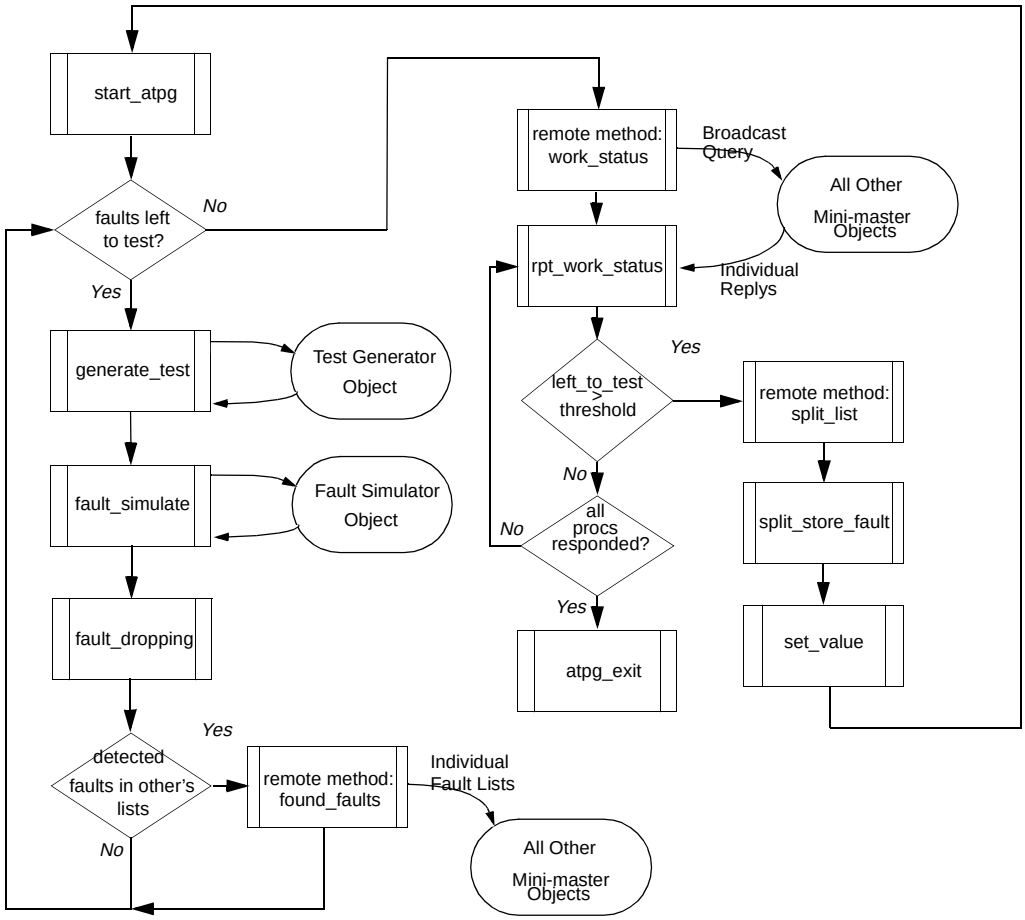

Figure 4. Flow chart of the PP-TGS ATPG process with dynamic load balancing and detected fault broadcasting.

combinational benchmark [30] and ISCAS ‘89 sequential benchmark circuits were tested. The sequential circuits were processed by assuming full scan design. Each D flip-flop input was mapped to a primary output, and, conversely, each D flip-flop output was mapped to a primary input.

Results are presented for four different communications configurations: no communications, load balancing only, detected fault broadcasting and load balancing, and vector broadcasting and load balancing. Table 1 records the execution runtimes of the various communications configurations. Table 2 displays the size of the test sets and the fault coverage. The runtime results do not include program instantiation and initialization time.

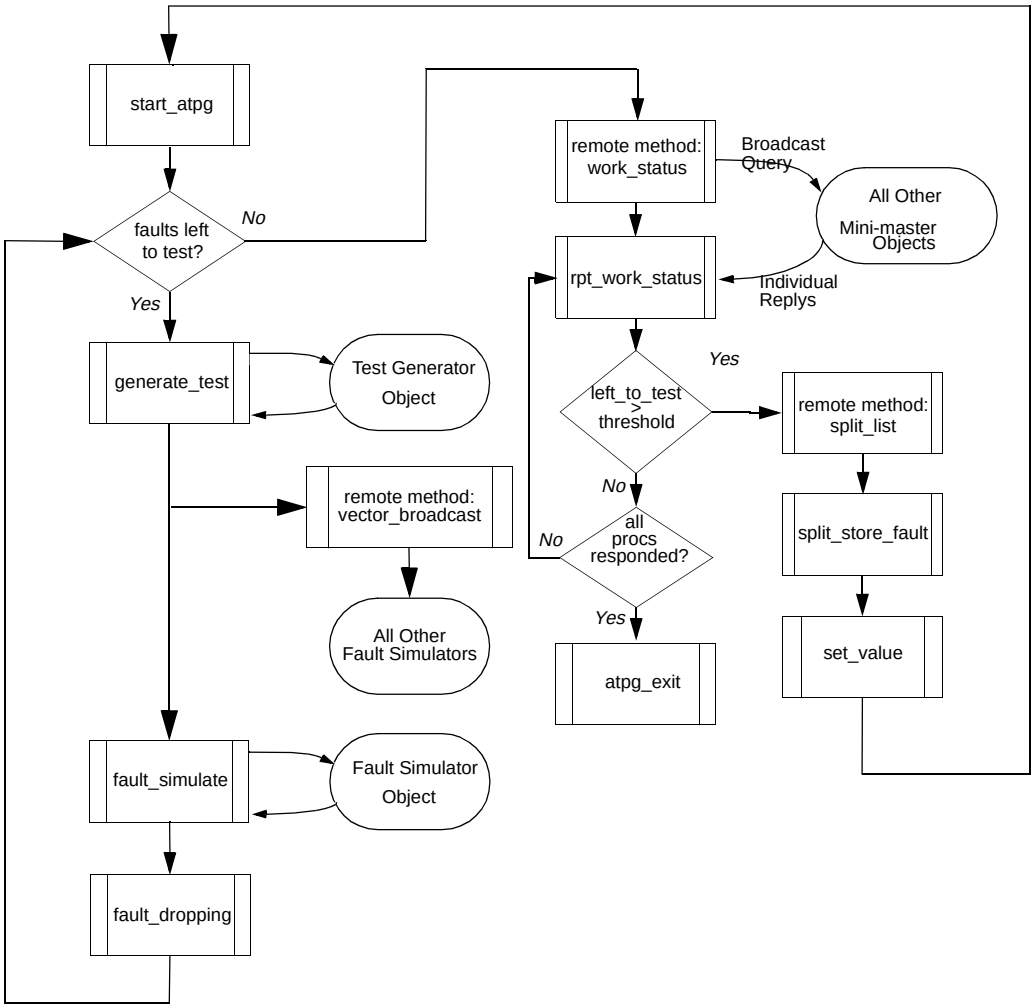

Figure 5. Flow chart of the PP-TGS ATPG process with dynamic load balancing and generated vector broadcasting.

3.1 Dynamic Load Balancing

Dynamic load balancing almost always improves system performance since it involves relatively little communications overhead. Figure 6 uses circuits c7552 and s9234 to illustrate the benefits of load balancing. When every *mini-master* has faults left to process, ATPG proceeds as if there were no interprocessor communication. When a *mini-master* completes its fault list, faults are passed from one processor to another in a point to point transfer. The message traffic associated with dynamic load balancing is linear with the number of processors. Since at most half of the *mini-masters* can split their fault lists with the other half, a maximum of N/2 load balancing fault packets, where N is the number of processors, can exist on the network at any

Table 1 Runtime (Seconds)

Circuit	Number of Procs	No Comm	Load Bal	Fault Br and Load Bal	Vect Br and Load Bal
c499	1	2.689			
	2	5.368	5.736	5.386	5.590
	4	5.897	6.896	6.259	5.383
	8	10.527	9.444	11.929	5.716
c1908	1	17.517			
	2	18.775	17.893	14.345	16.534
	4	13.478	14.988	9.674	12.058
	8	9.700	12.607	14.253	20.733
c3540	1	226.418			
	2	151.400	136.109	121.506	130.742
	4	97.480	87.024	64.954	78.153
	8	55.434	57.176	47.793	43.601
c7552	1	510.665			
	2	389.757	321.290	269.066	277.620
	4	234.550	211.350	147.077	168.626
	8	169.738	139.492	80.279	98.591
s5378	1	78.464			
	2	81.357	71.010	49.563	56.091
	4	52.779	45.256	27.966	37.455
	8	35.946	30.045	20.458	31.343
s9234	1	1772.982			
	2	1073.590	1031.520	938.655	962.313
	4	638.610	566.695	477.894	503.178
	8	605.617	380.773	255.293	363.497
s13207	1	607.378			
	2	544.585	442.483	341.721	368.881
	4	322.724	274.149	178.891	213.179
	8	318.330	180.114	103.756	147.102

given time. Because this limit will not be reached until the very end of the ATPG process when the fault lists exchanged are quite small, there is little opportunity for data collisions on the network. Therefore, the most significant communications penalty associated with dynamic load balancing is synchronization delay.

The dynamic load balancing system was tested in order to determine with which busy *mini-master* the idle manager should share work. The idle *mini-master* could select either the busy *mini-master* with the most work, thus minimizing the number of load balancing calls, or the first busy *mini-master* to respond, thus reducing synchronization delay. Analysis revealed that hard-to-detect faults impede response to *work_status* calls. Because the *test generator* is not interruptible, once a processor begins test generation on a hard-to-detect fault, it cannot respond to requests to share faults. If the idle processor is to accept work from the busiest processor, it must wait for all processors to respond before load balancing. This situation leaves the processor

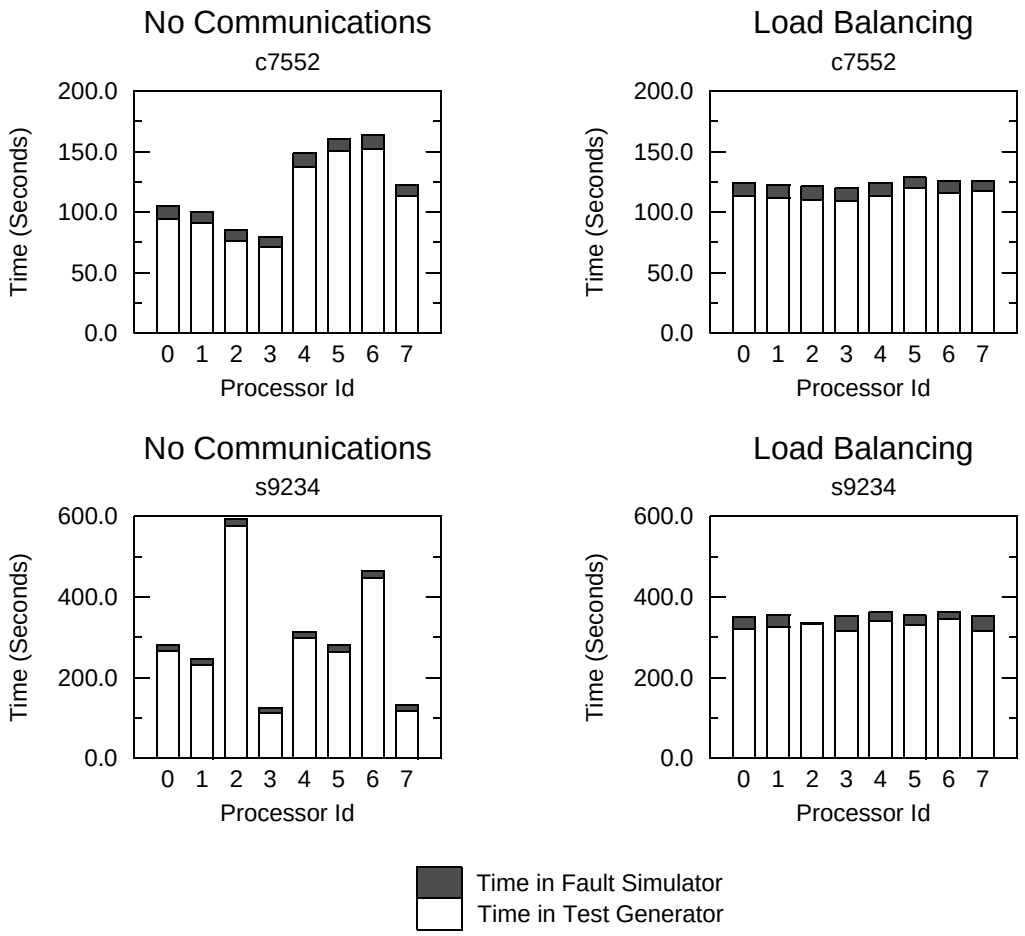

Figure 6. Benefits of Dynamic Load Balancing

idle for extended periods of time. For medium to large sized circuits, the best results are obtained by selecting the first busy *mini-master* to respond to the *work_status* call. The results in Table 1, Figure 6, and Figure 7 show that the performance of the PP-TGS system improved significantly with load balancing.

An experiment was conducted to determine if any benefit could be gained by using an interruptible *test generator.* The results of this experiment indicate that interruptible calls present no significant performance benefit because the overhead associated with interrupting the *test generator* negates the advantages of allowing processors to respond faster to *work-status* calls. However, the system presented herein uses a fairly low backtrack limit during test generation, so the maximum latency of response to *work_status* calls is limited. If a larger backtrack limit is used, a greater benefit for interruptible test generation would result.

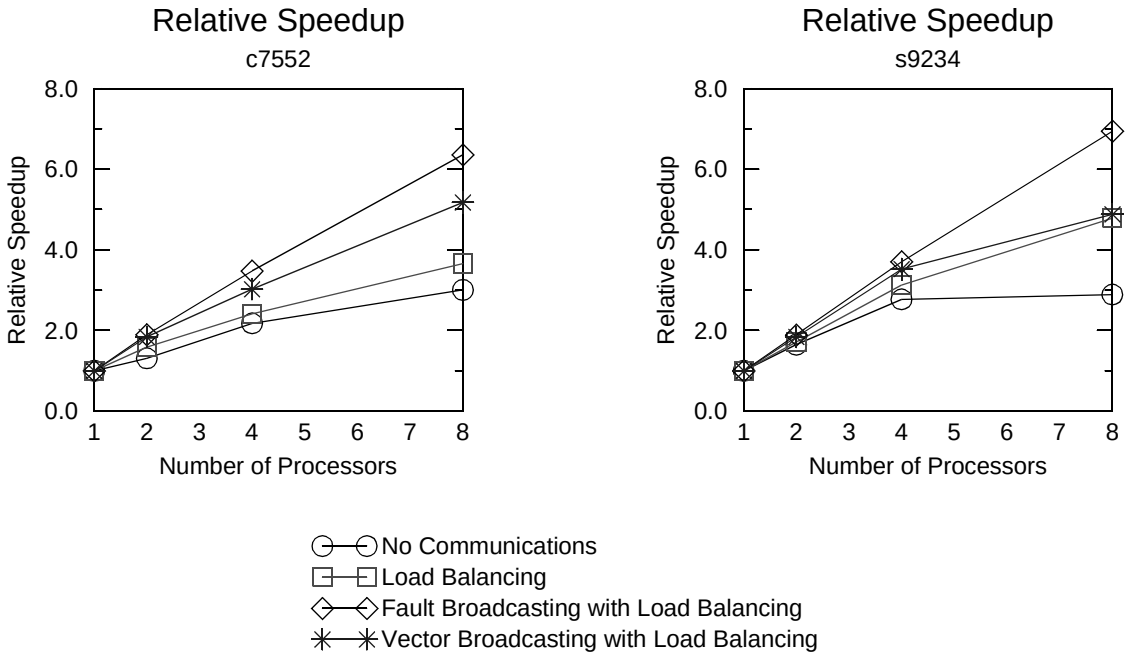

Figure 7. Relative Speedup

Test set size increases and reported fault coverage decreases as the number of processors increases if there is no communications. These results, shown in Table 2 and Figure 8, are similar for all circuits tested. This change is related to the number of fault partitions. Faults that would have been detected by the same vector had they been in the same partition must now be detected by separately generated vectors. Usually, the two generated vectors are not identical at every bit because the don't care inputs in the vector present after test generation are filled in with random '1' and '0' values for fault simulation. Therefore, these vectors are not identified as duplicates. This same partitioning issue is responsible for the decrease in reported fault coverage. Hard-to-detect faults covered by vectors generated in other fault partitions will not be marked as detected. While this approach reduces reported fault coverage, it should not effect actual fault coverage. When the PP-TGS system is run without communications, the number of partitions is equal to the number of processors

The test set and fault coverage problems are even worse with dynamic load balancing. Every time load balancing is invoked, an existing partition is divided into two new partitions. Although these new partitions have fewer faults than the original partitions, they produce the same effect as adding more processors to the network. Faults that would have been covered by the

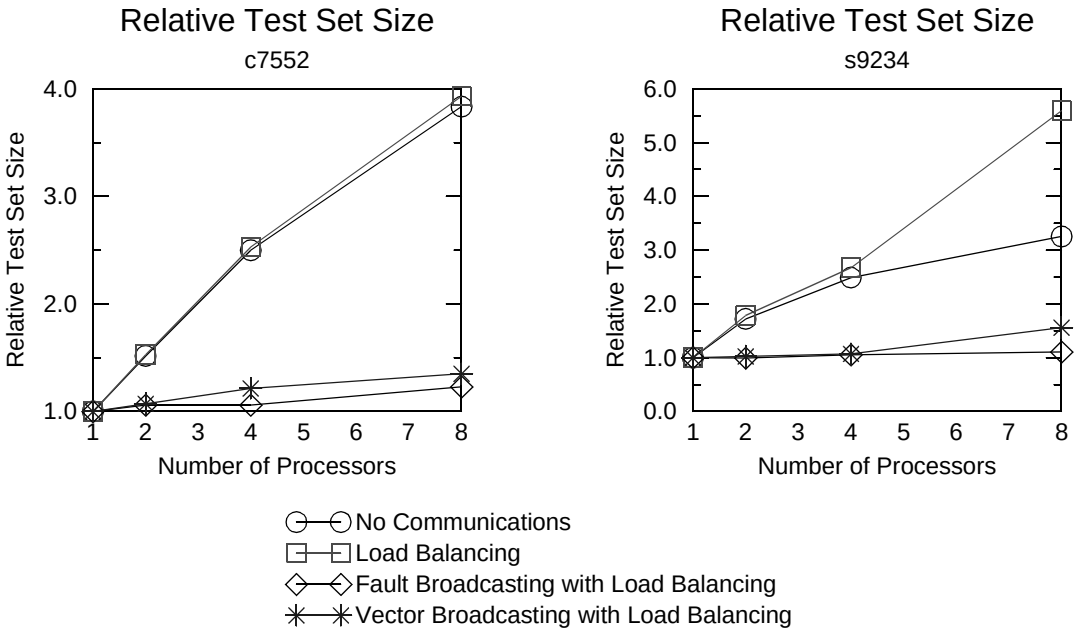

Figure 8. Relative Test Set Size

same vector had they remained in the same partition are now covered by separately generated vectors. The test set that results is larger than it would have been without load balancing. This increase in fault partitions is also responsible for the decrease in reported fault coverage.

3.2 Detected Fault and Generated Vector Broadcasting

For the multiple processor case, the increasing number of vectors generated is the result of redundant work performed to generate vectors for faults covered on remote processors. For the larger benchmark circuits, this redundant work is the major limit to the speedups achieved by the PP-TGS system without interprocessor communications. For the smaller benchmark circuits, speedups were limited by the grain size of the parallel algorithm.

Detected fault broadcasting and generated vector broadcasting are two methods of eliminating the redundant fault processing introduced by fault partitioning. They effectively merge the local partitions across the network into one inclusive global partition. Any fault covered by a generated vector is marked as covered, whether or not it is located in the partition from which the vector was generated. By reducing extraneous calls to the *test generator*, much of the redundancy that plagues fault partitioned ATPG can be eliminated. As observed in Table 2 and Figure 8, the size of the test set grows very little with respect to the uniprocessor case. The

Table 2 Test Set Size and Fault Coverage

Circuit	Number of Procs	Test Set Size				Fault Coverage (Percent)			
		No Comm	Load Bal	Fault Br Load Bal	Vect Br Load Bal	No Comm	Load Bal	Fault Br Load Bal	Vect Br Load Bal
c499	1	94				99.198			
	2	104	105	96	93	99.198	98.998	99.198	99.198
	4	136	150	96	96	97.194	95.591	99.198	99.198
	8	171	180	102	106	96.493	94.288	99.198	99.198
c1908	1	150				99.659			
	2	221	256	167	162	99.659	98.427	99.606	99.633
	4	332	344	169	175	99.606	98.637	99.633	98.659
	8	461	477	163	186	99.502	98.689	99.633	93.999
c3540	1	212				95.324			
	2	347	352	203	208	95.183	94.675	95.324	95.324
	4	531	533	211	225	94.788	94.138	95.324	95.367
	8	748	761	216	232	94.392	93.644	95.282	95.579
c7552	1	236				97.941			
	2	358	361	252	253	97.556	97.510	97.954	97.947
	4	590	597	252	287	97.596	97.431	97.914	97.927
	8	905	928	290	319	97.122	96.742	97.993	98.046
s5378	1	328				98.894			
	2	498	597	333	336	98.885	97.862	98.894	98.885
	4	683	788	332	347	98.858	97.743	98.885	98.876
	8	946	1008	353	379	98.858	97.972	98.885	98.785
s9234	1	453				93.981			
	2	781	809	451	466	93.975	93.806	93.981	93.997
	4	1129	1211	478	485	93.901	93.315	93.975	93.960
	8	1475	2535	502	708	93.759	87.724	93.991	92.205
s13207	1	554				98.920			
	2	889	993	554	570	98.920	98.407	98.920	98.913
	4	1198	1343	559	565	98.920	98.180	98.920	98.913
	8	1584	1900	579	625	98.920	97.533	98.920	98.826

increases that do occur can be attributed to the latencies involved with fault communications. The smaller test set leads to speedups which are much closer to linear, as seen in Table 1 and Figure 7. There is an additional benefit in that the reported fault coverages in Table 2 are much more accurate.

As discussed in Section 2.3.2, fault broadcasting and vector broadcasting communicate the same information to the remote partitions. Fault broadcasting fault simulates the global fault list on each processor and distributes the list of detected faults to the remote processors. Vector broadcasting distributes the reported test vectors to the various *mini-masters* prior to fault simulation. The *mini-masters* then perform fault simulation on the received vectors in order to determine the detected faults. As discussed previously, the broadcast vector can be considered an encoded fault list and the *fault simulator* a decoder. Fault broadcasting reduces redundant

processing at the cost of additional communications overhead. Table 1 and Table 2 show the comparison of fault broadcasting and vector broadcasting.

Generated vector broadcasting suffers from an important scalability problem. This problem is apparent from Table 3, which displays the total runtime and the average time spent in test generation and fault simulation for several of the tested circuits. Figure 9 depicts the average test generation and fault simulation times for s9234. The results were similar for all circuits. For fault broadcasting, fault simulation times decrease proportionally as the number of processors increases. However, with vector broadcasting, fault simulation times actually increase slightly. Because the processors spend a lot of time fault simulating the vector broadcasting calls, the latency with which each processor receives and processes the fault information increases. Therefore, faults cannot be quickly marked as detected and test generation will be performed on some faults unnecessarily, increasing the time spent in the *test generator.* With vector broadcasting, every vector generated during ATPG must be simulated by every *fault simulator.* Since the number of generated vectors slowly increases with increasing processors, fault simulation times on each processor increase as well. This situation places an absolute limit on runtime improvements that can be achieved with vector broadcasting, no matter how many processors or how fast a *test generator* is used. The additional communications overhead incurred by fault broadcasting is much less than the redundant fault simulation time required by vector broadcasting.

Table 3 Broadcasting Method Runtimes (Seconds)

Circuit	Number of Procs	Total Runtime		Average Time per Proc in FSIM		Average Time per Proc in TGEN	
		Fault Br	Vector Br	Fault Br	Vector Br	Fault Br	Vector Br
c7552	1	510.665	510.665	16.616	16.616	486.964	486.964
	2	269.066	277.620	10.671	19.208	247.513	246.855
	4	147.077	168.626	6.054	22.394	130.182	130.057
	8	80.279	98.591	4.119	23.883	64.768	66.609
s9234	1	1772.982	1772.982	31.666	31.666	1713.374	1713.374
	2	938.655	962.313	18.727	35.546	895.645	899.983
	4	477.894	503.178	10.761	36.259	447.692	447.882
	8	255.293	363.497	6.367	51.870	224.828	289.215
s13207	1	607.378	607.378	47.250	47.250	537.851	537.851
	2	341.721	368.881	28.396	54.121	289.625	289.051
	4	178.891	213.179	15.255	53.157	143.381	142.132
	8	103.756	147.102	8.565	58.914	72.262	74.342

In the PP-TGS system, the *fault simulator* is much faster than the *test generator.* This is the worst case scenario when comparing fault broadcasting to vector broadcasting. If the *test*

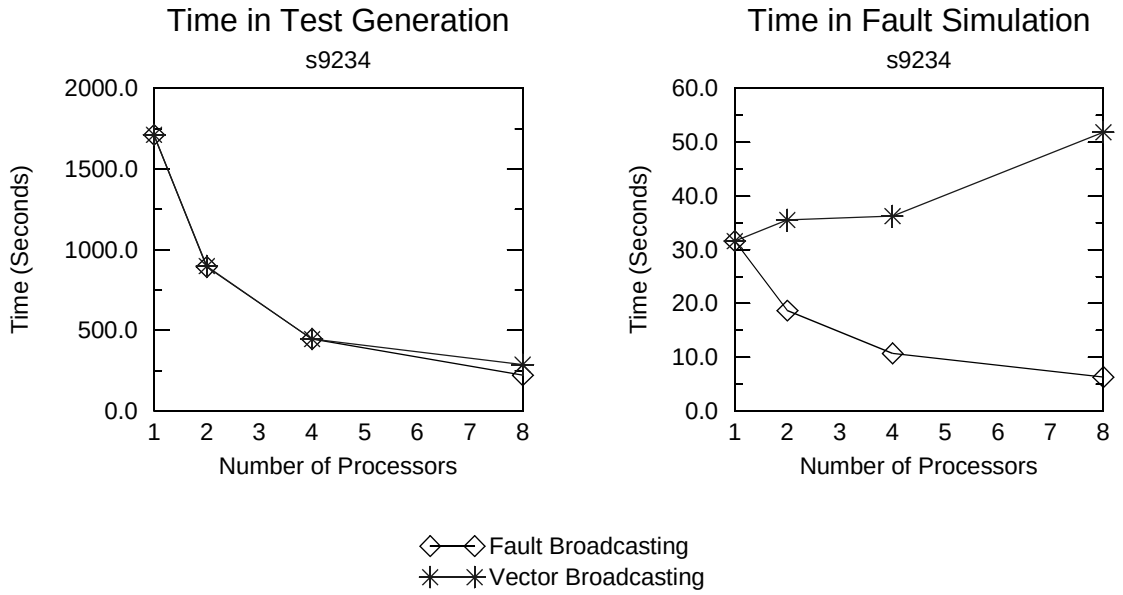

Figure 9. Average Processor Time in Test Generation and Fault Simulation

generator were comparatively faster, then vector broadcasting would exact an even greater performance penalty. Therefore, fault broadcasting is more efficient than vector broadcasting for fault partitioned ATPG.

Both fault broadcasting and vector broadcasting have communications scalability problems that will show up in larger networks. Bottlenecks can develop when several processors all attempt to broadcast information at once because communication is order N^2, where N is the number of processors in the network. If every processor simultaneously attempts a broadcast, N(N-1) fault packets will all contend for network access. Methods must be developed to address this problem.

IV. Analysis of Detected Fault Broadcasting on System Performance

In order to determine the efficiency of the detected fault broadcasting technique on various system parameters such as fault simulation time, test generation time, and communications latency, a set of analytical models was developed. The methodology presented was based upon the work presented in [5]. This set of models was used to analyze the costs associated with the various components of the PP-TGS algorithm. The number of undetected faults left after t test vectors have been generated can be modeled as $F_o e^{-kt/T_1}$, where F_o and k are constants that are defined to minimize the error introduced by the model and T_1 is the number of tests required for

maximum fault coverage [5]. This relation is illustrated in Figure 10 and is used in the analytical models to predict the behavior of the parallel ATPG system.

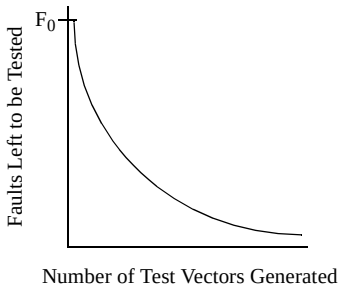

Figure 10. Model of undetected faults [5].

4.1 Analysis of PODEM Complexity

A detailed analysis of the complexity of PODEM demonstrated that the time for test generation can be determined from the number of backtrace, backtrack, imply, and objective operations required to generate a test [31]. Unfortunately, these numbers are difficult to estimate *apriori*. However, experimental evidence demonstrates that with a carefully chosen set of random faults that require multiple backtraces, backtracks, objectives and implies, the average time spent in each step of the PODEM model can be represented as the average time spent in test generation for the sample set [31]. In actuality, two constants can be estimated, κ_{succ} the approximate cost per successful test generation pass, and κ_{abort} the approximate cost per aborted test generation pass. The predicted cost of PODEM is therefore:

$$Cost_{TG} = \Upsilon_{succ}\kappa_{succ} + \Upsilon_{abort}\kappa_{abort}, \quad (4\text{-}1)$$

where Υ_{succ} is the number of successes and Υ_{abort} the number of aborts for the sample set.

4.2 Analysis of FSIM

FSIM uses a single fault propagation algorithm that simulates a fault for a given vector until it is detected or determined undetectable for the current vector. First, a good value simulation is performed for the vector. The time required for the good value simulation is proportional to the number of gates in the circuit. FSIM then simulates each fault in the fault list against the current test vector. Easily detected faults have long propagation zones but will be detected after the simulation of relatively few test vectors. Hard-to-detect and redundant faults tend to have short propagation zones but require the simulation of more vectors for detection. The result is that the

total number of calculations required per fault is fairly constant over the entire test generation process [27].

The time required to perform fault simulation can be modeled as a linear combination of the time required to perform good value simulations for every vector, μPG, and the total simulation time for every fault, λF.

$$Cost_{FS} = \mu PG + \lambda F$$

where

μ = average time to evaluate a gate
P = number of patterns to be simulated
G = number of gates in circuit
λ = average total calculation time per fault
F = number of faults in circuit

(4-2)

Since μG and λF are constant for a given circuit, the time required for fault simulation is a linear function of P, the number of patterns applied. This formula can be extended to an N processor system where P_m is the number of vectors for the *mth* processor. Since each processor performs fault simulation on the global fault list, λF is done by all processors. Equation (4-3) determines the average per processor cost in time of fault simulation:

$$Cost_{FS} = \frac{\sum_{m=1}^{N} (\mu P_m G)}{N} + \lambda F \quad . \tag{4-3}$$

4.3 Analysis of Dynamic Load Balancing

Whenever a processor becomes idle, it invokes *work_status*, which queries the other processors in the system to determine if they have work to share. Every processor, except for the processor requesting work, responds using *rpt_work_status*. The idle processor selects the first processor responding with available work to minimize its idle time. Because processes are non-interruptible in this implementation, there is a variable delay associated in an idle processor's wait for a response to its query for work. A queried processor typically would be performing ATPG, and must complete both test generation and fault simulation before responding. The amount of time that an idle processor must wait for a response can be modeled by assuming that each processor runs a fixed time loop with random starting times. Using the average time required for both test generation and for fault simulation as the loop time, the time required for a busy

processor to respond to an idle processor can be modeled using a uniform distribution as shown in Figure 11, where τ is the average ATPG time per fault. The distribution function is:

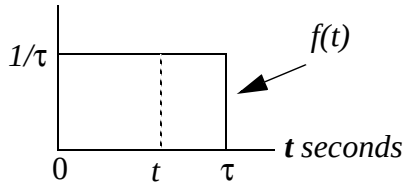

Figure 11. Uniform Distribution representing the *atpg* method

$$f(t) = \begin{cases} \frac{1}{\tau} & 0 \le t \le \tau \\ 0 & otherwise \end{cases}. \tag{4-4}$$

Whenever a single processor is queried, the probability that its response time will be less than or equal to some time *t* is:

$$p\{\boldsymbol{t} \le t\} = \int_0^t f(t)dt = \frac{1}{\tau}\int_0^t dt = \frac{t}{\tau}. \tag{4-5}$$

Similarly, the probability that the response time will be greater than *t* seconds is:

$$q\{\boldsymbol{t} > t\} = 1 - p\{\boldsymbol{t} \le t\} = 1 - \frac{t}{\tau}. \tag{4-6}$$

When two or more processors are queried for work, the likelihood that any of the processors' response time will be greater than *t* seconds is:

$$Q\{\boldsymbol{t} > t\} = q_1\{\boldsymbol{t} > t\} \times q_2\{\boldsymbol{t} > t\} \times \ldots \times q_z\{\boldsymbol{t} > t\}. \tag{4-7}$$

Since the response time for each queried processor is uniformly distributed, as shown in Figure 11, the probability that a processor will be unable to respond before some time *t* is equivalent across all processors. Therefore, equation (4-7) can be expressed as:

$$Q\{\boldsymbol{t} > t\} = (q\{\boldsymbol{t} > t\})^z, \tag{4-8}$$

and the probability that one or more processors will respond at some arbitrary time ***t*** that is less than *t* is:

$$P\{\boldsymbol{t} \le t\} = 1 - Q\{\boldsymbol{t} > t\} = 1 - \left(1 - \frac{t}{\tau}\right)^z. \tag{4-9}$$

The expected value of the response time for the first processor with available work to an idle processor's query can be derived with the use of equation (4-9).

The expected value for the response time is:

$$E(t) = \int_0^{\tau} t f(t) dt = \int_0^{\tau} t dP(t) dt \quad , \tag{4-10}$$

and substituting the derivative of equation (4-9) in equation (4-10) yields:

$$E(t) = \int_0^{\tau} t \left(\frac{z}{\tau} \left(1 - \frac{t}{\tau} \right)^{z-1} \right) dt = \frac{\tau}{z+1}. \tag{4-11}$$

In this case, *z* can be expressed as the number of processors in the system that can respond to the processor requesting work:

$$z = N - 1, \tag{4-12}$$

where N is the total number of processors in the system. Substituting equation (4-12) into equation (4-11) yields:

$$E(t) = \frac{\tau}{N}, \tag{4-13}$$

which is the average response time that an idle process must wait before selecting a processor with which to share work.

It should be noted that this expected value representing the average response time will be optimistic for two primary reasons. First, the above analysis assumed that the number of processors available to share work is a constant. In reality, the number of processors with available work is a function of the partitioning and will decrease as the number of detected faults increase. This decrease will cause the effective value of *N* to decrease. Secondly, at the end of the ATPG process, only the *harder to detect* faults remain. Since load balancing is typically invoked towards the end of ATPG, the time spent performing an *atpg* method will typically be greater than the average time required to perform ATPG per fault. This increase in average ATPG time will cause the effective value of τ to increase.

When a processor responds with work to share, the idle processor then invokes *split_list*. The amount of time required to split the fault list between the two processors is determined by the number of faults being transmitted between an idle processor and a busy processor and the latency

associated with the size of the fault list. Therefore, the average per processor cost in time associated with dynamic load balancing is:

$$Cost_{DLB} = \left| \sum_{m=1}^{N} \left(\left\{ \frac{\tau}{N} + \Delta_{mess} \right\} \nu_m + \{ \Delta_{fl} + t_{split} \} (\nu_m - 1) \right) \right| / N \quad , \tag{4-14}$$

where τ is the average time to complete an *atpg* method, Δ_{mess} is the system latency associated with an idle processor's request for work, t_{split} is the time required to split a fault list, Δ_{fl} is the system latency associated with the passing of undetected fault lists, ν_m is the number of times that processor *m* invokes *work_status* and $\nu_m - 1$ is the number of times that processor *m* invokes *split_list*.

4.4 Analysis of Detected Fault Broadcasting

During the detected fault broadcasting process, each processor collects the list of detected faults not contained within its own fault list so that this information can be distributed to the other processors. As this information is returned from the *fault simulator* during *fault_dropping*, it is placed in a packet, and passed to the other processors through *found_faults*. Packetization of the fault information is required because the number of faults broadcast after any given fault simulation step is variable and allocation and passing of a fixed data structure that is large enough to hold the maximum number of faults to be broadcast is very inefficient.

This packetization of faults means that the latency is proportional to the number of faults to be passed among the processors, even on a network where message setup dominates message transmission time. This fact, along with the fact that the amount of detected fault information being communicated among the processors decreases exponentially as the number of generated test vectors increases, means that the cost of detected fault broadcasting decreases as the number of generated test vectors increases. The average cost in time associated with detected fault broadcasting including local fault processing time can be represented as:

$$Cost_{DFB} = \sum_{m=1}^{N} \sum_{n=1}^{T_m} [\Phi_{mn} (t_{fp} + t_{fd_m} N)] / N \quad , \tag{4-15}$$

where *N* is the number of processors in the system, T_m is the number of test vectors generated by the *mth* processor, Φ_{mn} is the number of detected faults to be broadcast by the *mth* processor for the *nth* vector generated, t_{fp} is the average communications time per fault broadcast, and t_{fd} is the time required in the destination processor for receiving and dropping the detected faults.

The value for Φ can be extrapolated from the expression $F_o e^{-kz/T_1}$ representing the test generation/fault simulation process. Since this expression represents the number of faults left to evaluate after *z* test vectors, the percentage of detected faults for a given test vector, *z*, is simply:

$$\% \text{ of detected faults} = [e^{-k(z_{mn}-1)/T_m} - e^{-kz_{mn}/T_m}] \times 100 \quad , \tag{4-16}$$

and Φ_{mn} is:

$$\Phi_{mn} = \rho_{mn} \times [e^{-k(z_{mn}-1)/T_m} - me^{-kz_{mn}/T_m}] \times \Upsilon_m \quad , \tag{4-17}$$

where ρ_{mn} is the fraction of detected faults not contained within the *mth* processor's fault list for its *nth* vector generated and Υ_m is the number of faults partitioned to the *mth* processor.

4.5 Complete Cost for the PP-TGS System

The average time for the complete PP-TGS system is:

$$Cost_{Total} = Cost_{TG} + Cost_{FS} + Cost_{DLB} + Cost_{DFB}. \tag{4-18}$$

This formula can be used to project the speedup in the multi-processor environment. The uniprocessor model involves only the costs associated with PODEM and fault simulation. Therefore, the speedup in a multi-processor environment can be represented as:

$$Speedup = \frac{[Cost_{TG} + Cost_{FS}]_{uniprocessor}}{[Cost_{TG} + Cost_{FS} + Cost_{DLB} + Cost_{DFB}]_{multi-processor}} . \tag{4-19}$$

This equation can be used to predict the speedup attainable for a particular PP-TGS system.

4.6 Model Validation

In order to demonstrate that the models developed for the PP-TGS system are accurate, the models were used to predict the results of running the system in the Mentat parallel programming environment on a network of Sun SPARC2 workstations. This network supports multiple processes and its computational load from these processes ranges from very heavy during the day to just a few processes during the early morning hours. It was during the early morning hours that the data was collected.

4.6.1 Uniprocessor Environment

Initially, the software was compiled on a single Sun SPARC2 without Mentat. The ISCAS '85 benchmark circuit used to acquire the needed constants and to validate the software models was C7552.

The complete fault list for C7552 contains 15,104 faults, and a sample set of 30 faults was used. When the PP-TGS system was run on the sample set, 12 successful tests were generated in 3.652 seconds and 3 aborts occurred in 3.948 seconds, resulting in a κ_{succ} of 0.3043 seconds and a κ_{abort} of 1.316 seconds. The fault coverage was 66.126%. Using the derivation of the exponential curve for fault simulation and the constants determined for C7552 [5], the expression e^{-kz/T_1} representing the percentage of undetected faults can be solved as follows:

$$e^{-kz/T_1} = e^{-22.552(12/T_1)} = 1 - 0.66126 ,$$

which results in $T_1 = 250$.

Unfortunately, no model exists to predict the number of expected aborts. In addition, the sample set was not effective in determining the ratio of aborts to successes. Therefore, a complete ATPG run must be conducted to determine Υ_{succ} and Υ_{abort}. The actual uniprocessor test length was 236 vectors with 357 aborts. The total time required for test generation as determined by the PODEM model was:

$$Cost_{PODEM} = 236 \times 0.3043 + 357 \times 1.3161 = 541.676 \text{ seconds,}$$

while the actual time required for test generation was 486.964 seconds.

The *fault simulator* is modeled by equation (4-2) which is linear with respect to the number of vectors simulated. The total fault simulation time was estimated from the time to simulate 60 and 90 vectors. The time per good value simulation was determined to be 0.065 seconds and the time to simulate all faults was determined to be 1.750 seconds. The total estimated time for fault simulation was:

$$Cost_{FS} = 236 \times 0.0065 + 1.750 = 17.090 \text{ seconds,}$$

while the actual time required for fault simulation was 16.616 seconds.

The total time for the uniprocessor environment as predicted by the models is 558.766 seconds, and the actual runtime for C7552 was 517.869 seconds. The difference is 40.897 seconds (7.90%).

4.6.2 Multiprocessor Environment

In order to demonstrate the applicability of the models to the PP-TGS software, the results for the ISCAS circuit C7552 on 8 processors were analyzed. A complete ATPG run is required in order to determine the number of successes and aborts in test generation and the time required for

communications. Once these parameters have been calculated, they can be used to model networks of any size.

Applying the approximation for the cost of test generation as described in equation (4-1) to the *N* processor case yields:

$$Cost_{TG} = \left(\kappa_{succ} \sum_{m=1}^{N} \Upsilon_{succ_m} + \kappa_{abort} \sum_{m=1}^{N} \Upsilon_{abort_m} \right) / N \,, \tag{4-20}$$

where Υ_{succ_m} is the number of successful vectors generated on the *mth* processor and Υ_{abort_m} and abort is the number of aborts on the *mth* processor. The number of successes and aborts can be assumed to be constant on various networks configurations. This assumption is validated by the results in Section 3.2. Equation (4-3) can be used to determine the fault simulation time.

For dynamic load balancing, it was determined that equation (4-14) could be modified for the purpose of analysis. Experimental data was obtained by instrumenting timers within the PP-TGS software. These timers were placed so that each parameter identified within the models could be measured. The collected data demonstrated that the time required by a previously idle processor to store and to sort a fault list and the time for a busy processor to split its remaining fault list was an order of magnitude less than the end-to-end communications time required to pass this information between processors. Additionally, the latency associated with passing messages among the processors to determine how many faults that a processor has left was determined to be insignificant in comparison to the other factors. As a result of these findings, these components could be ignored for the purpose of this analysis. Thus, the model for dynamic load balancing can be reduced as follows:

$$Cost_{DLB} = \left| \sum_{m=1}^{N} \Delta_{fl}(\nu_m - 1) \right| / N \,. \tag{4-21}$$

The latency associated with the transmission of a fault list packet, Δ_{fl}, can be found experimentally. It was determined in Section 3.1 that ν_m increases linearly with the number of processors.

The cost of detected fault broadcasting is dominated by the time required to broadcast the detected fault information between processors as opposed to the time required to drop the detected faults from the local fault list. Therefore, equation (4-15) can be simplified as follows for the purpose of this analysis:

$$Cost_{DFB} = \sum_{m=1}^{N} \sum_{n=1}^{I_m} [\Phi_{mn} t_{fp}] / N \,, \tag{4-22}$$

where Φ_{mn} is the number of detected faults broadcast for the *nth* vector generated by the *mth* processor, t_{fp} is the time spent communicating the detected fault information by the *mth* processor and *N* is simply the number of processors in the system. Analysis in Section 3.2 reveals that t_{fp} varies linearly with the square of the number of processors.

Recall that equation (4-17) defined the value of Φ_{mn}. This equation can be used to determine an upper bound on the number of detected faults to be broadcast. Assuming a worst case scenario that every fault returned by the *fault simulator* is broadcast, the total number of faults broadcast by each processor, Φ_m,can be approximated as:

$$\Phi_m = \sum_{n=1}^{T_m} (e^{-k(z_{mn}-1)/T_m} - e^{-kz_{mn}/T_m})\Upsilon_m = \Upsilon_m \quad . \tag{4-23}$$

Thus, the sum of Φ_m for a given processor equals the number of faults in that processor's fault list. Using this result for Φ_{mn} in equation (4-22) yields:

$$Cost_{DFB} = \sum_{m=1}^{N} [\Upsilon_m t_{fp}]/N \; . \tag{4-24}$$

For the PP-TGS system running under Mentat, the costs associated with test generation and fault simulation calculated in Section 4.6.1 are not applicable because the serial version of the code was a C++ version without any Mentat overhead. Since the message passing latency between Mentat objects is significant even when the objects reside on the same processor [32], this overhead must be included in the analysis on the speedup bounds. Therefore, the multi-processor version of the Mentat code had to be modified to run on only one processor to include the overhead associated with the communications among the *mini-master, test generator,* and *fault simulator* objects. The same set of faults used to test the non-Mentat version of the code were used. The implementation of the software with the Mentat required 3.8243 seconds for the generation of 12 successes and 4.0356 seconds for 3 aborts. κ_{succ} is 0.3187 and κ_{abort} is 1.3452. Sample fault simulation times were once again taken after 60 and 90 passes through the fault simulator. The time per good value simulation was calculated to be 0.0776 seconds and the time to simulate all faults was estimated to be 1.5826 seconds.

Test generation on C7552 with interprocessor communications resulted in a test set of 302 vectors with 336 aborts. Using the approximation for the cost of test generation as presented in equation (4-20), the predicted time required for test generation and fault simulation was:

$$Cost_{TG} = \frac{302 \times 0.3187 + 336 \times 1.3452}{8} = 68.529 \quad \text{seconds, and}$$

$$Cost_{FS} = \frac{302 \times 0.07763}{8} + 1.5826 = 4.5131 \text{ seconds.}$$

There were 33 invocations of dynamic load balancing during ATPG for the circuit, meaning that each processor invoked dynamic load balancing an average of 4.125 times. Experimental data for the showed the latency to be 1.1547 seconds per broadcast. Using the approximations for the dynamic load balancing model presented in equation (4-21), the cost of dynamic load balancing for C7552 was:

$$Cost_{DLB} = \frac{8 \times 1.1547 \times (4.125 - 1)}{8} = 1.8783 \text{ seconds.}$$

C7552 contains 15,104 faults and for 8 processors, as previously stated, 302 test vectors were generated and 14,811 faults were broadcast. Experimental data showed that the time spent processing the detected fault information on all 8 processors for C7552 was 20.508 seconds. The time required to process the received detected fault information was:

$$t_{fp} = \frac{20.508}{14811 \times (8 - 1)} = 0.198 \quad \text{milliseconds per fault.}$$

This information, t_{fp}, can be used to predict the cost of detected fault broadcasting for C7552 for *N* processors. For the analysis of C7552 on 8 processors, the experimental data for the time spent processing the detected fault information will be used. For simplicity of analysis it will be assumed that all faults are broadcast, a worst case scenario. The average cost per processor for broadcasting the detected fault information using equation (4-24) was:

$$Cost_{DFB} = \frac{8 \times 0.000198 \times 15,104}{8} = 2.988 \text{ seconds.}$$

The total runtime cost for 8 processors from equation (4-18) was:

$$Cost_{total} = Cost_{TG} + Cost_{FS} + Cost_{DLB} + Cost_{DFB,}$$

and the predicted cost was 79.638 seconds. Table 4 reveals the actual and predicted performance for C7552 and S9234 on 8 processors.

The predicted speedup for C7552 was:

$$Speedup = \frac{558.766}{79.638} = 7.016.$$

The actual speedup for C7552 on 8 processors was 6.454. Among the reasons that the actual speedup was less than estimated was the difficulty in estimating the time required for PODEM to generate tests. Additionally, the assumptions made calculating the effects of dynamic load balancing in regards to expected value of the response time among the processors also effected the predictions.

Table 4 Estimated and actual times for C7552 on 8 processors

	C7552		S9234	
	Predicted Time (Seconds)	Actual Time (Seconds)	Predicted Time (Seconds)	Actual Time (Seconds)
Test Generation	68.529	63.101	233.366	224.828
Fault Simulation	4.513	4.4005	8.065	6.367
Dynamic Load Balancing	3.608		3.853	
Fault Broadcasting	2.988		3.519	
Total	79.638	79.113	248.803	255.293

4.6.3 Predictions Using Model

As discussed previously, experimental data from the sample fault set must be used to determine the average test generation and fault simulation times. Experimental data from a full ATPG run on the parallel system must be used to determine the number of aborts, the number of load balancing invocations, and the time for processing broadcast faults.

Using data gathered from the sample test generation and fault simulation runs and the complete eight processor ATPG run, the model was used to predict the performance of the PP-TGS system on networks of four, six, twelve, and sixteen processors. The calculations performed to derive the times for the four processor model are shown below:

$$Cost_{TG} = \frac{302 \times 0.3187 + 336 \times 1.3452}{4} = 137.058 \text{ seconds, and}$$

$$Cost_{FS} = \frac{302 \times 0.07763}{4} + 1.5826 = 7.443 \text{ seconds.}$$

Given that ν_m scales linearly with the number of processors (Section 3.1):

$$\nu_{m_4} = \frac{\nu_{m_8}}{8/4} = \frac{4.125}{2} = 2.063 \text{, and}$$

$$Cost_{DLB} = \frac{4 \times 1.1547 \times (2.063 - 1)}{4} = 1.227 \text{ seconds.}$$

Finally, given that t_{fp} scales linearly with the square of the number of processors (Section 3.2):

$$t_{fp_4} = \frac{t_{fp_8}}{(8/4)^2} = \frac{0.000198}{4} = 0.000050 \text{ , and}$$

$$Cost_{DFB} = \frac{4 \times 0.000050 \times 15,104}{4} = 0.748 \text{ seconds.}$$

Table 5 and Table 6 report the results of the model predictions for the C7552 and S9234 circuits. Figure 12 displays the results graphically. The predictions for the four and six processor networks were computed and compared to the actual run times. As noted earlier, the total number

of vectors and aborts generated by the ATPG software could be held constant. The data shows that the predictions were quite accurate. For C7552, the error was less than 1 percent for the four processor network and less than 4 percent for the six processor network. Therefore, this model can accurately be used to determine the effects of varying the number of processors in a parallel ATPG system.

Table 5 Predicted Performance of PP-TGS System for C7552

	4 Procs		6 Procs		12 Procs	16 Procs
	Pred. Time (Seconds)	Actual Time (Seconds)	Pred. Time (Seconds)	Actual Time (Seconds)	Pred. Time (Seconds)	Pred. Time (Seconds)
Test Generation	137.058	130.182	91.372	87.692	45.686	34.264
Fault Simulation	7.443	6.054	5.489	5.070	3.536	3.048
Dynamic Load Bal.	1.227		2.417		5.991	8.371
Fault Broadcasting	0.748		1.683		6.732	11.952
Total	146.476	147.077	100.961	103.652	61.945	57.635
Predicted Speedup	3.815		5.534		9.020	9.694

Table 6 Predicted Performance of PP-TGS System for S9234

	4 Procs		6 Procs		12 Procs	16 Procs
	Pred. Time (Seconds)	Actual Time (Seconds)	Pred. Time (Seconds)	Actual Time (Seconds)	Pred. Time (Seconds)	Pred. Time (Seconds)
Test Generation	466.732	447.692	311.154	319.370	155.577	116.683
Fault Simulation	14.947	10.761	10.359	9.341	5.771	4.624
Dynamic Load Bal.	1.926		2.889		5.779	7.706
Fault Broadcasting	0.879		1.979		7.919	14.079
Total	484.484	477.894	326.381	346.568	175.046	143.092
Predicted Speedup	3.509		5.209		9.714	11.883

The model can also be used to investigate the effect of varying different parameters of the PP-TGS system. For example, Figure 12 also shows the predicted behavior of the PP-TGS system for C7552 if the *test generator* was twice as fast. The model indicates that communication overhead limits the speedup attainable as the number of networked processors grows. It can be observed that communications accounts for over a quarter of the time required by the sixteen processor system. Future research will be needed to address the issue of efficient interprocessor communication over large networks, using the model to investigate what changes would be most beneficial.

V. Conclusions

As VLSI circuits become ever larger, the test generation problem becomes more important and more difficult. Parallel processing is an effective technique to help address this problem.

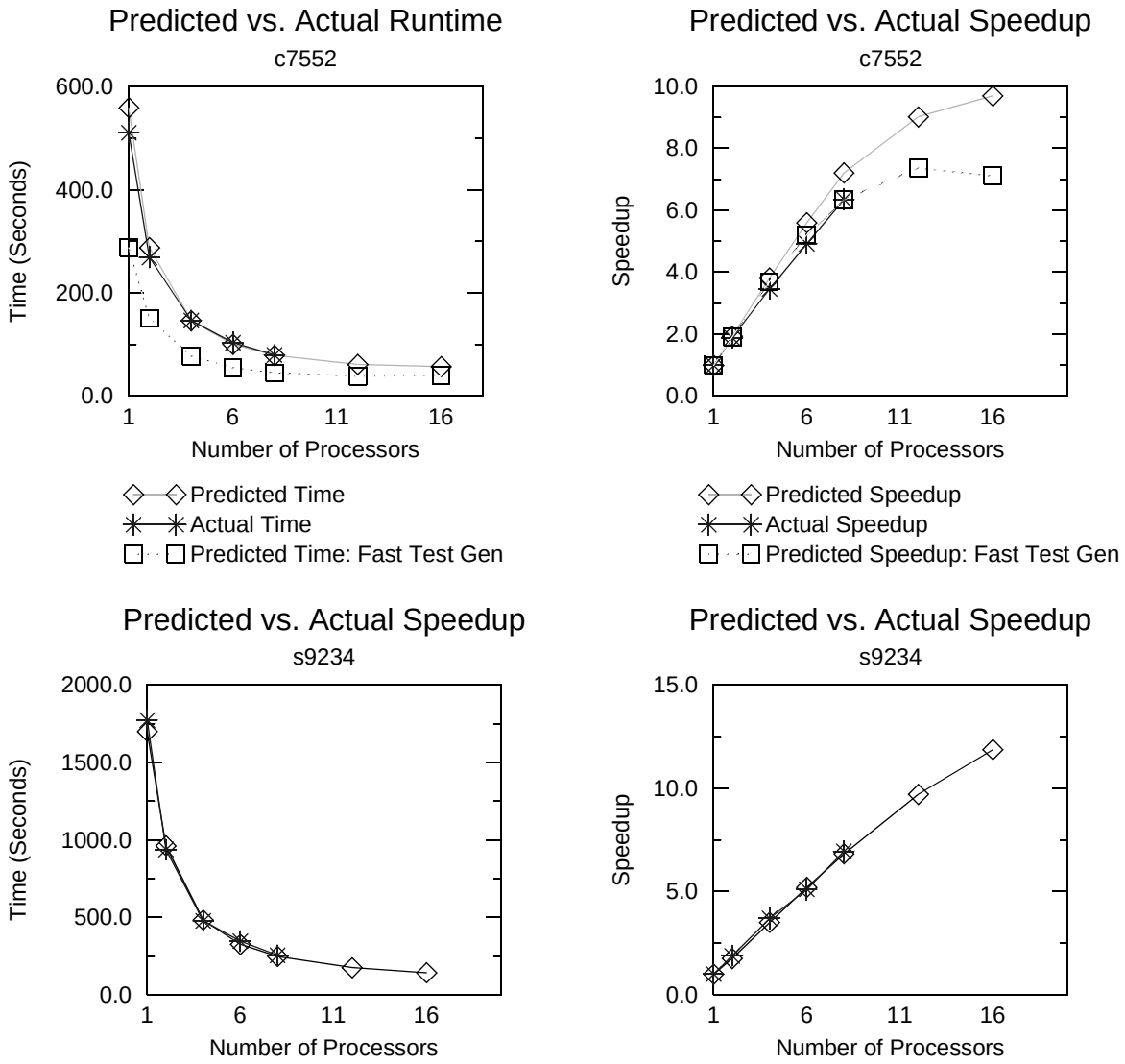

Figure 12. Predicated vs. Actual Performance of PP-TGS System

However, the average digital designer does not yet have access to dedicated parallel processing hardware. This paper presented a refinement of an existing parallelization technique for test generation that is aimed at a network of workstations environment. The goal was to show that effective speedups and reasonable test sets could be achieved in an environment to which almost every digital designer has access. Further, it was shown that this technique is generic and simple to implement given the currently available parallel programming environments like Mentat. The techniques described herein can be applied to any test generation system regardless of the ATPG or fault simulation algorithm used.

The PP-TGS system is based upon fault partitioning but it includes enhancements designed to address the drawbacks of static fault partitioning, namely poor processor load balancing, and poor speedups and increasing test set sizes because of redundant test vector generation. The load balancing issue was addressed by adding a dynamic load balancing scheme similar to that used in other systems of this type. The issue of redundant vector generation and its effect on speedups and test set size was addressed by adding detected fault broadcasting. Detected fault broadcasting was shown to be an effective enhancement to fault partitioned parallel test generation. It significantly increases speedups, produces smaller test sets, and does not require careful tuning to achieve these results as required by other techniques such as test vector broadcasting.

Analytical models were developed to determine the cost of the various parts of the PP-TGS system and predict the effects of varying network size and algorithm speed on system performance. The models were verified using runtimes for some of the benchmark circuits. The models demonstrated that communications latency restricted the maximum speedup attainable on a network of workstations platform. Future work is planned to evaluate the necessity of algorithmic fault partitioning when using fault broadcasting during parallel ATPG.

References

[1] Levitt, Marc, E., *ASIC Testing Upgraded*, **IEEE Spectrum**, pp. 26-29, May, 1992.

[2] S. Patil, P. Banerjee, *A Parallel Branch-and-Bound Algorithm for Test Generation*, **Proceedings of the 26th ACM/IEEE Design Automation Conference,** June 1989, pp. 339-334.

[3] R. H. Klenke, R. D. Williams, J. H. Aylor, *Parallel Processing Techniques for Automatic Test Pattern Generation*, **IEEE Computer**, pp. 71-84, January 1990.

[4] S. Patil, P. Banerjee, *Fault Partitioning Issues in an Integrated Parallel Test Generation/Fault Simulation Environment*, **IEEE International Test Conference**, September 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] H. Fujiwara, T. Inoue, *Optimum Granularity of Test Generation in a Distributed System*, **IEEE Transactions on Computer-Aided Design**, Vol. 9, No. 8, August 1990, pp. 885-892.

[7] 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.

[8] S. J. Chandra, J. H. Patil, *Test Generation in a Parallel Processing Environment*, **IEEE International Conference on Computer Design**, 1988, pp. 11-14.

[9] J. Sienicki, M. Bushnell, P. Agrawal, V. Agrawal, *An Asynchronous Algorithm for Sequential Circuit Test Generation on a Network of Workstations*, **IEEE International Conference on VLSI Design**, 1995, pp. 37-42.

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

[11] S. J. Chandra, J. H. Patil, *Experimental Evaluation of Testability Measures for Test Generation*, **IEEE Transactions on Computer-Aided Design**, Vol 8, No. 1, January 1989, pp. 93-97.

[12] S. Patil, P. Banerjee, *A Parallel Branch and Bound Algorithm for Test Generation*, **IEEE Transactions on Computer-Aided Design**, Vol. 9, No. 3, March 1990, pp. 313-322.

[13] B. Ramkumar and P. Banerjee, *Portable Parallel Test Generation for Sequential Circuits*, **International Conference on Computer-Aided Design**, 1992, pp. 220-223.

[14] G. A. Kramer, *Employing Massive Parallelism in Digital ATPG Algorithms*, **IEEE International Test Conference**, 1983, pp. 108-114.

[15] F. Hirose, K. Takayama, and N. Kawato, *A Method to Generate Tests for Combinational Logic Circuits Using an Ultra High Speed Logic Simulator*, **Proceedings of the 1988 IEEE International Test Conference**, September 1988, pp. 102-107.

[16] 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.

[17] S. Patil and P. Banerjee, *Fault Partitioning Issues in an Integrated Parallel Test Generation/Fault Simulation Environment*, **IEEE International Test Conference**, pp. 718-726, 1989.

[18] R. H. Klenke, L. M. Kaufman, J. H. Aylor, R. Waxman, P. Narayan, *Workstation Based Parallel ATPG*, **IEEE International Test Conference**, October, 1993.

[19] P. Agrawal, V. D. Agrawal, and J. Villoldo, *Sequential Circuit Test Generation on a Distributed System*, **Proceedings of the 30th ACM/IEEE Design Automation Conference**, pp. 107-111, June 1993.

[20] A. S. Grimshaw, *Easy to use Object Oriented Parallel Programming with MENTAT*, **IEEE Computer**, pp. 39-51, May, 1993.

[21] A. S. Grimshaw, E. C. Loyot, Jr., J. B. Weissman, *Mentat Programming Language (MPL) Reference Manual*, **Computer Science Technical Report No. TR91-32**, University of Virginia, 1991.

[22] A. S. Grimshaw, *The Mentat Run-Time System: Support for Medium Grain Parallel Computation*, **Proceedings of the Fifth Distributed Memory Computing Conference**, pp. 1064-1073, April 12, 1990.

[23] G. A. Geist, V. S. Sunderam, *Network-Based Concurrent Computing on the PVM System*, **Concurrency: Practice & Experience**, Vol. 4, No. 4, pp. 293-311, June 1992.

[24] B. Ramkumar and P. Banerjee, *ProperCAD: A Portable Object-oriented Parallel Environment for VLSI CAD*, **IEEE Transaction on Computer-Aided Design**, Vol. 13, pp. 829-842, July 1994.

[25] P. Goel, *An implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits*, **IEEE Transactions on Computers**, Vol. C-30, No. 3, pp. 215-222, March 1981.

[26] H. K. Lee and D. S. Ha, *An Efficient, Forward Fault Simulation Algorithm Based on the Parallel Pattern Single Fault Propagation*, **IEEE International Test Conference**, pp. 946-955, 1991.

[27] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlensa, et. al., *Fault Simulation for Structured VLSI*, **VLSI Systems Design**, pp. 20-32, December 1985.

[28] M. H. Schulz, M. E. Trischler, T. M. Sarfert, *SOCRATES: A Highly Efficient Automatic Test Pattern Generation System*, **IEEE International Test Conference**, pp. 1016-1026, September, 1987.

[29] J. A. Waicukauski, P. A. Shupe, D. J. Giramma, A. Matin, *ATPG for Ultra-Large Structured Designs*, **Proceedings of the 1990 IEEE International Test Conference**, pp. 44-51, September, 1990.

[30] Berlez, Franc, Hideo Fujiwara, *A Neural Netlist of Ten Combinational Benchmark Circuits and a Target Translator in FORTRAN*, **IEEE International Symposium on Circuits and Systems, Special Session on ATPG**, June 1985.

[31] R. H. Klenke, *An Investigation of Topologically Partitioned Parallel ATPG*, Ph.D. Dissertation, University of Virginia, January, 1993.

[32] A. S. Grimshaw, D. Mack, T. Strayer, *MMPS: Portable Message Passing Support for Parallel Computing*, **Proceedings of the Fifth Distributed Memory Computing Conference**, pp. 1064-1073, 1990.