Parallel Genetic Algorithms for Simulation-Based Sequential Circuit Test Generation

Dilip Krishnaswamy **Michael S. Hsiao** **Vikram Saxena**
Elizabeth M. Rudnick **Janak H. Patel**
Coordinated Science Laboratory, University of Illinois, 1308 West Main St., Urbana, IL 61801

Prithviraj Banerjee
4386 Technological Institute, Northwestern University, 2145 Sheridan Road, Evanston, IL 60208

Abstract

The problem of test generation belongs to the class of NP-complete problems and it is becoming more and more difficult as the complexity of VLSI circuits increases, and as long as execution times pose an additional problem. Parallel implementations can potentially provide significant speedups while retaining good quality results. In this paper, we present three parallel genetic algorithms for simulation-based sequential circuit test generation. Simulation-based test generators are more capable of handling the constraints of complex design features than deterministic test generators. The three parallel genetic algorithm implementations are portable and scalable over a wide range of distributed and shared memory MIMD machines. Significant speedups were obtained, and fault coverages were similar to and occasionally better than those obtained using a sequential genetic algorithm, due to the parallel search strategies adopted.

1 Introduction

Much research has been done in the area of sequential circuit test generation using both deterministic and simulation-based algorithms. In a typical deterministic algorithm, each target fault is excited and the fault effects are propagated to a primary output (PO); the required state is then justified using reverse time processing. Backtracing is a critical step and is used to determine the component input values required to obtain a particular output value. However, handling components other than simple gates is difficult because of the backtracing step. In a simulation-based approach, processing occurs in the forward direction only, and no backtracing is required. Therefore, complex design features and constraints are handled more easily [2]. Candidate tests are generated, and a logic or fault simulator is used to select the best test to apply in a given time frame. Tests are usually targeted toward detecting several faults simultaneously.

This research was supported in part by the Semiconductor Research Corporation under Contract SRC 95-DP-109 and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273 administered by the Army Research Office. We are also grateful to the National Center for Supercomputing Applications for providing us access to their Thinking Machines CM-5.

Genetic algorithms (GAs) have been used effectively for simulation-based test generation [3, 4, 5, 6, 7]. Experience has shown that GA-based automatic test pattern generation (ATPG) performs better on data-dominant circuits, while deterministic approaches perform better on highly sequential, control-dominant circuits [5]. However, the GA-based approach is better able to handle design and tester constraints. In this paper, we present three new parallel GAs for simulation-based sequential circuit ATPG. These parallel algorithms have been implemented using the ProperCAD II library [8], which provides a portable, parallel object-oriented platform for the development of parallel algorithms for VLSI CAD applications. The ProperCAD II library is a C++ library, built around the Actor paradigm of concurrent object-oriented computing [14]. It is portable across a variety of architectures. Supported architectures include distributed memory multicomputers, such as the Intel Paragon, Thinking Machines CM-5, and the IBM SP-2; shared memory multiprocessors, such as the SUN 4/690/MP, the SUN-SparcServer 1000E, and the SGI Challenge; and networks of workstations.

The first algorithm, ProperGATEST1 is a parallel version of the sequential algorithm which produces the same result as the sequential algorithm. The second algorithm, ProperGATEST2, uses a parallel search strategy where each processor executes the sequential genetic algorithm with a different seed, and uses migration to share information between processors. The third algorithm, ProperGATEST3, is a subpopulation based version of ProperGATEST2, where subpopulations are distributed across processors and information is migrated from one processor to another. Significant speedups have been observed for all three algorithms.

We begin with a discussion of the implementation of the sequential GA-based test generator in Section 2. In Section 3, we present three new parallel genetic algorithms for simulation-based sequential circuit test generation, and the three parallel genetic algorithms for test generation are discussed in detail. Results are presented and the various algorithms are compared in Section 4. Section 5 concludes the paper.

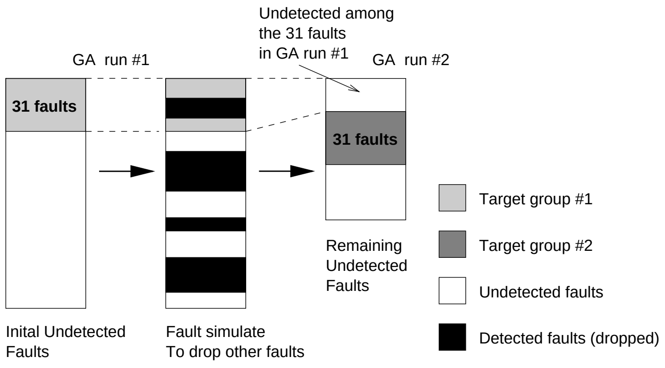
Undetected among the 31 faults in GA run #1
GA run #1
GA run #2
31 faults
31 faults
Target group #1
Target group #2
Undetected faults
Detected faults (dropped)
Remaining Undetected Faults
Inital Undetected Faults
Fault simulate To drop other faults

Figure 1: Fault grouping of 31 faults.

2 Sequential GA-Based Test Generation

The sequential implementation of the test generator uses a GA similar to the simple GA described by Goldberg [11]. The GA contains a population of *strings*, or individuals, in which each individual represents a sequence of test vectors. The population size used is a function of the string length. Each individual has an associated *fitness*, which measures the quality of the corresponding test sequence in terms of faults detected and other metrics. A sequential circuit fault simulator is used for the fitness evaluation. The population is initialized with random strings, or possibly test sequences provided by a previous run of the GA. The evolutionary processes of *selection*, *crossover*, and *mutation* are used to generate an entirely new population from the existing population, and evolution from one generation to the next is continued until little or no improvement is made in the fitness of the population or a maximum number of generations is reached.

2.1 Sequential Implementation

Processing in the sequential GA-based test generator is divided into three stages. Only the fitness function changes in each stage. In each of the three stages, the GA targets all faults in the fault list in groups of 31 faults, traversing the fault list several times until little or no more improvement is made. Between consecutive GA runs, useful vectors from the best sequence are added to the test set, and the two best sequences from the previous GA run are used to seed the population in the next GA run. The remaining individuals in the population are initialized with random test sequences. The test generation algorithm is given below.

```
While there is improvement in this stage
    While there is improvement in the GA
        For all undetected faults, in groups of 31 faults
        target-faults = next 31 undetected faults
        best-sequence = GA-test-generate (target-faults)
        fault-simulate (best-sequence)
        seed the next GA population (best 2 sequences)
        compute improvement;
    compute overall improvement
```

The GA population evolves over several generations until no more improvements in fitness values are obtained or the maximum of eight generations is reached. The population size is a function of the string length, and the string length is equal to the number of primary inputs in the circuit multiplied by the test sequence length. A multiple of the sequential depth of the circuit is used as the test sequence length, where the sequential depth is defined as the minimum number of flip-flops in a path between the primary inputs and the farthest gate. The test sequence length is set equal to the sequential depth in the first stage, two times the sequential depth in the second stage, and four times the sequential depth in the third stage. Also, the population size is doubled in the third stage to further expand the search space.

Each individual in the population is simulated using a group of 31 faults, for the purpose of evaluating the fitness of the individual. The reason 31 faults are chosen is to maximize the execution speed. Fault simulation on the entire fault list is very costly, and previous work has shown that small fault samples give good quality results as well [5]. The good circuit evaluation requires only one bit position out of the 32 available positions in a computer word, and the remaining positions can be used for faulty circuit simulations. Thus, groups of 31 faults are used as fault samples in the current work.

Figure 1 shows how faults are grouped for test generation. Initially the first 31 undetected faults are used as the fault group targeted for test generation. All individuals in the GA population target the same group of 31 faults, and the individuals evolve in successive generations until no more improvements in fitness values are obtained. Then a full-scale fault simulation is performed, using the entire list of undetected faults and the best test sequence found, and all detected faults are dropped from the fault list. An improved version of PROOFS [10] was used as the fault simulator. For the next GA run, the next group of 31 undetected faults is obtained, starting at the position in the fault list where the previous fault group left off. Only the good portion of the best test sequence is added to the test set to keep the test set compact.

2.2 Fitness function

The three stages of the test generator target different goals, and their corresponding fitness functions are different. In recent work by Wah et. al. [12] and Pomeranz and Reddy [13], the results have indicated that fault effect propagation should out-weigh fault detection when the easier faults have been detected, and maximization of state visitation increases fault coverage. Thus, these parameters are included for consideration when computing fitness values. The parameters that affect the fitness are as follows:
P1: Number of faults in the given fault group detected,
P2: Number of new states visited,
P3: Number of flip-flops that carry fault effects at the end of simulation.
All three parameters are evaluated but given different weights in the fitness computation for the three stages:
Stage 1: $fitness = 0.9 * P1 + 0.1 * P3$
Stage 2: $fitness = 0.1 * P1 + 0.45 * P2 + 0.45 * P3$
Stage 3: $fitness = 0.5 * P1 + 0.4 * P2 + 0.1 * P3$

In the first stage, the aim is to detect as many faults as possible with short sequences and minimal time. Thus, more weight is given to the number of faults detected. In the second stage, the goal is to maximize visitation of new states and fault effect propagation to state flip-flops. In the final stage, the focus is shifted once again to target the remainder of the faults that have been hard to detect in the prior two stages. Therefore, the fitness evaluation weights fault detections and new state identifications more heavily.

2.3 Fault Redundancy Identification

A simulation-based test generation approach such as a GA cannot identify redundant faults. Hence although a high fault coverage may be achieved, the efficiency of the test generator is no better than the fault coverage. Hence once the GA has finished generating test vectors, one should run a redundancy-identifying algorithm to identify the redundant faults in the list of undetected faults that remain. This will improve the efficiency of the overall test generation process. A deterministic test generator such as HITEC [1] can be used to accomplish this. HITEC targets each fault in a fault list one at a time. Usually, most redundant faults can be identified in a very short amount of time compared to the actual test generation process. So by assigning a short time-out of 1 sec per fault, one can identify most of the redundant faults among the faults that remain in the fault list after the GA has finished test generation. In principle, one could have used any program which would accomplish the task of redundancy identification. HITEC was used for the purpose, as a parallel version of HITEC, properHITEC [9], was already available.

Hence the overall approach that has been adopted in our approach is to use a simulation-based approach using genetic algorithms to get good fault coverage and small test sizes, followed by a short redundancy identification step to improve the efficiency of the test generator.

3 Parallel Genetic Algorithms for Simulation-Based Testing

Previous parallel approaches to test generation have focussed on parallelizing deterministic algorithms for testing. Excellent reviews on these approaches can be found in [15, 16]. A review of parallel genetic algorithms can be found in [18]. In GATTO [17], a parallel genetic algorithm was presented which is similar to ProperGATEST1 and the differences between these two algorithms will be highlighted in the discussion that follows. We have implemented parallelization with data decompostion, migration-based parallel GA and the subpopulation-based GA with migration approaches. Figure 2 shows a graphical outline of the approaches used in each of the algorithms for 4 processors.

3.1 ProperGATEST1: Parallelization using data decompostion

In this approach, we parallelize the loop which evaluates the fitness of all individuals in a given population, as illustrated in Figure 2(a). Each processor maintains its own copy of the entire population. The work of evaluating the fitness is equally and statically distributed over the processors. The fault list is the same on all processors. Hence, if we have N

individuals and P processors, we assign $\frac{N}{P}$ individuals to each processor. The fitness values computed by the respective processors are communicated to a single processor, which collects this information and broadcasts it to all processors. Each processor now has the information regarding the fitness values of all individuals and can evolve the next generation and compute the best individual in the population for the current generation. After processing for the last generation is completed, the best individual is added to the test set if it detects one or more faults out of the current set of 31 faults targeted. Each processor generates its own test set, and the test sets for all processors are identical, since the random number generators for all processors are initialized with the same seed. An exhaustive fault simulation is then done by all processors using this best-fit individual.

The final results in terms of the number of faults detected and the test set generated are identical to those obtained on a uniprocessor when this algorithm is used. Therefore, there is no degradation in the quality of the solution obtained through parallelization. This approach is similar to the one used in GATTO* [17]. One major difference is that, while we target 31 faults at a time using a fault parallel approach, only one fault is targeted at a time in GATTO*. Secondly, the same fitness function is used at all times in GATTO*, which may slow down execution. Thirdly, GATTO* required an additional central processor, which was not required in our thread-based implementation.

3.2 ProperGATEST2: Parallel migration based genetic search

In this algorithm, processors interact with each other by exchanging individuals, possibly the fittest individuals, among each other, at an interval determined by the epoch for the parallel GA. The hope is that with exchange of information between processors which are exploring different parts of the search space, there will be an improvement in the overall solution (the number of faults detected will be greater, the overall test set size will be smaller, etc.). This algorithm is illustrated in Figure 2(b). We always assign the same number of individuals to each processor that are assigned in the sequential algorithm, irrespective of the number of processors. Hence, each processor is assigned the same amount of workload as in the sequential case. Each processor is now following a different search path. There is now a possibility that the processors will finish faster due to the migration of information between them. This is indeed the case in our implementation as processors converge to their results faster with the added advantage that the quality of the result is improved in certain cases.

Each processor maintains an independent fault list and proceeds completely independently, generating its own test set. Periodically, each processor transmits the individual with the best fitness to a random processor. A queue of messages is maintained on each processor to receive migrating individuals from other processors. These individuals are absorbed into the local subpopulation as it evolves. This gives rise to the possibility of obtaining P different solutions to the problem using a randomized parallel genetic search.

3.3 ProperGATEST3: Subpopulation-based GA with migration

This approach is similar to the previous approach, except that each processor starts with a population of $M = \frac{N}{P}$ individuals. This algorithm is illustrated in Figure 2(c). One can expect speedups for two reasons. Each processor works on a subpopulation, and therefore, the population size being used is smaller, and each processor individually has less work to do. Also, due to migration of fit individuals from one processor to another, each processor can detect faults faster than if they were to run independent GA's with a reduced population size.

One possible disadvantage of this approach is that, for a circuit with a relatively small population size (corresponding to a circuit with small number of primary inputs and small sequential depth), the algorithm is not very scalable. If the population size for the GA becomes too low, one can expect a degradation in the results obtained. But if the population size is large, as is the case for large circuits, then this should not be problem.

4 Experimental Results

The algorithms ProperGATEST1, ProperGATEST2, and ProperGATEST3 were implemented in the ProperCAD II environment. All implementations are portable to a wide variety of parallel platforms. All algorithms were implemented on a SUN-SparcServer 1000E, a shared memory multiprocessor with 8 processors and 512 MB of memory. In addition, the algorithm ProperGATEST1 was ported to a network of SUN-Sparc5 workstations and to the Thinking Machines CM-5, a distributed memory multicomputer with a SUN-sparc1 processor on each node and 32 MB of memory per node, to demonstrate the portability of the implementations.

Each of the parallel genetic algorithms on completion was followed by a phase of properHITEC, executed in parallel, to drop any redundant faults among the faults left undetected by the genetic algorithm. A short time-out of 1 second was assigned per fault for this purpose. All three parallel genetic algorithms execute in the same fashion on a single processor. Table 1 shows the performance of the genetic algorithm running on a single processor. The circuits taken are part of the ISCAS89 benchmark suite. A collapsed fault list for each circuit was taken as the initial fault list. All execution times are in seconds. "Fault Det" refer to faults that were detected by the genetic algorithm and "Faults Red" refers to the redundant faults identified by ProperHITEC. It can be seen that the amount of time spent in the ProperHITEC phase is a small fraction of the overall execution time and it helps in improving the efficiency of the overall ATPG process.

Table 2 shows the performance of ProperGATEST1 on eight processors. It can be seen that the results remain unchanged from the uniprocessor run and that execution times have been reduced significantly.

As a comparison, Table 3 shows the results for the same circuits when a parallel deterministic algorithm such as ProperHITEC is used in isolation[9]. One must look at the total execution time, the overall test set size in terms of the number of vectors, the fault coverage and the fault efficiency when comparing different algorithms. One can observe that ProperGATEST1 performs better for some circuits such as s298, s344, s349 and s5378 while ProperHITEC performs better for circuits such as s1494 and s35932. We therefore believe that a hybrid strategy needs to be adopted in practice.

Table 4 shows the performance of ProperGATEST2 on eight processors. The speedups obtained with this algorithm were purely due to the migration of fit individuals from one processor to another. Nevertheless, good speedups were obtained. This algorithm takes longer to execute than ProperGATEST1 but provides better quality results, in terms of the fault coverage obtained, more often than not, when compared to ProperGATEST1.

The algorithm ProperGATEST3, whose performance can be seen in Table 5 provides excellent execution times, but the test set sizes are larger than the previous algorithms. The quality of the result in terms of the fault coverage and efficiency is usually worse than that given by ProperGATEST2 and better than ProperGATEST1 in most cases. For ProperGATEST3, the GA population size is inversely proportional to the number of processors. Hence, for a large number of processors, this algorithm may not always perform well.

We will now report results of portability of our parallel algorithms on various parallel platforms. For this part of the study we did not use the redundancy identification phase of ProperHITEC hence the fault coverage and efficiency numbers are not reported; we only report execution times and speedups on multiple processors. Owing to lack of space, we will only report results of one of the parallel algorithms, ProperGATEST1. Results of the other parallel algorithms are similar and can be obtained from the authors directly. Results for the algorithm ProperGATEST1, running on the SUN-SparcServer 1000E, are shown in Table 6. The numbers of faults detected in the parallel and sequential cases are the same. Hence the quality of the result was not affected, and significant speedups were obtained on 8 processors.

Table 7 shows the results obtained for the algorithm ProperGATEST1 on a network of 6 SUN-SPARC5 workstations. Table 8 shows results for the same algorithm on the Connection Machine-5 (CM5). These results demonstrate the performance and portability of the code on various parallel platforms. The quality of the results was unaffected, and good speedups were obtained.

5 Conclusion

In this paper, we have presented three parallel genetic algorithms, ProperGATEST1, ProperGATEST2, and ProperGATEST3, for simulation-based sequential test generation. The results obtained here using these algorithms are in general better than those obtained using GATTO* [17]. ProperGATEST1 is a good algorithm in general, as it provides significant speedups without degradation in the quality of the result. It exploits the parallelism available corresponding to the evaluation of fitness values of the individuals in a population. ProperGATEST2 exploits the search parallelism available through parallel randomized genetic search with migration of information. It attempts to improve the quality of the results and is a highly scalable implementation, as it can run over any number of processors irrespective of the population size. However, the speedups for this algorithm grow at a slower rate compared to the other algorithms. ProperGATEST3 provides a dual degree of parallelism by using a subpopulation-based GA approach to reduce the work-load among the processors and by exploiting the benefits of the randomized migration strategy used. However, there is a likelihood of degradation in the quality of the result for this algorithm for very large number of processors due to a corresponding de-

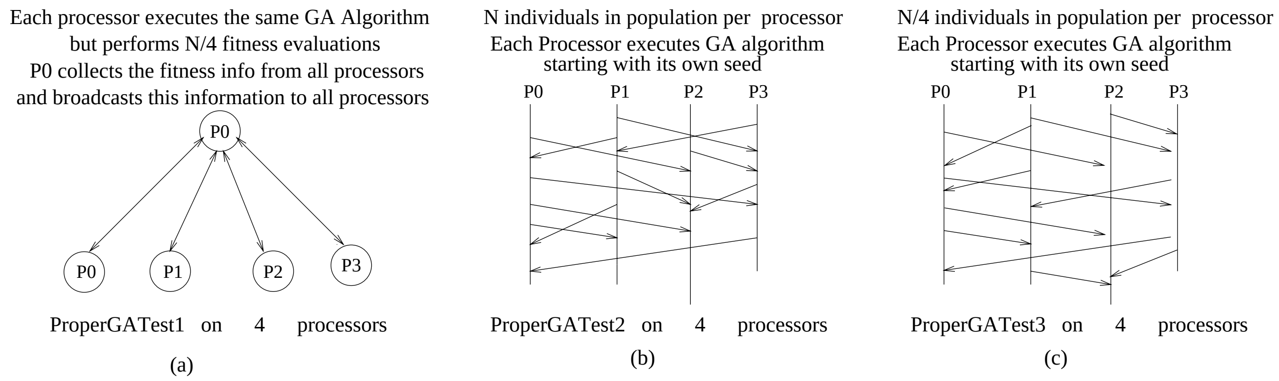
Each processor executes the same GA Algorithm
but performs N/4 fitness evaluations
P0 collects the fitness info from all processors
and broadcasts this information to all processors
P0
P0
P1
P2
P3
ProperGATest1 on 4 processors
(a)
N individuals in population per processor
Each Processor executes GA algorithm
starting with its own seed
P0
P1
P2
P3
ProperGATest2 on 4 processors
(b)
N/4 individuals in population per processor
Each Processor executes GA algorithm
starting with its own seed
P0
P1
P2
P3
ProperGATest3 on 4 processors
(c)

Figure 2: Comparison of three proposed parallel algorithms.

crease in the population size.

We have thus presented three new parallel genetic algorithms. All three parallel genetic algorithms are comparable in performance to each other and to existing parallel deterministic approaches such as ProperHITEC [9]. The algorithms are scalable and also easily amenable to parallelization. We have also demonstrated the need for a parallel redundancy identification program to improve the efficiency of simulation-based approaches such as the parallel genetic algorithms presented in this paper, to improve the efficiency of the overall test generation process.

References

[1] T. M. Niermann and J. H. Patel, "HITEC: A test generation package for sequential circuits," *Proceedings of the European Conference on Design Automation*, pp. 214-218, 1991.

[2] E. M. Rudnick and J. H. Patel, "A genetic approach to test application time reduction for full scan and partial scan circuits," *Proc. Eighth Int. Conf. VLSI Design*, pp. 288-293, 1995.

[3] D. G. Saab, Y. G. Saab, and J. A. Abraham, "CRIS: A test cultivation program for sequential VLSI circuits," *Proc. Int. Conf. Computer-Aided Design*, pp. 216-219, 1992.

[4] M. Srinivas and L. M. Patnaik, "A simulation-based test generation scheme using genetic algorithms," *Proc. Int. Conf. VLSI Design*, pp. 132-135, 1993.

[5] E. M. Rudnick, J. G. Holm, D. G. Saab, and J. H. Patel, "Application of simple genetic algorithms to sequential circuit test generation," *Proc. European Design and Test Conf.*, pp. 40-45, 1994.

[6] E. M. Rudnick, J. H. Patel, G. S. Greenstein, and T. M. Niermann, "Sequential circuit test generation in a genetic algorithm framework," *Proc. Design Automation Conf.,* pp. 698-704, 1994.

[7] P. Prinetto, M. Rebaudengo, and M. Sonza Reorda, "An automatic test pattern generator for large sequential circuits based on genetic algorithms," *Proc. Int. Test Conf.*, pp. 240-249, 1994.

[8] S. Parkes, J. A. Chandy, and P. Banerjee, "A Library-based approach to portable, parallel, object-oriented programming: Interface, implementation and application," *Proc. Supercomputing'94*, pp. 69-78, 1994.

[9] S. Parkes, P. Banerjee and J. Patel, "ProperHITEC: A portable, parallel, object-oriented approach to sequential test generation," *Proc. Design Automation Conf.*, pp. 717-721, 1994.

[10] T. M. Niermann, W. -T. Cheng, and J. H. Patel, "PROOFS: A fast, memory-efficient sequential circuit fault simulat or," *IEEE Trans. Computer-Aided Design*, pp. 198-207, February 1992.

[11] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning,* Reading, MA: Addison-Wesley, 1989.

[12] B. W. Wah, A. Ieumwananonthachai, L. C. Chu, and A. Aizawa, "Rational scheduling of experiments and generalization in genetics-based learning," *IEEE Trans. Knowledge and Data Engineering*, 1995.

[13] I. Pomeranz and S. M. Reddy, "LOCSTEP: A logic simulation based test generation procedure," *Proc. Fault Tolerant Computing Symp.*, June 1995.

[14] G. A. Agha, *Actors: A model of concurrent computation in distributed systems*, Cambridge MA: The MIT Press, 1986.

[15] P. Banerjee, *Parallel Algorithms for VLSI Computer-aided Design Applications*, Prentice Hall, Englewoods-Cliffs, NJ, 1994, pp. 477-590.

[16] R. Klenke, R. D. Williams and J. Aylor, "Parallel Processing Techniques for Automatic Test Pattern Generation," *IEEE Computer*, Jan. 1992, pp. 71-84.

[17] F. Corno, P. Prinetto, M. Rebaudengo, M. Sonza Reorda, and E. Veiluva, "A portable ATPG tool for parallel and distributed systems," *Proc. VLSI Test Symp.*, pp. 29-34.

[18] P. Adamidis, "Review of parallel genetic algorithms bibiliography," Techn. Report, Aristotle University of Thessaloniki, Thessaloniki, Greece, 1994.

Circuit	Faults	ProperGATEST1		ProperHITEC		Total	Test	Fault	Fault
		Faults Det	Time	Faults Red	Time	Time	Set Size	Coverage	Efficiency
s298	308	265	67.70	22	25.01	92.70	166	0.86	0.93
s344	342	329	60.08	8	5.45	65.53	55	0.96	0.995
s349	350	335	63.53	10	5.50	69.03	77	0.96	0.995
s1196	1242	1236	457.20	3	0.1	457.3	516	0.995	0.997
s1238	1355	1281	776.94	72	1.46	778.4	576	0.945	0.999
s1494	1506	1297	1378	20	107	1485	286	0.86	0.87
s5378	4603	3243	23864	114	1037	24901	468	0.70	0.73
s35932	39094	34456	43578	3984	1155	44733	158	0.88	0.983

Table 1: ProperGATest1: Uniprocessor run on a SUN-SPARCServer1000E shared-memory multiprocessor

Circuit	Faults	ProperGATEST1		ProperHITEC		Total	Test	Fault	Fault
		Faults Det	Time	Faults Red	Time	Time	Set Size	Coverage	Efficiency
s298	308	265	15.9	22	8.4	24.3	166	0.86	0.93
s344	342	329	13.9	8	3.5	17.4	55	0.96	0.995
s349	350	335	13.5	10	1.3	14.8	77	0.96	0.995
s1196	1242	1236	86.8	3	0.1	86.9	516	0.995	0.997
s1238	1355	1281	117.2	72	0.4	117.6	576	0.945	0.999
s1494	1506	1297	241.4	20	24	265.4	286	0.86	0.87
s5378	4603	3243	3804	114	198	4002	468	0.70	0.73
s35932	39094	34456	6328	3984	261	6589	158	0.88	0.983

Table 2: ProperGATest1: Eight processor run on a SUN-SPARCServer1000E shared-memory multiprocessor

Circuit	Faults	One Processor ProperHITEC				Eight Processor ProperHITEC			
		Faults Det	Faults Red	Test Size	Time	Faults Det	Faults Red	Test Size	Time
s298	308	265	22	306	974	262	22	322	268
s344	342	324	11	121	331	318	11	103	79
s349	350	335	10	137	182	335	10	124	46
s1196	1242	1239	3	453	24	1238	3	471	18
s1238	1355	1283	72	375	7.5	1281	70	373	21
s1494	1506	1453	50	1249	345	1451	50	1374	89
s5378	4603	3155	189	844	26141	3130	188	1002	7492
s35932	39094	35090	3984	207	1354	35043	3984	303	381

Table 3: ProperHITEC: One and eight processor runs on a SUN-SPARCServer1000E shared-memory multiprocessor

Circuit	Faults	ProperGATEST2		ProperHITEC		Total	Test	Fault	Fault
		Faults Det	Time	Faults Red	Time	Time	Set Size	Coverage	Efficiency
s298	308	265	21.9	22	8.3	30.2	156	0.86	0.93
s344	342	329	19.7	8	3.6	23.3	55	0.96	0.995
s349	350	335	23.5	10	1.3	24.8	70	0.96	0.995
s1196	1242	1235	154.9	3	0.1	155	548	0.994	0.997
s1238	1355	1282	362.1	72	0.4	362.5	592	0.946	0.999
s1494	1506	1365	382.5	20	26	408.5	278	0.91	0.92
s5378	4603	3261	4619	112	194	4813	548	0.71	0.73
s35932	39094	35006	8137	3984	214	8351	226	0.896	0.997

Table 4: ProperGATest2: Eight processor run on a SUN-SPARCServer1000E shared-memory multiprocessor

Circuit	Faults	ProperGATEST3		ProperHITEC		Total	Test	Fault	Fault
		Faults Det	Time	Faults Red	Time	Time	Set Size	Coverage	Efficiency
s298	308	264	10.2	22	8.4	19.6	184	0.857	0.928
s344	342	329	4.7	8	3.6	8.0	75	0.96	0.995
s349	350	335	4.8	10	1.3	6.1	90	0.96	0.995
s1196	1242	1235	96.2	3	0.1	96.3	612	0.994	0.997
s1238	1355	1271	113.9	72	0.4	114.3	586	0.94	0.991
s1494	1506	1348	71.2	20	26	77.2	312	0.895	0.91
s5378	4603	3214	3227	114	233	3460	542	0.70	0.72
s35932	39094	34832	5924	3984	238	6162	284	0.890	0.992

Table 5: ProperGATest3: Eight processor run on a SUN-SPARCServer1000E shared-memory multiprocessor

Circuit	Faults	Faults Det	1 processor	2 processors		4 processors		8 processors	
			Time	Time	Speedup	Time	Speedup	Time	Speedup
s298	308	265	67.70	35.47	1.91	21.92	3.09	15.95	4.24
s344	342	329	60.08	31.75	1.89	18.15	3.31	13.97	4.30
s349	350	335	63.53	35.08	1.87	19.92	3.8	13.45	4.72
s1196	1242	1236	457.20	231.03	1.98	117.38	3.89	86.85	5.26
s1238	1355	1281	776.94	455.46	1.71	242.40	3.21	117.18	6.63
s1494	1506	1297	1378.09	719.37	1.92	398.34	3.46	241.38	5.71

Table 6: ProperGATest1: Runtime in seconds and speedup on SUN-SPARCServer 1000E (shared memory multiprocessor)

Circuit	Faults	Faults Det	1 processor	2 processors		4 processors		6 processors	
			Time	Time	Speedup	Time	Speedup	Time	Speedup
s298	308	265	119.84	60.53	1.98	35.99	3.33	33.16	3.61
s344	342	329	78.57	42.76	1.84	36.26	2.17	32.47	2.42
s349	350	335	76.35	42.38	1.82	28.98	2.63	21.95	3.48
s1196	1242	1226	680.05	540.59	1.26	377.61	1.80	298.68	2.28
s1238	1355	1281	2267.79	1455.86	1.56	747.99	3.03	504.13	4.50
s1494	1506	1304	1887.49	984.29	1.92	615.16	3.07	433.05	4.36

Table 7: ProperGATest1: Runtime in seconds and speedup on a network of 6 workstations (SUN-SPARC5)

Circuit	Faults	Faults Det	1 proc	2 proc		4 proc		8 proc		16 proc	
			Time	Time	Spd	Time	Spd	Time	Spd	Time	Spd
s298	308	265	368.47	193.86	1.90	100.40	3.67	62.06	5.94	51.68	7.12
s344	342	329	232.83	123.83	1.88	69.56	3.35	41.21	5.65	34.26	6.80
s349	350	335	271.69	147.19	1.85	76.82	3.54	48.65	5.58	37.04	7.33
s1196	1242	1226	820.68	652.84	1.26	439.54	1.87	247.87	3.31	236.48	3.47
s1238	1355	1281	2860.66	1845.04	1.55	739.18	3.87	556.00	5.51	499.69	5.72
s1494	1506	1304	3307.48	1702.13	1.94	824.46	4.01	532.95	6.21	399.85	8.27

Table 8: ProperGATest1: Runtime in seconds and speedup (Spd) on the CM-5 (distributed memory multicomputer)