To appear: 1995 International Conference on Computer Design, October 2-4, 1995, Austin, TX

A Parallel Algorithm for Fault Simulation based on PROOFS

Steven Parkes
parkes@sierravista.com
Sierra Vista Research, Inc.
236 N Santa Cruz Ave, Suite 210
Los Gatos, CA 95030-7258

Prithviraj Banerjee   Janak Patel
{banerjee,patel}@crhc.uiuc.edu
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois
1308 West Main St.
Urbana, IL 61801-2307

Abstract

Fault simulation for sequential circuits numbers among the highly compute-intensive tasks in the integrated circuit design process. In the quest for rapid design turn-around, parallelization has been proposed to speed fault simulation. In this paper, we introduce `ProperPROOFS`, a parallel extension of the `PROOFS` fault simulation package. `ProperPROOFS` exploits parallelism based on fault partitioning, incorporating static and dynamic partitioning schemes and a new asynchronous and distributed method of fault redistribution. We present results for circuits in the ISCAS-89 benchmark set across several parallel architectures. A detailed evaluation of results provides new insight into the use of fault partitioning to parallelize high performance serial fault simulation applications.

1 Introduction

Fault simulation is used to determine the set of faults in a circuit that are covered by a set of test vectors. It is typically used during automatic test pattern generation in order to minimize test generation costs and in a stand-alone context to evaluate the efficacy of existing vector sets. The products of fault simulation are lists of detected and undetected faults. On large circuits with large test sets, fault simulation can take hours and in some cases days.

Several algorithms have been proposed for parallelizing fault simulation. The general conclusion of previous work is that fault list partitioning is adequate and scalable to a large number of processors. However, most reported work uses as metric the ratio of the run time of the parallel application on several processors to that on a single processor; comparisons with state-of-the-art serial algorithms and absolute run times are uncommon. Most published results are also limited in extent, generally including a small number of experiments on small circuits.

In this paper, we present `ProperPROOFS`, a parallel extension of the `PROOFS` package for fault simulation [1]. `ProperPROOFS` is based on fault list partitioning, incorporating both static and dynamic partitioning. Implemented in `ProperPROOFS` is a new asynchronous and distributed method for dynamic fault partitioning. Parallelism in `ProperPROOFS` is expressed using the actor model of concurrent object-oriented programming [2] as implemented in the ProperCAD II library [3, 4]. `ProperPROOFS` runs on all architectures supported by the library including shared memory, distributed memory, and hybrid architectures.

This research was supported in part by the Semiconductor Research Corporation under contract 94-DP-109 and the Advanced Research Projects Agency under contract DAA-H04-94-G-0273. We gratefully acknowledge additional support from the Argonne National Laboratory, the San Diego Supercomputer Center, and the National Center for Supercomputer Applications.

2 Fault simulation

2.1 Serial fault simulation

There are several significantly different approaches to fault simulation. While most work in parallel fault simulation have disregarded the architectural differences of the serial fault simulator, these characteristics can have a significant impact on the performance of parallel applications.

The task of fault simulation is to determine for each fault in a given list whether the simulation of the faulty and fault-free circuits differs in any primary output. While a fault simulator can be built in a straightforward manner from any logic simulator, the resulting performance would be low due to significant duplicated computation. As a result, various techniques to reduce duplicate computation have been developed. These approaches may be classified by the manner in which they compute and store the good and faulty circuit states. The most popular approaches are concurrent, deductive, differential, and bit-parallel fault simulation.

Concurrent In concurrent fault simulation [5], node values for all faulty machines are computed from good circuit values and previous faulty circuit values generated by previous vectors; good circuit simulation only occurs once for each vector. This technique results in fewer simulation events at the expense of an increase in storage requirements.

Deductive Deductive fault simulation [6] differs from concurrent fault simulation in that values for faulty circuits are only computed and stored where they differ from the good circuit value. Storage requirements are reduced at the expense of computational cost.

Differential In differential fault simulation [7], faulty circuit node values are computed and stored only where the value differs from that of the previously processed fault. While this reduces memory requirements, in introduces a dependence between successive faults in the fault list, thereby precluding fault dropping.

Bit-parallel In bit-parallel[1] fault simulation [8], all bits in the processor word are used; groups of faults of size equal to the processor word size are computed concurrently. While on current machines, this potentially leads to a 32- or 64-X speed up, in practice smaller improvements are observed due to an increase in the number of events processed by the simulator.

2.2 Parallel fault simulation

A number of approaches have been proposed for the parallelization of fault simulation for both combinational and sequential circuits [9].

Fault partitioning In fault partitioning, the fault list is divided among available processors, each processor simulating all faults in its partition independently. Due to variations in event activity, static partitioning has generally been considered ineffective in achieving speed ups for all but a small number of processors.

Several implementations based on fault partitioning have been reported. Most implement dynamic partitioning via a master-slave model in which a single processor is dedicated to maintaining the list of undetected faults. Slaves request fault groups from the master, perform simulation, and report back results. Duba et al. [10] report a parallel implementation of CHIEFS for which speedups from five to six on eight processors were achieved. Markas et al. [11] report a distributed algorithm for which speedups ranged from two to six on eight workstations. In both case, results are presented for a small number of fairly small circuits.

[1] In the literature, what will be called *bit-parallel* is often referred to as *parallel fault simulation*. Due to the ambiguity of the term 'parallel' in this context, we will prefer the phrase bit-parallel to describe this class of techniques.

Circuit partitioning The main alternative to fault partitioning is circuit partitioning, in which the circuit being simulated is partitioned among available processors. Circuit-partitioned fault simulation is effectively a variant of parallel logic simulation, itself a difficult problem [12].

Fault simulation based on circuit partitioning has been reported by Mueller-Thuns et al. [13] and Nelson [14] for vector-synchronous implementations on message passing machines. Ghosh [15] presents an implementation based on asynchronous logic simulation techniques that, while novel, falls short of achieving high processor efficiency. In [16], Patil et al. present a circuit-partitioned approach applicable to shared memory machines machines that incorporates techniques from parallel logic simulation.

3 `ProperPROOFS`

`ProperPROOFS` is based on the serial fault simulator `PROOFS`. It uses fault partitioning to achieve concurrency and can be run with either static or dynamic partitioning. Parallelism in `ProperPROOFS` is expressed using primitives from the ProperCAD II library.

3.1 The `PROOFS` fault simulator

`PROOFS`, developed by Niermann et al. [1], combines techniques from bit-parallel, concurrent, and differential fault simulation to achieve performance exceeding that achieved by the individual techniques. A parallel fault simulator which does not retain the high performance techniques of `PROOFS` is likely to yield disappointing performance. The most significant mechanisms in `PROOFS` are:

Dynamic fault grouping `PROOFS` uses a dynamic method of grouping faults into words to perform bit-parallel fault simulation; good circuit simulation is performed separately.

Fault dropping The combination of techniques in `PROOFS` allows for fault dropping as soon as a fault is detected.

Faulty state storage For each test vector, `PROOFS` simulates the good circuit using an event-driven logic simulator, storing the values at every node for reference during faulty simulation. Only the state values for faulty circuits that are not detected and that are not equal to the good circuit values are computed and saved between vectors.

3.2 Fault partitioning

Partitioning in `ProperPROOFS` is implemented using both static and dynamic distribution. Static partitioning is performed by dividing the fault list by the number of available processors at the start of processing. When using dynamic partitioning, faults are initially statically partitioned, followed by on-demand asynchronous fault redistribution.

3.3 Asynchronous fault redistribution

Asynchronous fault redistribution occurs when a processor completes simulation of its existing fault list. That processor requests faults from another processor. When a request is received, the receiving processor either splits its fault list or forwards the request if it too has an empty fault list.

Requesting faults Each processor simulates all vectors for all faults in its current partition. At completion, the processor sends a request to another processor, selected at random, for more faults to be simulated; after sending a split request, the processor remains idle until new faults are supplied. If faults are available, they are returned via a request to simulate a new set of faults starting from a specified vector (Figure 1).

Splitting fault lists Upon receiving a request to split a fault list, the processor first determines whether it has any faults remaining to be simulated. If it does not, the request is forwarded at random to another processor (Figure 2). If the processor does have faults to simulate, it returns half to the requesting processor.

Termination detection When asynchronous fault redistribution is used, no processor maintains the global status of the fault simulation; detection of completion of the simulation must be addressed separately. `ProperPROOFS` uses quiescence detection, a mechanism by which the run time system informs the application when no significant tasks are available for execution.

3.4 Implementation

The architecture of `ProperPROOFS` is shown in Figure 3. Most classes in the parallel application are derived from a `PROOFS` class and a ProperCAD II class.

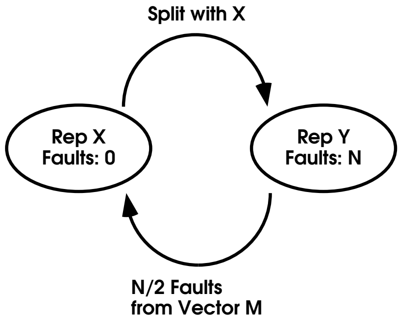

Figure 1: Split request in fault redistribution

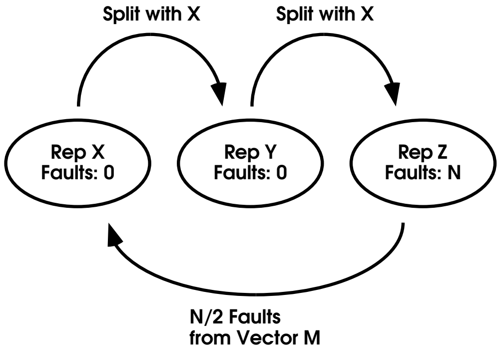

Figure 2: Forwarding of split requests

ProperCAD II `ProperPROOFS` is based on the ProperCAD II C++ class library for concurrent object-oriented programming [3, 4]. Use of the class library provides two significant advantages:

- Incremental parallelization, the ability to incrementally parallelize and reuse existing serial code
- Portability across a wide range of parallel architectures

The ProperCAD II library is based on actors [2], active objects which communicate by executing continuations, the parallel, object-oriented extrapolation of remote procedure calls. The high-level actor model obviates the need for application designers to deal with low level mutual-exclusion primitives. The actor model has been extended to include aggregates [17], collections of actors, individually called *representatives*.

The ProperCAD II library supports actor semantics via `Actor` and `Aggregate` base classes. For example, in `ProperPROOFS` the main computation aggregate class is `FaultSimulatorAggregate`, derived from the ProperCAD II `Aggregate` class and the `PROOFS` `FaultSimulator` class:

```
class FaultSimulatorAggregate :
        public Aggregate,
        public FaultSimulator { ... };
```

Incremental parallelism in `ProperPROOFS` finds expression in the limited number of actor and aggregate

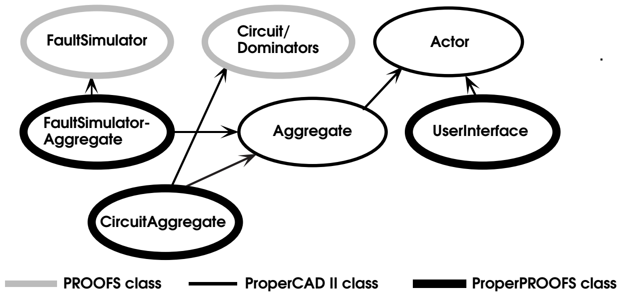

Figure 3: `ProperPROOFS` organization

classes. Code reuse is demonstrated by the use of a `PROOFS` class as a base class in the parallel application.

Communication in actor programs occurs via continuations, similar to a function pointers. For example, in `ProperPROOFS`, the `detect` member of the fault database aggregate class `FaultDataBase` is called by `FaultSimulatorAggregate` representatives when a fault is detected:

```
FaultDataBase::
        detect::Continuation cont ( faultDB );
cont( fault );
```

The first line declares a continuation `cont` which will call the member function `detect` for the object identified by `faultDB`. The second line calls the continuation with the fault index `fault` as its argument.

Rescheduling The actor model does not support preemption. It is therefore necessary to have representatives reschedule themselves, in effect polling for split requests. Without rescheduling, an actor that has faults that are available to be split would never receive a split request until it had already simulated all faults.

Rescheduling is implemented by having each representative call a method `Simulate` on itself periodically. Measurements on `PROOFS` showed that fault groups take hundreds of microseconds to compute, making rescheduling overhead insignificant.

Splitting fault lists When faults are split, several values are required to restart the simulation on the other processor:

- next vector to be simulated
- good circuit state
- pending good circuit events
- faulty circuit states

As the number of undetected faults and unsimulated vectors falls, the usefulness of splitting faults also falls. Therefore, a lower bound is placed on the number of vectors and faults that may be split below which all remaining simulation is performed without rescheduling. This keeps the frequency of redistribution low enough that the impact of communication overhead on execution time is insignificant.

4 Performance and Evaluation

4.1 Static partitioning

Results for `ProperPROOFS` without fault redistribution are shown in Table 1 for circuits from the ISCAS-89 benchmark set on an INTEL iPSC/860 hypercube. The results for `ProperPROOFS` are reported as speed up relative to the serial `PROOFS` code.

The data show that on a single processor, `ProperPROOFS` adds little overhead. Speed ups, however, vary widely. On eight processors, speed ups varied from 1.1 to 6.5. Even on the largest circuits, speed ups as low as 2.4 were observed. This is agrees with previous work indicating that static fault distribution does not scale well.

Table 1: Static distribution on iPSC/860

Circuit	PROOFS (sec)	ProperPROOFS Processors (speedup)			
		1	2	4	8
s208	9.1	0.99	1.51	2.60	4.48
s298	4.5	0.94	1.21	1.60	1.62
s344	4.4	0.89	1.01	1.05	1.10
s349	4.5	0.89	1.01	1.04	1.09
s382	10.9	0.94	1.29	1.92	2.38
s386	3.0	0.93	1.38	1.68	2.19
s400	12.2	0.95	1.36	1.91	2.45
s420	23.5	0.97	1.49	2.56	4.43
s444	14.5	0.96	1.34	2.02	2.84
s510	37.8	0.96	1.62	2.55	4.41
s526	17.4	0.96	1.45	1.87	2.52
s526n	16.8	0.96	1.41	1.88	2.54
s641	5.9	0.96	1.12	1.50	1.57
s713	6.7	0.97	1.19	1.58	1.74
s820	11.5	0.96	1.20	1.54	1.83
s832	11.7	0.95	1.22	1.55	1.88
s838	63.2	1.01	1.50	2.55	4.37
s953	101.9	1.03	1.41	2.66	3.96
s1196	9.9	0.99	1.41	1.90	2.28
s1238	11.4	0.99	1.50	2.00	2.51
s1423	46.1	0.98	1.64	2.70	4.14
s1488	21.5	0.98	1.26	1.50	1.65
s1494	21.8	0.96	1.27	1.52	1.67
s5378	150.3	0.99	1.34	2.19	3.74
s9234	458.8	1.00	1.60	2.48	2.90
s13207	1598	0.99	1.38	1.93	3.24
s15850	966.8	0.98	1.75	2.96	4.88
s35932	1124	1.01	1.39	1.90	2.43
s38417	5112	0.97	1.84	3.60	6.49

1000 random vectors

4.2 Dynamic partitioning

Tables 2 and 3 show the results of `ProperPROOFS` with dynamic redistribution on an Intel Paragon and a Sun 4/690MP.

Comparison of `PROOFS` and `ProperPROOFS` on a single processor again shows that rescheduling and other concurrency issues contribute little overhead. Paragon results show moderate speedup for many of the larger circuits, though a significant range is still observed.

The results for the Sun MP show that while speedup is moderate on two and three processors, often little improvement is achieved on four processors. The is likely due to memory contention caused by excessive forwarding of split

Table 2: Dynamic redistribution on Intel Paragon

Circuit	PROOFS (sec)	ProperPROOFS Processors (speedup)			
		1	4	16	64
s208	87.6	0.98	2.59	6.27	8.14
s298	32.5	0.94	1.30	1.44	1.44
s344	38.9	0.97	1.05	1.14	1.16
s349	39.4	0.97	1.04	1.14	1.16
s382	107.0	0.97	2.31	3.15	3.47
s386	25.2	0.94	1.44	2.80	3.02
s400	118.1	0.97	2.29	3.15	3.55
s420	230.7	0.97	2.97	7.22	10.6
s444	138.3	0.98	2.47	3.69	4.19
s510	366.2	0.98	3.34	7.85	15.1
s526n	162.0	1.00	2.43	3.51	4.12
s526	166.7	1.00	2.52	3.69	4.33
s641	50.4	0.95	1.33	1.42	1.44
s713	58.0	0.96	1.44	1.60	1.58
s820	101.3	0.98	1.65	1.95	1.95
s832	104.1	0.98	1.65	1.98	2.02
s838	619.2	0.99	3.22	7.47	11.7
s953	959.4	0.99	3.39	7.98	13.7
s1196	48.8	0.95	1.30	1.43	1.59
s1238	59.8	0.95	1.35	1.67	1.89
s1423	356.6	0.99	2.74	5.05	6.37
s1488	190.9	0.97	1.49	1.58	1.65
s1494	194.0	0.98	1.51	1.61	1.70
s5378	1094	1.01	2.37	3.35	3.53
s9234	4716	1.00	3.48	7.47	12.1
s13207	1685	0.98	3.68	9.18	16.0
s15850	1034	0.98	3.48	8.22	13.6
s35932	1105	0.99	2.27	3.04	3.21
s38417	5401	0.99	4.15	13.3	27.7
s38584	3370	0.97	3.50	7.53	15.3

s13207–s38584: 1000 random vectors
all others: 10000 random vectors

Table 3: Dynamic redistribution on Sun 4/670MP

Circuit	PROOFS (sec)	ProperPROOFS Processors (speedup)			
		1	2	3	4
s208	83.3	0.97	1.50	2.11	2.28
s298	32.3	0.99	1.09	1.12	1.12
s344	35.5	0.94	1.03	1.05	0.92
s349	36.4	0.94	1.04	1.07	0.99
s382	101.7	1.00	1.41	1.74	1.86
s386	24.1	0.89	0.99	1.14	1.29
s400	106.7	1.04	1.59	1.85	1.80
s420	246.6	1.12	1.76	2.53	2.82
s444	122.9	0.96	1.64	1.83	1.99
s510	324.8	1.06	1.82	2.14	2.57
s526n	161.7	1.13	1.88	1.91	2.24
s526	166.9	1.10	1.87	2.01	2.38
s641	41.4	0.93	0.88	0.91	0.97
s713	46.9	0.82	0.79	1.00	1.05
s820	92.7	0.99	1.24	1.29	1.43
s832	95.1	0.99	1.24	1.43	1.30
s838	571.7	1.01	1.95	2.23	2.27
s953	790.5	1.08	1.86	2.07	2.40
s1196	45.9	0.95	1.02	1.27	1.09
s1238	53.4	0.93	1.04	1.17	1.17
s1423	308.0	0.99	1.43	1.68	1.97
s1488	164.9	0.93	1.25	1.40	1.19
s1494	166.2	0.91	1.26	1.35	1.31
s5378	978.7	0.97	1.41	1.75	1.40
s9234	4353	0.97	1.53	1.81	1.77
s13207	17973	1.01	1.79	2.08	2.18
s15850	8684	0.97	1.62	1.95	1.91
s35932	1068	0.91	1.46	1.48	1.62
s38417	8270	0.97	1.41	1.53	1.48
s38584	3519	0.95	1.47	1.77	1.76

s15850–s38584: 1000 random vectors
all others: 10000 random vectors

requests. This contention is difficult to evaluate and is being studied in current work.

4.3 Scalability

To investigate issues of scalability, we studied in detail the run time dynamics of s35932 on the Paragon (Table 4) which demonstrated only a small amount of speedup over all configurations. The number of good circuit evaluations grows linearly in the number of processors because every processor must perform the good circuit simulation before performing fault simulation on its partition. Redistribution exacerbates the problem: not only must good circuit simulation be performed per processor, it must be duplicated in part when a set of faults is redistributed.

The good circuit resimulation overhead caused by fault redistribution partially and sometimes completely mitigates the advantage of dynamic redistribution (Table 5.) While dynamic redistribution generally achieves either equivalent or better results than those of the static partitioning, the incremental speedup is low.

Several possibilities exist for mitigating the extra costs implied by fault partitioning. For example, the extra simu-

Table 4: Good and faulty simulation of s35932 using dynamic redistribution

Processors	Good Events (millions)	Good Time (sec)	Faulty Events (millions)	Faulty Time (sec)
1	8.04	77.4	34.6	524.2
4	76.5	738.3	34.8	522.9
16	227	2182	35.2	538.9
64	557	5393	35.5	601.7

1000 random vectors

Table 5: Distribution comparison on iPSC/860

Circuit	PROOFS (sec)	Dynamic (speedup)		Static (speedup)	
		4	8	4	8
s208	9.1	2.38	3.86	2.60	4.48
s298	4.5	1.51	1.59	1.60	1.62
s344	4.4	1.05	1.10	1.05	1.10
s349	4.5	1.04	1.10	1.04	1.09
s382	10.9	2.11	2.54	1.92	2.38
s386	3.0	1.30	2.12	1.68	2.19
s400	12.2	2.12	2.67	1.91	2.45
s420	23.5	2.66	4.38	2.56	4.43
s444	14.5	2.32	2.95	2.02	2.84
s510	37.8	3.24	5.36	2.55	4.41
s526	17.4	2.39	2.96	1.87	2.52
s526n	16.8	2.31	2.89	1.88	2.54
s641	5.9	1.38	1.57	1.50	1.57
s713	6.7	1.51	1.73	1.58	1.74
s820	11.5	1.55	1.78	1.54	1.83
s832	11.7	1.55	1.83	1.55	1.88
s838	63.2	3.12	4.84	2.55	4.37
s953	101.9	3.35	5.64	2.66	3.96
s1196	9.9	1.72	2.08	1.90	2.28
s1238	11.4	1.78	2.27	2.00	2.51
s1423	46.1	2.90	4.37	2.70	4.14
s1488	21.5	1.51	1.63	1.50	1.65
s1494	21.8	1.56	1.68	1.52	1.67
s5378	150.3	2.75	4.30	2.19	3.74
s9234	458.8	3.50	4.79	2.48	2.90
s13207	1598	3.81	6.82	1.93	3.24
s15850	966.8	3.58	5.86	2.96	4.88
s35932	1124	2.34	2.85	1.90	2.43
s38417	5112	3.87	7.31	3.60	6.49

1000 random vectors

lations caused by load redistribution can be eliminated by saving the state of the good circuit the first time it is computed. Unfortunately, these methods generally require large amounts of storage. Eliminating the overhead of good circuit simulation is a topic of current study.

4.4 Parallelization Effort

Since fault simulator design is itself a difficult task, it is important that parallelization preserve investment in the serial application and not be excessively costly. `ProperPROOFS` implementation took approximately one man-month, with additional improvements and debugging requiring approximately one-half man-month. Table 6 gives several metrics of the `PROOFS` and `ProperPROOFS` implementations. All significant classes are shared between the parallel and serial versions. Most of the functions unique to `ProperPROOFS` are small, performing most of their computation through calls to the serial code.

Table 6: Comparison of software metrics for `PROOFS` and `ProperPROOFS`

Metric	PROOFS	Proper PROOFS	Shared
Lines of code with ';'	140	733	1981
Classes excluding Continuations	0	4	12
Continuations	0	17	0

5 Summary

In this paper we introduced `ProperPROOFS`, an approach to parallel fault simulation using dynamic and static fault partitioning to parallelize the `PROOFS` fault simulator. Results running on several architectures show that `ProperPROOFS` achieves moderate utilization for a number of difficult fault simulation problems. Detailed evaluation indicates significant parallelization barriers and identifies areas for future work.

References

[1] T. M. Niermann, W.-T. Cheng, and J. H. Patel, "PROOFS: A fast, memory efficient sequential circuit fault simulator," *IEEE Transactions on Computer Aided Design*, pp. 198–207, 1992.

[2] G. A. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. Cambridge, MA: The MIT Press, 1986.

[3] S. Parkes, J. A. Chandy, and P. Banerjee, "A library-based approach to portable, parallel, object-oriented programming: Interface, implementation, and application," in *Proceedings, Supercomputing '94*, pp. 69–78, Nov. 1994.

[4] S. M. Parkes, *A Class Library Approach to Concurrent Object-Oriented Programming with Applications to VLSI CAD*. PhD thesis, University of Illinois, Urbana, IL, Oct. 1994.

[5] E. G. Ulrich and T. Baker, "Concurrent simulation of nearly identical digital networks," *IEEE Computer*, vol. 7, pp. 39–44, Apr. 1974.

[6] D. B. Armstrong, "A deductive method for simulating faults in logic circuits," *IEEE Transactions on Computers*, vol. 21, pp. 462–471, May 1972.

[7] W.-T. Cheng and M.-L. Yu, "Differential fault simulation — a fast method using minimal memory," in *Proceedings of the 26th ACM/IEEE Design Automation Conference*, pp. 424–428, June 1989.

[8] S. Seshu, "On an improved diagnosis program," *IEEE Transactions on Electronic Computers*, vol. 14, pp. 76–79, 1965.

[9] P. Banerjee, *Parallel Algorithms for VLSI Computer-Aided Design*. Englewood Cliffs, NJ: PTR Prentice Hall, 1994.

[10] P. A. Duba, R. K. Roy, J. A. Abraham, and W. A. Rogers, "Fault simulation in a distributed environment," in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 686–691, June 1988.

[11] T. Markas, M. Royals, and N. Kanopoulos, “On distributed fault simulation,” *IEEE Computer*, vol. 7, pp. 40–52, Jan. 1990.

[12] L. Soule and T. Blank, “Parallel logic simulation on general purpose machines,” in *Proceedings of the 25th ACM/IEEE Design Automation Conference*, pp. 166–171, June 1988.

[13] R. B. Mueller-Thuns, D. G. Saab, R. F. Damiano, and J. A. Abraham, “Portable parallel logic and fault simulation,” in *Digest of Papers, International Conference on Computer-Aided Design*, pp. 506–509, Nov. 1989.

[14] J. F. Nelson, “Deductive fault simulation on hypercube multiprocessors,” in *Proceedings of the 9th AT&T Conference on Electronic Testing*, Oct. 1987.

[15] S. Ghosh, “NODIFS: A novel, distributed circuit partitioning based algorithm for fault simulation of combinational and sequential digital designs on loosely coupled parallel processors,” tech. rep., LEMS, Division of Engineering, Brown University, Providence, RI, 1991.

[16] S. Patil, P. Banerjee, and J. Patel, “Parallel test generation for sequential circuits on general purpose multiprocessors,” in *Proceedings of the 28th ACM/IEEE Design Automation Conference*, (San Fransisco, CA), June 1991.

[17] A. A. Chien, *Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs*. Cambridge, MA: The MIT Press, 1993.