PROOFS: A FAST, MEMORY EFFICIENT SEQUENTIAL CIRCUIT FAULT SIMULATOR

Thomas M. Niermann
University of Illinois
Urbana, Il. 61801

Wu-Tung Cheng
ADAS Software Inc.
San Jose, Ca. 95132

Janak H. Patel
University of Illinois
Urbana, Il. 61801

This paper describes PROOFS, a super fast fault simulator for synchronous sequential circuits. PROOFS achieves high performance by combining all the advantages of *differential fault simulation*, *single fault propagation*, and *parallel fault simulation*, while minimizing their individual disadvantages. PROOFS minimizes the memory requirements, reduces the number of events that need to be evaluated, and simplifies the complexity of the software implementation. PROOFS requires an average of one fifth the memory required for concurrent fault simulation and runs 6 to 67 times faster on the ISCAS sequential benchmarks.

1. Introduction

With the development of VLSI technologies, test sequences with very high fault coverage have become increasingly important in order to maintain acceptable field reject rates [1]. Fault simulators are used to determine which faults are detected by a test sequence. This information not only grades the quality of this test sequence but also speeds up the test generation process. After a test sequence is generated for one target fault by a time consuming test generator, a fault simulator is usually used for finding other faults that are also detected. In this manner, the number of faults which need to be attempted by a test generator can be dramatically reduced. Fault simulators are also used to find test vectors by guided search methods[2].

The *single stuck-at fault model* has been successfully used in many contemporary fault simulators. Therefore, only single stuck-at faults are considered in this paper. In fault simulation, each test pattern is run with the good machine as well as with every faulty machine, where the good machine is the fault free circuit description and a faulty machine is the circuit with one line fixed at a high voltage (a stuck-at-1 fault) or staying fixed at a low voltage (a stuck-at-0 fault). If the output responses of any one faulty machine differs from those of the good machine, the corresponding fault is said to have been detected.

This paper presents an improved fault simulation algorithm based on a combination of the parallel, concurrent and differential fault simulation algorithms. This fault simulator is shown to require much less memory while being 6.6 to 67 times faster than a traditional concurrent fault simulator.

Table 1 represents the task of fault simulation. Each column corresponds to a test vector and each row corresponds to a machine. There is one good machine and m faulty machines corresponding to different faulty lines, n test vectors, and $(m+1)n$ machine states. The task of fault simulation is to find all the primary output values of the $(m+1)n$ machine status, and determine which faulty machines have output vectors different from the good machine. There have been different fault simulation strategies developed based on different orders of filling in this table.

Table 1. The Tasks of Fault Simulation

	V_1	...	V_j	...	V_n
Good	G_1	...	G_j	...	G_n
$Fault_1$	$F_{1,1}$	...	$F_{1,j}$	...	$F_{1,n}$
$Fault_2$	$F_{2,1}$	...	$F_{2,j}$	...	$F_{2,n}$
.	.	...	.	...	.
$Fault_i$	$F_{i,1}$	...	$F_{i,j}$	...	$F_{i,n}$
$Fault_{i+1}$	$F_{i+1,1}$	...	$F_{i+1,j}$	...	$F_{i+1,n}$
.	.	...	.	...	.
$Fault_m$	$F_{m,1}$	...	$F_{m,j}$	...	$F_{m,n}$

1.1. Concurrent

Concurrent[3,4] and deductive fault[5] simulation both use a strategy of filling the table from left to right, computing all the faulty machines' values for a vector concurrently using the same previously computed information to compute a $F_{i,j}$, see Figure 1. They use the values from the previous vector for the same faulty machine $F_{i,j-1}$ that are different from the good circuit values and the current good machine values G_j. This strategy results in a low number of events, but it has two main drawbacks. First for every line in the circuit there is a large list of active faulty machines associated with it. These lists on all the lines have a large memory requirement. The second drawback is that to evaluate a gate for a fault, the faulty machine list for each of the gates inputs must be searched to check for common faulty machines among the lists. This list searching has a big time overhead. PROOFS also determines $F_{i,j}$ from $F_{i,j-1}$ and G_j, but it avoids the large memory requirement by only storing the faulty machine values at the state-nodes, such as flip-flops and latches. It also avoids the time overhead of the gate evaluation by using a technique that is as fast as logic simulation gate evaluation.

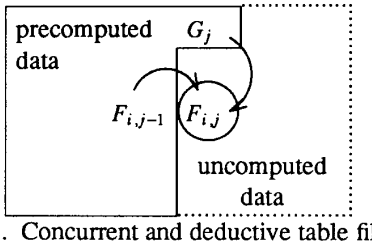

Fig. 1. Concurrent and deductive table filling

1.2. Differential

The differential simulation algorithm [6] determines $F_{i,j}$ exclusively from $F_{i-1,j}$, as shown in Figure 2. To determine if a fault is detected a count of the number of times each primary output has changed is kept. Differential simulation requires very little memory because it only stores one copy of all the line values of the circuit and the differences between adjacent faulty machines. The differential fault simulation algorithm suffers from the inability to drop detected faults easily because subsequent faulty machines rely on the differences from the dropped faulty machine. Fault dropping is easy in PROOFS since no faulty machine depends on any other faulty machine for its simulation.

1.3. Parallel

The parallel fault simulation algorithm [7] determines $F_{i,j}$ exclusively from $F_{i,j-1}$ but takes advantage of the word level parallelism of the computer to simulate 32 (for a 32 bit machine) faulty machines per pass, as in Figure 3. The parallel algorithm suffers from the repetition of the good machine simulation in every pass and also from the inability to drop detected faults. After a fault is detected, one bit space is wasted for the remainder of that pass of the simulation. The algorithm presented in this paper makes full utilization of all the bits in the parallel word for every vector by using a dynamic fault grouping strategy and avoids the resimulation of the good circuit.

1.4. Other

A different strategy is to generate each machine status from its *reference machine* status in the same column (same test vector, different machine). This strategy was first used exclusively for combinational circuits in *testdetect* [8] (which was later called *single fault propagation* [9]). Since single fault propagation always uses the good machine as the reference machine, it simulates only the fault effects of each faulty machine from the good machine. For example, in Table 1, $F_{1,j+1}$ is generated from G_{j+1} by simulating the fault site of F_1 as the initial event source. However, restoring the status of the good machine before every faulty machine simulation results in a performance overhead. Single fault propagation was further improved by Waicukauski [10] to use the full length of a computer word for a parallel simulation of several machine statuses in the same row of Table 1 (same machine with several different input vectors). This is the well-known *parallel pattern single fault propagation* technique. Furthermore, there are efficient heuristics proposed in [11-14] to trace the fault effects in combinational circuits. Therefore, the number of faulty machines which need to be simulated explicitly is very small, and thus the total overhead to restore the good machine status before every faulty machine simulation is reduced. In general, for combinational circuits, single fault propagation is faster than concurrent fault simulation [13].

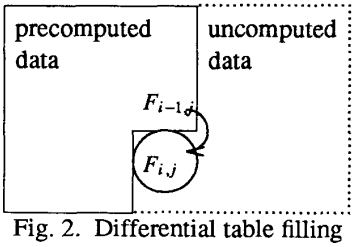

Fig. 2. Differential table filling

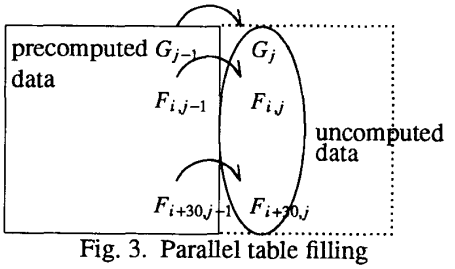

Fig. 3. Parallel table filling

Moreover, there are other approximate fault simulation approaches which trade some loss of accuracy in the results for a significant reduction in computational time, namely, *fast fault grading* [17], *critical path tracing* [18], and *statistical fault analysis* [19]. In the remainder of this paper, we will deal only with accurate fault simulation of sequential circuits.

2. PROOFS

The PROOFS fault simulation algorithm can be thought of as a hybrid of the concurrent, differential, and parallel fault simulation algorithms. It retains the advantage of fault dropping that concurrent allows while, exploiting the word level parallelism of the computer, and retaining the low memory requirement of differential. It uses a dynamic fault grouping strategy to fully utilize all the bit spaces in the computer word to simulate several faulty machine at once. This dynamic strategy avoids wasting space in the machine word for faults already detected. In addition the dynamic strategy removes faults from the parallel word in the time frames in which they are inactive. A different technique to inject faults is used to avoid a computation overhead for each evaluation. Instead of using bit masks, the circuit is changed to reflect the faults that are active.

The overall algorithm of PROOFS is shown in Figure 4. It consists of a main loop which reads in the next input vector, does the good machine evaluation, and then for each fault grouping does the faulty evaluation. To evaluate a fault group, first the group-id is incremented so that the faulty values of other machines are not used in the evaluation of this machine. Next, the 32 faulty machines to be included in the fault group are decided upon. The faults are then injected into the circuit and the node values for the state-nodes from the previous input vector are inserted into the faulty line values. The faulty machines in this fault group are evaluated, and the state-node values are stored for the next vector. We will now go through each of these steps and the data structures used in detail.

The line value data structures used in PROOFS are shown in Figure 5. First there is a one dimensional array that stores the good circuit value of every line in the circuit. There is also an array which keeps the value of every line in the faulty circuit. Accompanying the faulty circuit value is a group-id for each value. Each value consists of two 32 bit words V0 and V1 where each bit is used to store a different faulty machine's value. A four valued

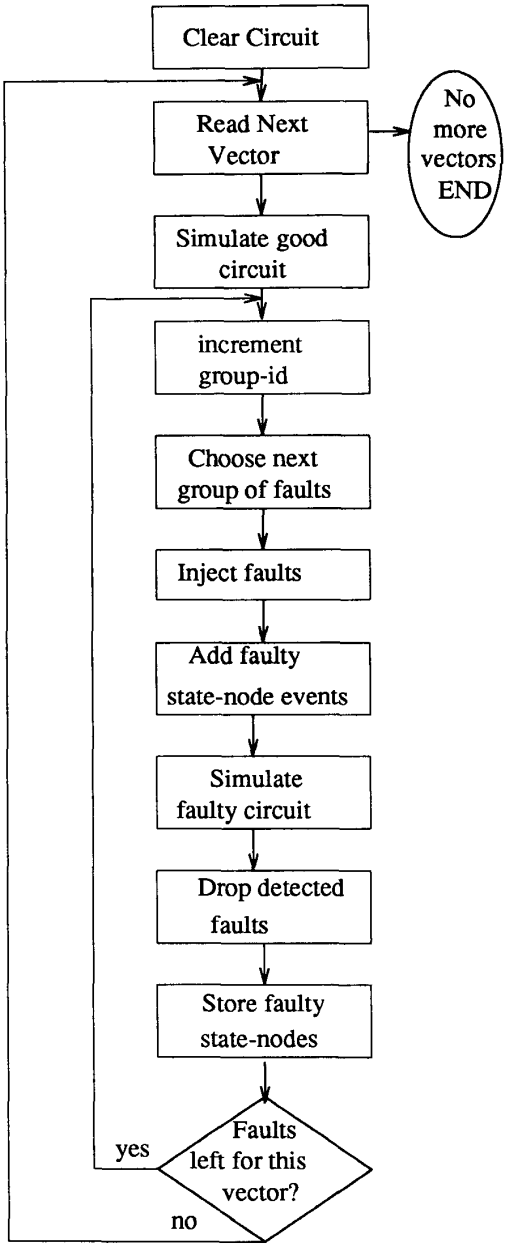

Fig. 4. PROOFS algorithm

logic (0, 1, X,and Z) is used. To code these four values two bits are used, one in V0 and one in V1; 0 is coded as (1,0), 1 as (0,1), X as (0,0), and Z as (1,1). Table 2 shows the logic used to evaluate different gates, 32 faulty machines at a time. In Table 2, each of the gates have two inputs A and B and an output V. The tristate gate, TRIG, has an input A and an enable input E. The BUS element has two inputs A and B wired to a bus. In our simulator, tristate gate outputs can only be connected to a BUS element. The good machine values also use a 32 bit word to allow easy comparison to the faulty machine values.

Good Circuit Simulation: The good machine evaluation is performed by a standard event driven logic simulator. The value of every node in the circuit is kept in a single array which is separate from where the values of the faulty machines are kept. When the circuit is parsed, it is also levelized to prevent the repetition of evaluating a gate that receives multiple events on its inputs.

Fault grouping: The dynamic fault grouping strategy regroups the faulty machines into groups of 32 machines for each vector that is applied to the circuit. The fault list consists of a linked list of all the remaining undetected faults as shown in Figure 7. To choose the next group of 32 faults, the list is traversed linearly, and a fault is added to the fault group if the faulty machine is active. A faulty machine is considered *active* if the faulty machine propogated a value to a state-node different from the good machine value in the previous time frame, or if in the current time frame, the fault results in a different value from the good machine line value for the faulty line (e.g. a good machine value of 1 on line L for fault L s-a-0). The faulty machine corresponding to an *inactive* fault is guaranteed to have no line values different from the good machine. Therefore, adding it to the fault group would waste one bit space in the word for the current time frame. The exclusion of the inactive faults results in significant reduction in the number of distinct group simulations, that is, it reduces the number of iterations of the inner loop in Figure 4.

Group-Id: Before evaluating any fault group, the group-id is incremented. A group-id is kept to distinguish between faulty machine values from different faulty machines groups. For the good machine the value of every line is kept, but for the faulty machine every line has a value and a group-id. If the group-id does not match the current time stamp then the faulty value is a residual value from a previous computation and the good machine value should be used. An example of a gate evaluation is shown in Figure . In this example the current group-id is 29, so to evaluate the gate for the faulty machine group, the good machine value of the middle input is used because the group-id is not 29. Without this group-id feature, one has to clean up the faulty value array by copying the good machine values. Therefore the group-id avoids the overhead to restore the good machine values for every fault.

Fault injection: Next, every fault in the chosen fault group must be injected into the circuit. The faults are injected into the circuit in a novel way. Traditionally, fault injection is accomplished by associating a bit-mask with each gate. The mask essentially is a flag for each line associated with gate inputs or outputs. The flag indicates if the values to be used during the gate evaluation are the line values produced by the predecessor gates or the value to be used is a stuck-at value associated with the gate. This method requires that the flags be examined for every gate evaluation even though only a few gates have faults. Instead of using

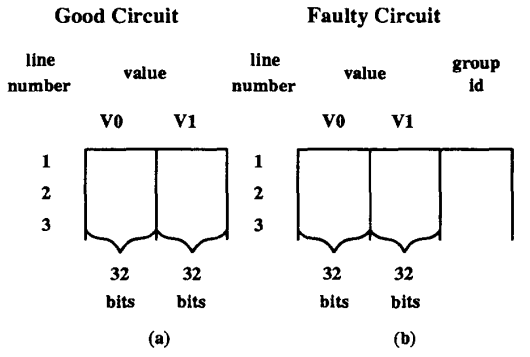

Fig. 5. Line Value Storage (a) good circuit (b) faulty circuit

Table 2. Gate evaluation

	V0	V1
AND	$A_0 \mid B_0$	$A_1 \& B_1$
OR	$A_0 \& B_0$	$A_1 \mid B_1$
INV	A_1	A_0
XOR	$(A_0 \& B_0) \mid (A_1 \& B_1)$	$(A_0 \& B_1) \mid (A_1 \& B_0)$
TRIG	$E_0 \mid (A_0 \& E_1)$	$E_0 \mid (A_1 \& E_1)$
BUS	$(A_0 \& B_0 \& B_1) \mid (A_0 \& A_1 \& B_0)$	$(A_1 \& B_0 \& B_1) \mid (A_0 \& A_1 \& B_1)$

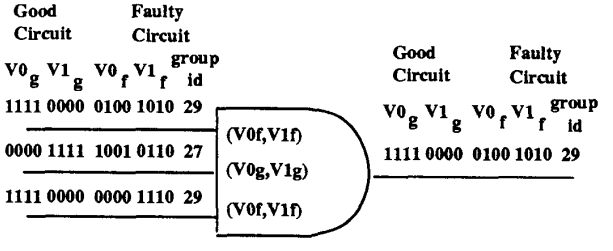

Fig. 6. Gate Evaluation

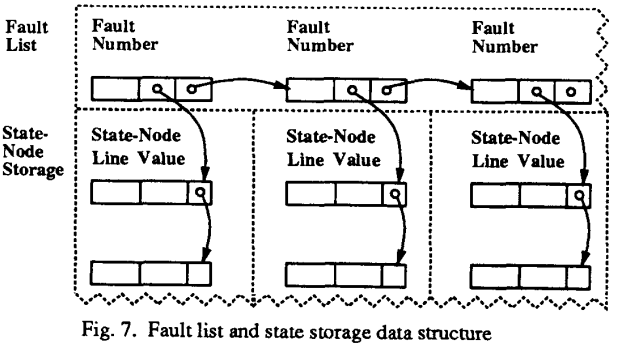

Fig. 7. Fault list and state storage data structure

faulty bit masks at every line, extra gates are inserted into the circuit. To inject a fault at a line stuck-at-1 an OR gate is inserted at that line. Figure 8 illustates the injection of the fault A stuck-at-1 in the third bit position of the value word. First, all the succesors of A become the succesors of the added OR gate, then A becomes one of the inputs of the OR gate. The other input of the OR gate is a dummy line which is set to the value of all zeros except a one in the bit position of the injected fault. Therefore, if the first four faulty values of line A were (wxyz), after evaluating the OR gate, it would have the first four faulty values be (wx1z), forcing the third bit position to the value 1. Similarly, to inject a stuck-at-0 fault an AND gate would be used with a dummy line value of all ones except a zero at the fault position. This technique of fault injection saves time compared to the technique of fault bit masks because the bit masks require extra time for every evaluation irrespective of whether a fault was active on that node, or not. Our technique only requires the changing of a few pointers per fault and evaluations are as fast as an efficient logic simulation. For each fault in the circuit, the pointers that need to be changed to inject that fault are determined in the preprocessing stage; therefore fault injection takes very little time.

State-node events: Each of the remaining faulty machines in the fault list has associated with it a linked list which contains each state-node of the faulty machine, which has a value different from the good machine value. The state-nodes are stored in this list from the previous input vector. These values must be inserted into the circuit with the current input vector and events must be scheduled for their successors.

Simulate faulty machines: The faulty machines in the current fault group are evaluated using an event driven fault

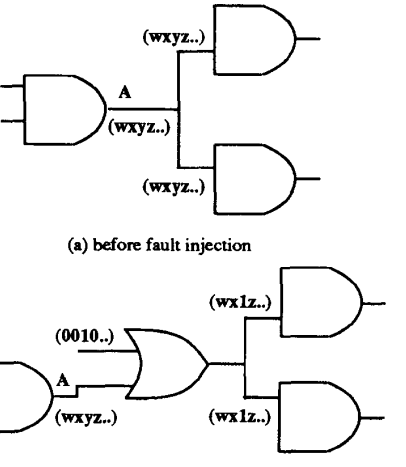

Fig. 8. Fault injection

simulator. A gate is evaluated by checking if each gate input faulty line has the current group-id. If it does, then the faulty value of that input is used, else the good value is used, as shown in Figure 6.

Dropping faults and storing fault state-nodes: A fault is detected if there is a faulty value at a primary output and the good circuit has a known value on that primary output. If a fault is detected then it is removed from the linked list of faults. If a fault is not detected then all the state-nodes with values different from the good machine values are stored in the linked list associated with the fault for the next vector.

2.1. Fault ordering

The proper grouping of faults is crucial in exploiting the benefit of parallelism from the 32 bit operation. If faults that cause the same events are in the same fault group, then the number of events to evaluate the faulty circuits is reduced. The fault list is constantly changing because of the dropping of detected faults, therefore it is necessary to consider the order of the complete list rather than a static grouping as in parallel simulation.

The faults are ordered by a depth first search of the circuit starting at the primary outputs. The parity of the faults are maintained in this search. This ordering tends to put faults with the same sensitized paths to an output, in the same word. This increases the probability that a sensitizing path is completely in a word. This fault ordering often reduced the number of faulty machine events by 50% over a random fault order. We also tried other ordering heuristics, for example depth first from the primary inputs, breadth first , and random ordering. We found that the depth first ordering from the primary outputs caused the fewest events.

3. RESULTS

The PROOFS algorithm was implemented in C++ and run on many of the ISCAS sequential benchmark circuits[21,22]. Table 3 shows a summary of the circuits and test vectors used to evaluate the fault simulator. The vectors were generated using the sequential circuit test generator STG3 [23-25]. These circuits were run on a SUN 3/280. Aside from running these circuits on PROOFS, they were also run on a state of the art concurrent fault simulator. The run time and average memory usage of the PROOFS and concurrent algorithms are shown in Table 4. A comparison of the two algorithms shows the PROOFS is 6 to 67 times faster than the concurrent algorithm while always requiring less memory. Memory is reduced by up to 7.5 times over the concurrent algorithm.

4. Conclusion

A very fast fault simulation algorithm has been described in this paper. Several new techniques were introduced as part of this algorithm. The new techniques introduced were: 1. Use of group-id to avoid the overhead of restoring the good values after each fault propagation 2. Concept of active and inactive faults to prevent eventless fault simulation 3. Efficient method of fault injection by circuit modification and 4. An efficient fault ordering to minimize events in word parallel operations. It has been shown that this algorithm is 6.6 to 67 times faster than a state of the art concurrent fault simulator while also requiring much less memory. It also has the advantage of being easy to implement because many parts of the code are only slight enhancements to a logic simulator.

Acknowledgement: This research was performed at the University of Illinois, Urbana-Champaign, while Wu-Tung Cheng was a Visiting Research Assistant Professor. The work was supported in part by the Semiconductor Research Corporation under contract SRC-89-DP-109, and in part by AT&T. We would also like to thank C. Vivekanand for his valuable input in the preparation of this manuscript.

5. REFERENCES

[1] V. D. Agrawal, S. C. Seth, and P. Agrawal, "LSI Product Quality and Fault Coverage," *18th Design Automation Conference,* June 1981, pp. 196-203.

[2] V. D. Agrawal, K. T. Cheng, and P. Agrawal, "Contest: A concurrent test generator for sequential circuits" *25th Design Automation Conference,* June 1988, pp.84-89

[3] E. G. Ulrich, and T. Baker, "The Concurrent Simulation of Nearly Identical Digital Networks," *10th Design Automation Workshop,* Vol. 6, June 1973, pp. 145-150.

[4] P. Goel, H. Lichaa, T. E. Rosser, T. J. Stroh and E. B. Eichelberger, "LSSD Fault Simulation Using Conjunctive Combinational and Sequential Methods," *International Test Conference,* November 1980, pp. 371-376.

[5] D. B. Armstrong, "A Deductive Method for Simulating Faults in Logic Circuits," *IEEE Trans. Comput.* Vol. C-21, No. 5, May 1972, pp. 464-471.

[6] W.-T. Cheng, and M.-L. Yu, "Differential Fault Simulation - A Fast Method Using Minimal Memory," *26th Design Automation Conference,* June 1989, pp. 424-428.

[7] S. Seshu, "On An Improved Diagnosis Program," *IEEE Trans. Electron. Comput.* Vol. EC-14, February 1965, pp. 76-79.

[8] J. P. Roth, W. G. Bouricius, and P. R. Schneider, "Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits," *IEEE Trans. Electron. Comput.,* Vol. EC-16, No. 5, October 1967, pp. 567-580.

[9] F. Ozguner, et al., "On Fault Simulation Techniques," *Journal of Design Automation and Fault Tolerant Computing,* Vol. 3, No. 2, 1979, pp. 83-92.

[10] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom, and T. McCarthy, "Fault Simulation for Structured VLSI," *VLSI System Design,* December 1985, pp. 20-32.

[11] S. J. Hong, "Fault Simulation Strategy for Combinational Logic Networks," *8th International Fault-Tolerant Computing Symposium,* June 1978, pp. 96-99.

[12] K. J. Antreich and M. H. Schulz, "Accelerated Fault Simulation and Fault Grading in Combinational Circuits," *IEEE Trans. on Computer-Aided Design,* November 1987, pp. 704-712.

[13] W. Ke, S. C. Seth, and B. B Bhattacharya, "A Fast Fault Simulation Algorithm for Combinational Circuits," *International Conference on Computer-Aided Design,* November 1988, pp. 166-169.

[14] F. Maamari and J. Kajski, "A Fault Simulation Method Based on Stem Regions," *International Conference on Computer-Aided Design,* November 1988, pp. 170-173.

[15] P. Goel, and P. R. Moorby, "Fault-Simulation Techniques for VLSI Circuits," *VLSI Design,* July, 1984, pp. 22-26.

[16] S. Davidson, and J. L. Lewandowski, "ESIM/AFS - A

Table 3. Circuits Descriptions

Circuit	Gates	Flip Flops	Vectors	Total Faults	Detected Faults	Fault Coverage
s208	96	8	111	416	285	68.5
s298	119	14	162	596	526	88.2
s344	160	15	90	652	624	95.7
s349	161	15	90	662	630	95.1
s382	158	21	2463	764	722	94.5
s386	159	6	174	772	696	90.1
s400	164	21	1282	800	742	92.7
s420	196	16	172	840	374	44.5
s444	181	21	1880	888	824	92.7
s526	193	21	754	1052	847	80.5
s526n	194	21	654	1052	849	80.7
s641	379	19	133	1276	1114	87.3
s713	393	19	107	1426	1184	83.0
s820	289	5	411	1640	1419	86.5
s832	287	5	377	1664	1445	86.8
s838	390	32	137	1676	526	31.3
s953	395	29	16	1906	147	7.7
s1196	529	118	313	2392	2392	10
s1238	508	18	351	2476	2396	96.7
s1423	657	74	36	2846	1780	62.5
s1488	653	6	590	2976	2815	94.5
s1494	647	6	471	2988	2797	93.6
s5378	2779	179	408	10590	8183	77.2
s35932	16065	1728	88	70584	62665	88.7

Table 4. Comparison of PROOFS to Concurrent

Circuit	PROOFS		Concurrent		Speedup Ratio	Memory Reduction Ratio
	Run Time (sec)	Maximum Memory(Kb)	Run Time (sec)	Maximum Memory(Kb)		
s208	2.3	80	24.3	552	11	7
s298	4.1	96	32.5	608	8	6
s344	2.9	104	33.5	656	12	6
s349	2.9	112	33.7	656	12	6
s382	67.2	112	445.1	728	7	7
s386	3.0	104	32.2	664	11	6
s400	37.1	112	347.2	720	9	6
s420	9.8	120	69.3	632	7	5
s444	56.2	120	757.5	808	13	7
s526	40.2	120	349.5	896	9	8
s526n	35.5	120	235.2	992	7	8
s641	5.0	208	38.5	600	8	3
s713	5.5	216	54.5	752	10	3
s820	23.4	192	182.0	744	8	4
s832	23.2	176	175.7	816	8	5
s838	19.3	224	145.8	888	8	4
s953	3.3	176	32.0	720	10	4
s1196	11.9	216	111.9	720	9	3
s1238	16.7	216	133.2	728	8	3
s1423	9.1	344	254.1	1288	28	4
s1488	53.7	272	544.6	1392	11	5
s1494	44.2	272	516.8	1184	12	4
s5378	174.0	752	1367.8	1544	8	2
s35932	358.3	5872	24148.6	5576	67	0.95*

*-Concurrent required 15 passes while PROOFS required 1 pass

Concurrent Architectural Level Fault Simulator," *International Test Conference*, November 1985, pp. 663-698.

[17] F. Brglez, "A Fast Fault Grader: Analysis and Applications," *International Test Conference*, November 1985, pp. 785-794.

[18] M. Abramovici, P. R. Menon, and D. T. Miller, "Critical Path Tracing - An Alternative to Fault Simulation," *20th Design Automation Conference*, June 1983, pp. 214-220.

[19] S. K. Jain, and V. D. Agrawal, "STAFAN: An Alternative to Fault Simulation," *21th Design Automation Conference*, June 1984, pp. 18-23.

[20] Y. H. Levendel, and P. R. Menon, "Fault-Simulation Methods - Extensions and Comparison," *The Bell System Technical Journal* Vol. 60, No. 9, November 1981, pp. 2235-2258.

[21] F. Brglez, and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits and A Target Translator in Fortran," *International Symposium of Circuits & Systems*, June 1985, pp. 662-698.

[22] F. Brglez, D. Bryan, and K. Kozminski, "Combinational Profiles of Sequential Benchmark Circuits," *International Symposium of Circuits & Systems*, May 1989, pp. 1929-1934

[23] W.-T. Cheng, "SPLIT Circuit Model for Test Generation," *25th Design Automation Conference*, June 1988, pp. 96-101.

[24] W.-T. Cheng, "The BACK Algorithm for Sequential Test Generation," *International Conference on Computer Design*, October 1988, pp. 66-69.

[25] W. -T. Cheng, and S. Davidson, "Sequential Circuit Test Generator (STG) Benchmark Results," *International Symposium of Circuits & Systems*, May 1989, pp. 1938-1941