Parallel Sequence Fault Simulation for Synchronous Sequential Circuits

Chen-Pin Kung and Chen-Shang Lin

Department of Electrical Engineering
National Taiwan University

Abstract

A novel parallel sequence fault simulation (PSF) algorithm for synchronous sequential circuits is presented. The algorithm partitions a given test sequence into subsequences of equal length and then performs fault simulation with these subsequences in parallel. To overcome the state dependency of sequential circuits, a multiple-pass method is developed to use minimal simulation passes. The experimental results of PSF on the benchmark circuits show that the speedup ratio over a serial sequence fault simulator is 6.6 on average for random test sequences.

1 Introduction

Fault simulation has been playing a major role in VLSI testing. Its applications range from grading the quality of test sets to directing the generation of test vectors[7]. As the sizes of circuits and test sets grow increasingly, efficient fault simulation algorithms have been developed to meet the challenge. These algorithms, in general, take advantages of either the small difference between faulty and fault-free circuits or the parallelism of computer word operations for multiple faults[6].

A drastically different approach has been proposed in the *parallel pattern single fault propagation* (PPSFP)[1] for combinational circuits. The PPSFP is recognized as one of the most efficient fault simulation algorithms. Two factors contribute its efficiency. First, it takes advantage of test pattern independence for parallel pattern processing. Second, it propagates each fault individually to reduce the memory requirement. Although PPSFP is shown to be extremely efficient for combinational circuits, it can not be applied to sequential circuits because the behavior of a sequential circuit at a specified time depends not only on the present input vector but also on the previous state which, in turn, might depend on the entire preceding input sequence.

In this paper, we propose a parallel sequence fault simulation (PSF) algorithm which extends the concept of PPSFP to synchronous sequential circuits. The algorithm partitions a given test sequence into subsequences of equal length and then performs the fault simulation with these subsequences in parallel. To solve the problem of state dependency of sequential circuits, a multiple-pass method is developed and a detection mechanism is devised to mini-

mize the number of simulation passes. The experimental results on the benchmark circuits show that the speedup ratio over a serial sequence fault simulator[8] is 6.6 on average for random sequences.

The remaining parts of the paper are organized as follows. The principle of parallel sequence logic simulation for synchronous sequential circuits will be introduced in the next section. Then, in Section 3, based on the same principle, the parallel sequence fault simulation (PSF) algorithm and its implementation will be described. The experimental results of PSF on sequential benchmark circuits will be shown in Section 4. Finally the conclusions will be given.

2 Parallel Sequence Logic Simulation

In this section, the principle of parallel sequence logic simulation will be described.

In a synchronous sequential circuit, the values in the memory elements and the present input determine the output and new values of the memory elements. The values of memory elements constitute a state vector, or simply a state, s of the circuit. A state vector is completely specified if all its values are either 0 or 1. And a state is unspecified if all its values are indeterminate, denoted by X . Given a completely specified initial state vector s^b and an completely specified input sequence T , a synchronous sequential circuit by logic simulation produces an output sequence and a final state vector s^f which is also completely specified. On the other hand, when the initial state is unspecified, the logic simulator in general can only determine an incompletely specified final state with some indeterminate values. However, it is evident that, with the same input sequence, the final state obtained from unspecified initial state covers that from the completely specified one. A state (vector) s_1 is said to *cover* another state s_2 if each value in s_1 is either X or is equal to the corresponding value in s_2 . Then, in general, we have the following observation.

Observation 1:

Let s_1^f and s_2^f (o_1^f and o_2^f) be the final states (output responses) of a sequential circuit after the same input sequence is applied with initial states s_1^b and s_2^b respectively. If s_1^b covers s_2^b then s_1^f covers s_2^f (o_1^f covers o_2^f).

The implication of the Observation is that with less information on the initial state, i.e. more X 's, the logic simulation can still produce a final state containing the actual one. Furthermore, given a more specified initial state, the final state would have less uncertainties, X 's. This observation is the basis of parallel sequence logic simulation.

A given test sequence T can be viewed as a concatenation of a number of equal-length subsequences, say N_S subsequences, $T_0 | T_1 | T_2 | \dots | T_{N_S-1}$ where $|$ denotes the concatenation operation. In the conventional logic simulation, the whole sequence T is applied serially to simulate a circuit. Let the states at the end of each subsequence be denoted by s_i^f , $i = 0, \dots, N_S - 1$. We name the simulation with input sequence applied serially as *serial sequence simulation*.

In the parallel sequence logic simulation, these separate subsequences are simulated in parallel and the status of the circuit corresponding to these subsequences are arranged into computer words to take advantage of the inherent parallelism of computer word operation. Since, in a sequential circuit, the responses may depend on the preceding inputs, the correct result in general can not be guaranteed by only one pass of parallel sequence simulation. However, from Observation 1, the parallel sequence result will contain the correct result, if the initial state for each subsequence is unspecified.

In order to obtain the correct result, multiple passes of simulation are developed. At the beginning of each simulation pass, the final state of the previous subsequence becomes the initial state as illustrated in Fig. 1. This transporting of states between two consecutive passes is formalized as follows. Define the extended state vector S as the vector of states of all subsequences, $s_0, s_1, \dots, s_{N_S-1}$, at the same time instant. An extended state vector in the simulation pass i is then denoted by $S(i)$ and its initial and final extended state vectors are denoted by $S^b(i)$ and $S^f(i)$, respectively. Then the initial states of each pass are as follows:

$$\begin{aligned} S^b(1) &= X \text{ and} \\ s_j^b(i) &= s_{j-1}^f(i-1) \text{ for } j \geq i-1 \text{ and } N_S \geq i > 1, \\ s_j^b(i) &= X \text{ for } 0 \leq j < i-1 \text{ and } N_S \geq i > 1. \end{aligned}$$

Basically, the above transporting rule is to allow the resultant state from a subsequence to become the initial state of the successive subsequence. It can be seen that $s_{j-1}^f(j) = s_{j-1}^f$, namely, the final state of the subsequence T_{j-1} in pass j is the correct state since it is obtained by exactly the same partial sequence, $T_0 | T_1 | \dots | T_{j-1}$, as the serial sequence simulation. Furthermore, in the simulation pass j , all the states $s_{j-1}(j)$'s as well as the output responses of T_{j-1} are the correct results. Therefore for pass $j+1$, the subsequences $T_0, \dots, T_{j-1}$ are no longer needed for simulation and hence their initial states are set to be X and the responses are ignored. For the remaining $s_j^f(i)$ for $j > i-1$, it still covers the correct s_j^f and is being refined to the correct one with each additional pass. Clearly, when the number of simulation passes is equal to the number of subsequences, the correct states and responses of the

Figure 1: State transporting in parallel sequence simulation

entire sequence can be obtained. In this extreme case, it is obviously more efficient to use the serial sequence simulation. Fortunately, in most cases, the parallel sequence simulation needs only a few passes to produce the correct result and it outperforms the serial sequence simulation as will be shown in our experiments. What we need now is a mechanism to detect when the correct result has been produced so that the simulation can be terminated. This mechanism is described in the following. Define, for each $i = 1, \dots, N_S$, a mask vector $M(i)$ of the same dimension as the extended state vector $S(i)$ and with entries equal to zero for those corresponding to $s_k(i)$, $k < i-1$, and $s_{N_S-1}(i)$, and equal to one otherwise. Then the detection condition can be defined in the following.

Definition:

During the parallel sequence simulation, the simulation is said to be stable at pass ℓ if $M(\ell) \cdot S^f(\ell) = M(\ell) \cdot S^f(\ell-1)$.

In other words, when the final state of each relevant subsequence is identical to that of preceding simulation pass, the parallel sequence simulation is called stable. When the simulation is stable, all the subsequent passes can only produce identical states which therefore must also be the correct states and responses. Moreover, this is the minimum pass to have correct states because the initial states in the previous pass have not been correct. Note that the final state of the last subsequence, $s_{N_S-1}^f(i)$, is masked out in $M(i)$ because its states and responses are always correct if the previous subsequence has produced the correct states. Therefore we have the following result.

Observation 2:

If the parallel sequence simulation becomes stable at pass ℓ , then all the states $M(\ell) \cdot S(\ell)$, $s_{N_S-1}^f(\ell)$, and $s_{j-1}(j)$ for $j < \ell$ are the correct states and their corresponding responses are also the correct responses. Moreover, ℓ is the minimum simulation passes to obtain the correct states and responses of the whole sequence.

Based on the above observation, we can derive the correct simulation results with a few simulation passes by terminating simulation when it is found to be stable. The same

principle will be applied to the parallel sequence fault simulation.

3 Parallel Sequence Fault Simulation (PSF)

In this section, we extend the principle of parallel sequence logic simulation to fault simulation. The described parallel sequence fault simulation is applicable to any single-fault-propagation simulator [5], i.e., the faulty circuit associated with a single fault is parallel-sequence simulated at a time. Basically, the simulation of a faulty circuit with parallel sequence is the same as the fault-free logic simulation. The fault simulation produces correct result when the simulation is stable. Therefore, a fault can be declared undetectable only both fault and good parallel sequence simulations are stable. Detectable faults however can be determined even before the stable condition is met. When a certain primary output of the faulty circuit produces a determinate value, 0 or 1, which is different from the determinate value of the good circuit, the fault is detected. This is because, from Observation 1, a determinate value is a correct value for faulty circuit as well as good circuit. In our implementation, a detected fault is dropped immediately without further simulation. The above discussions are summarized in the next observation.

Observation 3:

A fault is detected if, in parallel sequence fault simulation, there exists a primary output which has determinate but different values in good simulation and fault simulation regardless of the stable condition. And a fault F can not be detected by the whole test sequence if F is not detected in parallel sequence fault simulation when both fault and good simulations are stable.

From observation 2, the final state differences for each subsequence between the good and faulty circuits will remain unchanged in further simulation passes if the good and fault simulations are stable. Since the detectability of a fault can not be determined before the good simulation is stable even though the fault simulation has been stable. One simple way to decide if a fault simulation is stable is to compare the state lists of current and preceding simulation passes when the good circuit simulation has been stable. For the sake of saving memory requirement and enhancing the performance in PSF, state lists are not retained for previous simulation pass. Instead, two counters, C_d and C_x , for each fault are used to record the necessary information for all subsequences chosen by the mask vector, where C_d is the total number of determinate state variables which are different from the good circuit and C_x is the total number of indeterminate state variables which are different from the good circuit.

When the good circuit simulation becomes stable, for each faulty circuit, new C_d and C_x are computed and compared with the stored values. If these values remain the

```

partition the test sequence into  $N_s$  subsequences;
initialize the circuit status;
for each fault
   $C_x = C_d = 0$ ;
  while (fault simulation is not stable)
  {
    for each simulation pass
    {
      for every parallel test vector
        perform active fault simulation
    }
    /* determine if a simulation is stable
    * by checking the final states
    */
    if (the final states of good circuit
        are identical to preceding pass)
      then good simulation is stable;
    for each undropped fault
    {
      compute the values of  $C'_x, C'_d$  /* new  $C_x$  and  $C_d$  */
      if ( $C'_x == C_x$  and  $C'_d == C_d$  and
          the good simulation is stable)
        then drop the fault; /* it is undetectable */
      else
        update  $C_x$  and  $C_d$ ;
    }
    if there is any undropped fault
      then shift each parallel test vector one bit;
    else
      fault simulation is stable;
  }

```

Figure 2: Algorithm for parallel sequence fault simulation.

same then this fault simulation is stable. Since the checking begins when good simulation is stable, a faulty circuit must have some indeterminate values change to determinate values if it is not yet stable. Thus, either C_d or C_x or both will change. This guarantees to identify a stable fault simulation when it has been stable. The above discussions can be summarized in the following observation.

Observation 4:

After the good simulation has been stable, a fault simulation is stable if both C_x and C_d are identical to that of preceding simulation pass.

Based on the above discussions, the algorithm of parallel-sequence fault simulation are given in Fig. 2. In PSF, we adopt the active fault simulation technique proposed in [3,4] to reduce the number of events by simulating only the active faults. In each simulation pass, the initial extended state vector is determined according to the transporting rule, the simulation is proceeded as in the active fault simulation, and the stable condition is verified at the end of pass to determine the termination of fault simulation. Note that in PSF algorithm, each parallel test vector is shifted one bit at the end of pass, to achieve the state transporting as well as the state mask.

It is also worth noting that when a combinational circuit is simulated, PSF needs only one pass. In other words, PSF is reduced into a parallel pattern simulator for combinational circuits.

4 Results and Discussions

The PSF algorithm was implemented in C and runs on SUN4/390 workstation. To evaluate the performance of PSF, a serial sequence fault simulator SSF also based on active fault simulation is implemented. The active fault simulation is an improved differential fault simulation[2] which has been reported to achieve 3-12 times faster than an existing concurrent fault simulation.

To evaluate the performance of parallel sequence fault simulator for a variety of circuits and test sequences, our experiment has been conducted on the sequential benchmark circuits each with three different sets of test sequences. These three sets are randomly generated test sequences, sequences from a simulation-based ATPG[7], sequences from a propagation-then-justify ATPG. The comparison results of PSF and SSF for these test sets are shown in Tables 1, 2, and 3, respectively. Also note that, in the experiment, each test sequence T is made possible to be partitioned into equal-length subsequences by inserting appropriate number of X 's at the beginning of T . These inserted X 's do not affect the results of simulation, since the circuits have indeterminate initial states.

In each table, the column N_1 is the minimum number of simulation pass at which the good simulation becomes stable, N_2 is the average number of simulation passes to identify an undetected fault, and S_i is the length of each subsequence. Note that in all three tables, N_2 is only slightly larger than N_1 , because the fault simulation is recognized stable after the good simulation is stable. Therefore, N_1 is an excellent indicator for the performance of PSF. The circuit s5378 has poor performance with PSF in these tables because it has particularly high N_1 resulting from its oscillatory subcircuits.

Table 1 shows the results of random sequences. For all but one (s5378) circuits PSF is faster than the SSF. The average speedup (excluding s510) of PSF over SSF is 6.6. The unusually high speedup of s510 is because the circuit is not initializable and the parallel sequence logic simulation becomes stable at the end of pass 1.

The test sequences in Tables 2 and 3 are generated by a simulation-based ATPG and a propagate-then-justify ATPG, respectively. The average speedup ratios of PSF over SSF are 3.0 and 3.09. Clearly, the performance of PSF for test sequences generated by ATPGs is not as good as for random sequences. One obvious reason is that the test sequences from ATPGs for comparative fault coverage tend to be shorter. However, the test length is only one of the factors for the PSF performance. We have also found that PSF encounter degraded performance with circuits which are difficult to initialize. For example, in s1494, the FFs can not be initialized unless a particular reset pin is en-

Table 1: Results for random test sequences

circuit	#vec.	S_i	fault cov.	CPU time (sec)		N_1 / N_2	speedup ratio
				SSF	PSF		
s208	640	20	34.88	5.02	0.58	2/2.07	8.66
s298	640	20	73.05	6.15	0.77	2/2.13	7.99
s344	640	20	94.74	5.72	0.70	2/2.11	8.17
s349	640	20	93.71	5.95	0.70	2/2.01	8.50
s382	640	20	12.28	13.23	2.0	2/2.42	6.62
s386	640	20	54.17	4.0	0.65	2/2	6.15
s400	640	20	12.03	13.63	2.12	2/2.41	6.43
s420	640	20	26.28	13.58	1.50	2/2.02	9.05
s444	640	20	11.18	15.73	2.35	2/2.43	6.69
s510	320	10	0.0	10.77	0.53	1/1.05	20.32
s526	640	20	8.65	19.67	2.82	2/2.29	6.97
s641	640	20	81.56	7.50	1.25	2/2.02	6.0
s713	640	20	76.94	8.70	1.40	2/2.01	6.21
s820	640	20	37.29	12.52	1.45	2/2	8.63
s832	640	20	33.22	13.22	1.62	2/2	8.16
s838	640	20	22.17	43.60	5.43	2/2.02	8.03
s953	640	20	8.34	117	37.9	2/2.34	3.08
s1196	640	20	83.01	11.0	1.40	2/2	7.85
s1238	640	20	77.27	12.80	1.62	2/2	7.90
s1423	640	20	36.17	64.22	13.95	2/2.17	4.60
s1488	640	20	55.52	23.32	3.03	2/2	7.70
s1494	640	20	54.58	24.15	3.10	2/2	7.79
s5378	640	20	53.29	160	169	29/29.01	0.95
s9234	320	10	0.26	474	87.1	4/4.02	5.44
s13207	640	20	5.96	1504	321	5/5.32	4.68
s15850	320	10	0.75	946	165	3/3.02	5.73
s35932	640	20	68.65	1150	259	2/2.0	4.44
s38417	640	20	3.53	4778	804	2/2.34	5.94

abled. The worst-case scenario of PSF occurs when this reset pin is enabled only at the first few vectors in the entire test sequence. Then, during PSF simulation, states for most of the subsequences remain indeterminate, and consequently the number of simulation passes to become stable is close to the number of subsequences. The phenomenon is more significant with the simulation-based ATPG. The basic mechanism of simulation-based ATPG is to initialize the circuit once and then to detect faults by appending more test vectors. Therefore, with such test sequences, PSF generally needs more simulation passes to become stable and it shows a slightly inferior performance in Table 2 than in Table 3. Another factor for the performance is the number of events. This can be seen in s35932 and s38417, both of which have more than 1000 FFs. In s38417, only a small part can be initialized and hence only a small number of faulty events are generated during the simulation. As a result, the fault coverage is pretty low, but the speedup ratio is better than that of s35932.

In summary, PSF delivers better performance than SSF in most cases, although the speedup ratios vary with circuits and test sequence construction. To alleviate the worst-case scenarios, a simple strategy might be employed by performing the parallel sequence logic simulation before the fault simulation. If N_1 is more than a certain threshold ratio, say $1/2$, of N_S , serial sequence fault simulator might be preferred.

Table 2: Results for test sequences generated by a simulation-based ATPG

circuit	#vec.	S_1	fault cov.	CPU time (sec)		N_1 / N_2	speedup ratio
				SSF	PSF		
s208	400	13	63.26	3.12	0.75	4/4	4.16
s298	350	11	85.71	3.75	1.00	5/5.02	3.75
s344	300	10	95.03	3.42	0.45	2/2	7.6
s349	99	4	95.43	1.17	0.47	4/4.06	2.49
s382	1014	32	88.97	26.23	12.88	10/10.02	2.04
s386	300	10	81.77	1.18	0.83	8/8	1.42
s400	1512	48	87.5	37.13	16.25	9/9	2.29
s420	200	7	40.70	4.83	1.68	4/4.01	2.88
s444	1692	54	85.87	48.35	23.23	10/10.03	2.08
s526	2682	86	79.28	85.62	33.38	8/8.13	2.57
s641	211	7	84.37	3.67	2.57	7/7.01	1.43
s713	350	11	81.76	4.96	1.87	4/4	2.65
s820	900	29	93.88	10.05	2.18	3/3	4.61
s832	703	22	91.61	9.88	3.52	5/5	2.81
s838	150	5	29.05	12.30	13.40	10/10.02	0.86
s1196	650	21	97.26	10.62	0.98	2/2	10.84
s1238	500	16	91.44	9.25	1.08	2/2	8.56
s1423	1501	48	82.97	371	350	9/9.06	1.06
s1488	300	10	94.01	10.63	13.48	29/29.01	0.79
s1494	850	27	96.02	24.37	32.42	31/31.02	0.75
s5378	501	16	69.15	85.35	110	26/26.01	0.76
s13207	149	5	5.81	352	166	10/10.07	2.12
s35932	298	10	88.26	477	439	4/4.01	1.09
s38417	199	7	3.24	1446	590	5/5.26	2.45

5 Conclusion

A novel parallel sequence fault simulation algorithm (PSF) has been described. Conceptually an extension of parallel-pattern single-fault propagation of combinational circuits to sequential circuits, the PSF partitions a given test sequence into subsequences of equal length and then performs fault simulation for these subsequence in parallel. To overcome the state-dependency of sequential circuits, a multiple-pass method has been developed and a detection mechanism has been devised to use minimal simulation passes.

The performance of the parallel sequence fault simulator has been extensively evaluated on the sequential benchmark circuits with three different sets of test sequences. These three sets are randomly generated test sequences, sequences from a simulation-based ATPG, sequences from a propagation-then-justify ATPG. The results has also been compared with serial sequence fault simulator (SSF) based on active fault simulation. The parallel sequence simulator performs surprisingly well against the serial sequence one. The speedup ratio is especially significant, 6.6, in the cases of random sequences. Although the speedup ratios vary with circuits and test sequence construction, PSF has delivered far better performance than SSF in most cases. For the few adverse cases, a screen strategy has also been proposed by first running a parallel sequence logic simulation with little overhead.

Table 3: Results for test sequences generated by a forward-then-justify ATPG

circuit	#vec	S_1	fault cov.	CPU time (sec)		N_1 / N_2	speedup ratio
				SSF	PSF		
s208	218	7	63.72	1.73	0.63	5/5	2.75
s298	354	12	83.12	3.32	0.98	4/4.12	3.38
s344	147	5	95.91	1.63	0.43	3/3	3.79
s349	243	8	95.71	2.82	0.43	2/2	6.56
s382	353	12	78.20	11.35	7.63	12/12	1.51
s386	402	13	81.77	2.27	0.68	3/3	3.34
s400	388	13	77.12	13.53	10.12	14/14.01	1.34
s420	142	5	41.63	3.35	2.13	9/9	1.57
s444	167	6	76.16	10.05	7.60	15/15.10	1.32
s526	296	10	52.97	13.87	12.95	7/7.28	1.07
s641	351	11	79.02	4.78	1.28	4/4	3.73
s713	600	19	81.41	9.07	1.95	3/3	4.65
s820	1005	32	95.76	11.65	2.48	3/3	4.69
s832	1026	33	94.02	14.40	2.58	3/3	5.58
s838	198	7	29.40	14.83	8.35	5/5.01	1.78
s1196	322	11	99.76	3.65	0.73	2/2	5.0
s1238	315	10	91.0	5.87	0.93	2/2	6.31
s1423	139	5	40.86	20.12	13.53	5/5.39	1.49
s1488	1097	35	93.33	49.93	9.80	4/4	5.09
s1494	1345	43	96.28	58.55	14.45	6/6	4.05
s5378	188	10	64.82	41.45	41.57	14/14.05	0.99
s13207	122	4	6.12	290	209	16/16.01	1.39
s35932	638	20	89.76	852	753	4/4	1.13
s38417	74	3	3.51	526	305	8/8.05	1.72

References

- [1] J. A. Waicukauski, E. B. Eichelberger, D. O. Forlenza, E. Lindbloom and T. McCarthy, "Fault Simulation for Structured VLSI," *VLSI System Design*, pp. 20-32, December 1985.
- [2] W. T. Cheng and M. L. Yu, "Differential Fault Simulation - A Fast Method Using Minimal Memory," *Proc. 26th DAC*, pp. 424-428, June 1989.
- [3] T. M. Nieman, W. T. Cheng and J. H. Patel, "PROOFS: A Fast, Memory Efficient Sequential Circuit Fault Simulator", *Proc. 27th DAC*, pp. 535-540, 1990.
- [4] E. Vandris and G. Sobelman, "Algorithms for Fast, Memory Efficient Switch-Level Fault simulation", *Proc. 28th DAC*, pp. 138-143, 1991.
- [5] F. Osguner, et al., "On Fault Simulation Techniques," *Journal of Design Automation and Fault Tolerant Computing*, Vol. 3, No. 2, pp. 83-92, 1979.
- [6] M. Abramovici, et al., "What is the Path to Fast Fault Simulation?" *Proc. International Test Conference*, pp. 183-192, 1988.
- [7] V. D. Agrawal, K. T. Cheng, and P. Agrawal "A Directed Search Method for Test Generation Using a Concurrent Simulator," *IEEE Trans. Computer-Aided Design*, Vol. 8, pp. 131-138, February 1989.
- [8] C. P. Kung and C. S. Lin, "Parallel Sequence Fault Simulation", *Proc. International Symposium on VLSI Technology, Systems, and Applications*, May, 1991.