COFS – A cell oriented fault simulator

Joachim Hartmann Björn Schieffer Uwe Sparmann
Lehrstuhl Prof. Hotz
Fachbereich 14 – Informatik
Universität des Saarlandes
D-6600 Saarbrücken, West Germany

Abstract

Currently, in most fault simulators physical defects have to be modeled as stuck-at faults in networks of primitive gates such as AND, NAND, etc. On the other hand, the inadequacy of the stuck-at fault model for today's technologies has been pointed out by many authors. In this paper, a compiler driven fault simulator is presented which handles arbitrary combinational faults. The set of basic cells and their fault models are not coded into the algorithm but can be specified by the user in a library. It is shown that this increased flexibility, which is obtained by cell instead of line oriented fault injection, does not degrade the performance of the simulator but, in contrast, results in a considerable speed-up (up to factor 40).

1 Introduction

As has been observed by many authors a large amount of the physical defects typical of today's VLSI technologies can not be correctly modeled by stuck-at faults [BART82, BA83, GCV80, LA83]. In order to overcome this problem, techniques are under development for automatic extraction of fault models for moderate sized basic cells which take into account aspects like physical layout or defect statistics of the manufacturing process [Mal87, FC88]. But most of the algorithms for test generation and fault simulation can not take advantage of such an approach since they have been tailored to the stuck-at fault model. Thus, it is necessary to generalize these algorithms such that they can also be used in the above setting. Examples of first attempts in this direction are [RGA87, CA87]. Since test generation and fault simulation are extremely time consuming tasks, this generalization must not degrade performance.

In this paper, the above questions are discussed with respect to fault simulation. In particular, we consider the compiler driven fault simulation approach presented in [KS85, DG87]. The original algorithm is tailored to the stuck-at fault model and thus works in a line oriented fashion. Faults are inserted by setting a specific line to a fixed logical value. Internal faults of complex cells such as XORs, multiplexers, and full-adders can only be modeled by replacing these cells by networks of simple gates (AND, NAND, etc.). Since this gate-level representation in most cases differs from the physical realization and since faults can only be attached to lines, some physical failures do not correspond to the modeled stuck-at faults and vice versa. In contrast, COFS (<u>c</u>ell <u>o</u>riented <u>f</u>ault <u>s</u>imulator) works in a cell oriented manner. Arbitrary combinational failures of basic cells can be described by giving the function of the faulted cell. The set of basic cells with their corresponding fault models is specified in a library. Thus, it is very easy to add a new basic cell or to modify the fault model of an existing one. In particular, the library could be generated by a tool which automates the process of fault modeling like [FC88]. The flexibility in fault handling is achieved by a cell oriented fault injection. It is checked whether the current input combination of the cell under fault injection allows fault detection and, if so, the output combination is set to the faulty value.

At first sight one could suppose that the generalizations described above would result in performance degradations compared to the original algorithm. But in fact, the contrary is true. To illustrate this effect, we compare the line based algorithm to COFS considering the special case that only stuck-at faults are modeled in the library. It will be shown that the cell based approach naturally implies a number of improvements compared to line oriented fault simulation. Some of these (like use of high-level representations for signal evaluation) have already been used in other fault simulators [RGA87, SSB89], other ones (like parallel insertion of faults) are specific to our approach. The level of improvement compared to the original algorithm depends on the use of complex cells like multiplexers or full-adders. Thus, simulation of a realistic number of patterns yields for circuits consisting only of simple gates (AND, NAND, etc.) a speed-up factor of approximtely 2 while for circuits heavily using complex cells speed-up factors of even up to 39 were gained.

The paper is structured in four sections. We start by explaining our fault modeling technique. Then, an overview of compiler driven fault simulation is given. The modifications necessary to generalize the approach of [DG87] to our fault model are discussed in Section 4. Section 5 analyzes the performance of our fault simulator. Besides a theoretical analysis, experimental results for the benchmarks [BF85] and some circuits with complex cells are also presented.

2 The fault model

As an example of the inadequacy of the stuck-at model consider the NMOS gate shown in Figure 1. The function of the correct gate is $\overline{(i1 \vee i2) \wedge (i3 \vee i4)}$. If the gate is faulted by the open marked f_{open}, the faulty gate computes the function $\overline{(i1 \wedge i3) \vee (i2 \wedge i4)}$. Clearly, this fault can not be modeled correctly by a stuck-at fault and thus may not be tested by a test set which has been constructed with respect to this fault model.

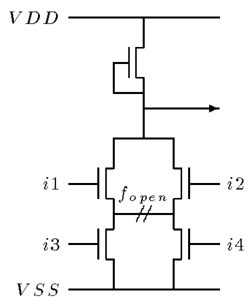

Figure 1: NMOS cell faulted by an open

In order to overcome such deficiencies, new fault models have been derived for technologies like NMOS and CMOS [GCV80, Wad78]. But the fault modes of circuits depend not only on the technology used for fabrication but also on defect statistics of the process and the physical layout of the chip. Thus using these fault models for test generation and fault simulation implies the following inefficiencies [Mal87]:

1. Some of the physical failures are not correctly modeled by the fault model and therefore may not be detected by the generated test.

2. Computing time is wasted for faults which only occur in the model and not in the actual circuit.

Since physical failures depend on so many conditions (and thus change whenever one of these factors changes), it is important to automate the process of fault modeling. In [Mal87, FC88] a method called *Inductive Fault Analysis* (IFA) is proposed which can be used to derive realistic fault models for circuits by taking into account fabrication technology, defect statistics of the actual process, and physical layout. By analyzing examples of CMOS circuits the following results were obtained in [FC88]:

1. Only less than 50% of the faults extracted by IFA could be correctly modeled by stuck-at faults.

2. No more than 7% of the extracted faults caused sequential behavior.

From the above observations it follows that the most urgent demand on algorithms for fault simulation and test generation is the ability to cope with arbitrary combinational fault models. Algorithms should be developed which are not tailored to a specific fault model but can work on a fault model extracted for the basic cells of a circuit by use of a method like IFA. Our fault simulator can handle arbitrary combinational fault models and thus is an important step in this direction.

As a consequence of our approach, we associate faults to cells. All faults that may occur in a cell of type t form the set F_t. A fault $f \in F_t$ can alter the cell's behavior in an arbitrary manner as long as it remains combinational. It is described by its *extended test set (ETS)* which specifies the input/output pairs for which correct and faulty cells differ, i.e.

$$ETS_t(f) := \{(i,o) | out_t(i) \neq out_t^f(i) = o\},$$

where out_t and out_t^f denote the functions of the correct and faulted cell, respectively. As an example let us consider the fault of Figure 1. It can be described by the extended test set

$$ETS_t(f_{open}) = \{(0110, 1), (1001, 1)\}.$$

(The value of input i_k is given by the k^{th} bit from the left.) Of course, stuck-at faults can be described in our model as well. For example the stuck-at 1 fault at input i_1 of the above cell has the following ETS

$$ETS_t(i1 \text{ s-a-1}) = \{(001*, 0), (0001, 0)\},$$

where $(001*, 0)$ is an abbreviation for the tests $(0010, 0), (0011, 0)$ ('$*$' stands for an arbitrary element from $\{0, 1\}$).

Surely, the above modeling technique is only adequate for moderate sized primitives. (The representation of an extended test set can grow exponentially in the number of inputs and outputs.) Since exact fault modeling using a method like IFA is a very expensive process, it is only applicable to cells of this kind. Thus, the above restriction seems natural.

3 Fault simulation by single fault propagation

In this section, we will give some background on fault simulation which is necessary for an understanding of our algorithm.

Fault simulation relying on *single fault propagation* is a wide spread technique for combinational circuits due to its efficiency and ease of implementation [KS85, WEF*85, DG87, HSU87, AS87, BCRR87, KCD90, BHKS91]. In the *compiler driven* approach [KS85, DG87, BCRR87, KCD90], a software routine is generated for the simulation of the circuit. For each gate it contains an instruction block calculating the gate's function. To ensure that no gate is evaluated before its inputs have been computed, the order of the instruction blocks has to respect a topological order[1] of the gates.

In order to raise the number of gate evaluations per second, several input patterns are simulated in parallel. This can be done by making use of the machine word length M. For each signal, M bits representing M different assignments are stored in one word. With the help of bitwise logical operations (INV, AND, OR, XOR), M different input combinations of the circuit can be handled simultaneously.

After a simulation of the correct circuit, single faults are injected by overwriting the correct signal values with the erroneous ones. Then, a simulation beginning at the fault site is performed (*single fault propagation*). If there is a difference between correct and faulty simulation at a primary output (PO), a test has been found. Faults at gates with a high topological sort number (TSN) are injected earlier than those at gates with low TSN. Thus, the values of signals above the fault location have not been changed in previous simulations. Consequently, the fault free simulation has to be done once only.

The event driven approach [WEF*85, AS87, HSU87, KCD90] prevents, in addition, unnecessary computations at gates below the fault site. Gates are evaluated only if they receive faulty signals at their inputs. This is usually done by handling priority ordered lists for the choice of the next gate that has to be processed.

[1] The gates of a circuit are topologically ordered, if for all gates g its predecessors occur earlier than g itself.

```
simulation( cell )
long cell;
{
 extern long signal[] ;
 register long r1 , r2 , r3 , r4 ;

 switch( cell )
 {
  case 0:
   signal[62] = signal[37] & signal[36] ;
  case 1:
   signal[63] = signal[37] & signal[38] ;
  case 2:
   signal[64] = signal[37] & signal[39] ;
                  .
                  .
                  .
  case 290:
   r1 = signal[64] ^ signal[63] ;
   signal[326] = (signal[64] & signal[63]) | ( r1 & signal[62]) ;
   signal[327] =  r1 ^ signal[62] ;
                  .
                  .
                  .
  case 1103:
   signal[1379] = signal[1361] & signal[1318] ;
  default:
 }
}
```

Figure 2: Source code for a simulation routine

4 Description of COFS

As a basis for our implementation, we adopted the compiler driven single fault propagation (CDSFP) method as proposed in [DG87]. In this section, the main differences between COFS and the approach of [DG87] are described. They are caused by the use of complex cells as primitives and the general combinational fault model. The next section discusses their influence on performance.

Since the repertoire of cells that COFS is able to handle can be augmented, information concerning the logical function and the fault model is read in from a library. Faults are given by their extented test sets. The logical function is described by a sequence of C statements reflecting the cell's i/o-behavior. This code will be copied in the simulation routine for each instance of the cell. Thus, a user of COFS has influence on the code generated which has a large impact on the performance as we will see later on.

As an example of a functional description consider the following library entry for a full-adder whose inputs are denoted by `i1`, `i2`, `i3` and the outputs by `o1` (carry) and `o2` (sum).

```
r1 = i1^i2;
o1 = (i1&i2) | (r1&i3);  /* carry output */
o2 = r1^i3;              /* sum output   */
```

This code segment is generated for each full-adder in the circuit, after `i1`, `i2`, `i3` and `o1`, `o2` have been replaced by the corresponding input and output signals, respectively. `r1` stands for an internal signal which should be stored in a register for short access times. This is supported by C for variables declared in the storage class `register`. Figure 2 gives an extract of the simulation routine generated for a 16-bit floating point multiplier. Each case in the `switch` statement contains the code for exactly one basic cell. If a simulation starting at the full-adder with TSN 290 is to be done, `simulation(290)` is called which causes the `switch` statement to branch to case 290. In [DG87] skipping of instructions is implemented by a `switch` statement, too. But there, one signal instead of one cell is evaluated in each `case`-block.

In fault simulators supporting only the single stuck-at model, fault injection is done simply by inverting the assignment of a line. Thus, it is possible to simulate up to two faults simultaneously in different bit positions, namely the stuck-at 0 and the stuck-at 1 fault at the line. In COFS, more faults can be inserted at a time. This is possible since faults are injected cell oriented instead of line oriented. For a detailed description of the fault injection phase, we need the following definitions. It is

$$INPUTS_t := \bigcup_{f \in F_t} \{i \mid \exists o : (i, o) \in ETS_t(f)\}$$

the set of input combinations that allow a fault injection at a cell of type t. For an $i \in INPUTS_t$ the corresponding faulty output combinations form

$$OUTPUTS_t(i) := \{o \mid \exists f \in F_t : (i, o) \in ETS_t(f)\}.$$

The set

$$FAULTS_t(i, o) := \{f \in F_t \mid (i, o) \in ETS_t(f)\}$$

contains those faults that can be tested with input i and then lead to the faulty output o, and

$$FAULTS_t(i) := \bigcup_{o \in OUTPUTS_t(i)} FAULTS_t(i, o)$$

comprises all faults that can be tested by input combination i.

A bit position from $\{1, \ldots, M\}$ is said to *match* the input combination i at gate g if g has i as input for this bit position. Let t be the type of the cell under fault injection (CUFI). Then, the fault injection works as follows:

First, it is determined for all $i \in INPUTS_t$, which bit positions match them at the CUFI. For all matched i's, an element o from $OUTPUTS_t(i)$ is inserted at the matching bit positions. After that, a simulation starting at the cell following the CUFI in the topological order is performed. If a faulty signal reaches a PO in a bit position that matched i and where o was inserted, the faults from $FAULTS_t(i, o)$ are removed from the fault list. Injection of faulty output combinations, subsequent simulation, and contingent fault removal are repeated until for all matched i's each element from $OUTPUTS_t(i)$ has been taken into account. Throughout the whole process, only those input and output combinations are considered that are able to detect uncovered faults at the CUFI. Moreover, word parallel operations are used in all stages. In the appendix, a detailed description of the fault injection is given in a PASCAL-like notation.

5 Performance of COFS

As is apparent from the previous section, the more general fault model has to be paid by a more expensive fault injection phase. On the other hand,

- the use of hierarchy given in complex cells as primitives and
- the cell oriented fault injection

promise speed-ups such that there is no performance degradation but even an acceleration compared to a conventional CDSFP simulator.

When primitives are given as complex cells instead of simple gates, fewer elements have to be processed. This already approves in the preprocessing that generates the simulation routine. Table 1 illustrates this fact. Times taken for the generation from an ASCII netlist format including the compilation have been measured for circuits with high-level primitives and their expanded counterparts. c499, c1355 are taken from the ISCAS benchmark circuits [BF85]. Note that c1355 is c499 where XORs have been replaced by their gate-level representations. The other circuits are arithmetical units: ia64 is a 64-bit integer adder, fa16 a 16-bit floating point adder, im16 a 16-bit integer multiplier, and fm16 a 16-bit floating point multiplier. It turns out that the preprocessing is accelerated by a factor of up to 3. These times (like those in the following tables) were measured on a SUN 4/60 FC-16-XX workstation with 12 MIPS.

circuit	preprocessing time		speed-up
	simple gates	complex cells	factor
c499/c1355	8.6 s	4.6 s	1.9
ia64	38.1 s	16.5 s	2.3
fa16	26.5 s	13.4 s	2.0
im16	84.7 s	26.9 s	3.1
fm16	54.9 s	20.2 s	2.7

Table 1: Preprocessing times

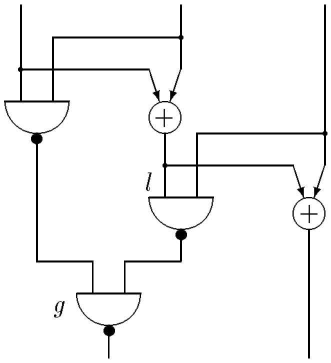

Figure 3: Full-adder

More important than consequences for the code generation are those for the running time of the fault simulation itself. In the remaining part of this section, we will discuss this issue. Since no other comparison to conventional fault simulators is possible, we consider the single stuck-at fault model. There are two kinds of time savings. On the one hand, the number of simulation runs is reduced, and on the other hand, simulation runs themselves become faster.

<u>Reduction in the number of simulation runs:</u>

1. As was mentioned in Section 3 event driven techniques are used to speed up simulations. For a long time, they have not been used for compiler driven simulators to avoid time consuming conditional jumps. (In a new paper [KCD90], the CDSFP approach has been combined with an event driven mechanism. The high overhead for event manipulation is amortized by a dynamic increase in the number of patterns per simulation run.) However, the implementation of COFS preserves straight line code and is, in a certain sense, event driven since a fault is inserted only if it is actually propagated to an output of the CUFI. A simple example of that is a two-input AND gate where faults at an input need not to be inserted when the other input carries the value 0. A less obvious instance is shown in Figure 3 for a full-adder (FA). For input combination 111, a gate-level fault simulator would insert 1 at l to check whether l is tested for stuck-at 1. Actually, the pattern is not able to test the fault because the NAND gate g is not sensitized. As l s-a-1 $\notin FAULTS_{FA}(111)$, COFS wastes no time for this fault.

2. A line oriented fault injection can insert at most two faults per simulation run. The number of input/faulty ouput pairs that can simultaneously be processed in COFS is bounded only by the word length M. Suppose for instance that the input combinations 00, 01, 10, and 11 of an XOR are distributed among the M bit positions. COFS is able to process faulty output values for all of them in only one simulation run. This covers the stuck-at 1 faults at $a1$, e, d, and c (cf. Figure 4). An exact treatment of these faults in a gate-level description would require one separate fault insertion for each line leading to four simulation runs.

3. Distinct faults are not only treated simultaneously in different bit positions but also in single ones. To see that, recall that one simulation run identifies for a bit position whether for a given (matched) input i

type t	2-input AND	XOR	MUX	HA	FA
average size of $FAULTS_t(i,o)$	1.5	5.8	2.5	2.6	4.5

Table 2: Average size of $FAULTS_t(i,o)$ for some cell types

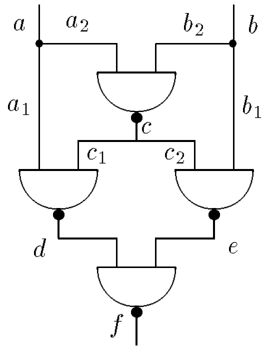

Figure 4: XOR

a faulty output o at the CUFI propagates to a PO. If this is the case, the elements of $FAULTS_t(i,o)$ are removed from the fault list. When $FAULTS_t(i,o)$ contains more than one element, several faults have been processed by one simulation run. From Table 2, which gives the average size of $FAULTS_t(i,o)$ for some important cell types[2], it becomes clear that often more than one fault is tested by a pair (i,o). (For a fair comparison, the sets $FAULTS_t(i,o)$ were reduced by a straightforward fault collapsing due to equivalence relations at simple gates [BF85].)

A consequence of the effect described above is a *complete* exploitation of dominance and equivalence relations within cells. The bigger the primitives are, the more fault equivalences (and dominances) can be used. For instance in an XOR cell as shown in Figure 4, the following 16 faults remain after the simple fault collapsing mentioned above [BF85]: stuck-at 1 faults at all lines and the stuck-at 0 faults at a, b, c, and f. If equivalence relations are fully exploited, only 10 faults have to be considered since the faults of the following sets have identical test sets: $\{a1$ s-a-1$, b1$ s-a-1$\}$, $\{c$ s-a-1$, c1$ s-a-1$, c2$ s-a-1$\}$, $\{b2$ s-a-1$, d$ s-a-1$\}$, $\{a2$ s-a-1$, e$ s-a-1$\}$, and $\{c$ s-a-0$, f$ s-a-0$\}$.

Table 3 illustrates the reduction of simulation runs for some circuits (c6288 and c7552 are the biggest benchmarks circuits). They have been examined for simulations of 224, 1024, 30016 patterns and a simulation with a dynamic abort (fault simulation is stopped as soon as 64 subsequent patterns did not test a not yet covered fault). A dynamic abort criterion yields more realistic measurements than a predefined number of patterns because it makes no sense to continue fault simulation if there are only faults with low detection probabilities left or to stop if there is the chance to detect further faults by random patterns. The third and forth column of Table 3 give results for a line and cell oriented fault injection, respectively. The last column lists the speed-up factors. For the cell oriented fault injection, circuit representations with complex cells as primitives were used if they were available (not so for c6288 and c7552). In contrast, in the line oriented approach, complex primitives had to be expanded for the modeling of internal faults.

As can be seen from Table 3, the reduction of the number of simulation runs grows with increasing complexity of the basic cells. As an example consider the measurements with dynamic abort criterion. For the benchmark circuits c6288 and c7552 which consist of simple gates (AND, NAND, OR, NOR and NOT), only reduction factors of 2.5 and 2.1 were measured. For the remaining circuits which are mostly built from complex cells (XORs, multiplexers, and full-adders) these factors range from 3.8 to 18.3. On the other hand, the speed-up factor can not be predicted by considering only the complexity of the basic cells of the circuit under consideration since effects from techniques like event-checking at the cell outputs heavily depend on

[2]Note that the sets $FAULTS_t(i,o)$ and consequently their average size as well depend on the cell's gate expansion. Table 2 is based upon standard realizations as given in Figure 3, 4, and 5.

circuit	patterns	simulation runs		speed-up
		line oriented	cell oriented	factor
c6288	224	6324	2555	2.5
	1024	6769	2980	2.3
	30016	22967	18381	1.5
	192†	6306	2538	2.5
c7552	224	11643	5756	2.0
	1024	25333	12335	2.1
	30016	345266	183175	1.9
	1024†	25333	12335	2.1
c499/ c1355	224	1632	313	5.2
	1024	2220	550	4.0
	30016	10407	7826	1.3
	448†	1852	386	4.8
ia64	224	10557	1234	8.6
	1024	28853	1383	20.7
	30016	637183	3266	195.1
	832†	24579	1346	18.3
fa16	224	9114	2417	3.8
	1024	25328	6610	3.8
	30016	390434	98049	4.0
	896†	22917	5985	3.8
im16	224	17939	2631	6.8
	1024	25102	3093	8.1
	30016	263165	16727	15.8
	448†	19980	2778	7.2
fm16	224	28783	5198	5.5
	1024	85314	14904	5.7
	30016	509262	104649	4.7
	1888†	128967	22913	5.6

Table 3: Number of simulation runs († dynamic abort)

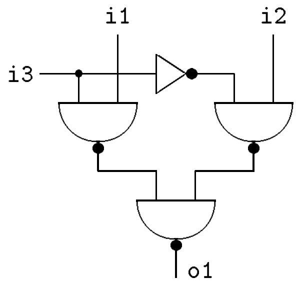

Figure 5: Multiplexer

the overall circuit structure. (For example circuits ia64 and fa16 both mainly consist of multiplexers but their speed-up factors 18.3 and 3.8 are quite different.)

Acceleration of simulation runs:

1. In our approach, internal signals of complex cells are only used temporarily and can therefore be hold in registers. Thus, compared to a gate-level simulator, COFS saves a store operation after each evaluation of such a signal since it needs not be rewritten in main memory. Consider for example the code that is generated for a full-adder as described in Section 4. There, `r1` and the subexpressions `i1 & i2` and `r1 & i3` correspond to such internal signals. (A similar effect is exploited for the acceleration of simulation code in [BCRR87].)

2. In COFS there is no need to generate code for a cell in such a way that it reflects its gate-level realization. Rather, any code can be used as long as it computes the correct logic function. For example in MOS technologies inverting logic gates (NAND, NOR) are preferred to non-inverting ones (AND, OR) because they need less area and switch faster. However, on simulating them, one has to perform the corresponding non-inverting logical operation followed by an inversion since most programming languages only dispose of non-inverting logical operations. Therefore, a multiplexer which is composed as shown in Figure 5 is better simulated by

   ```
   o1 = (i1 & i3) | (i2 & (~i3));
   ```

   than by

   ```
   o1 = ~((~(i1 & i3)) & (~(i2 & (~i3)));
   ```

   which is the one-to-one correspondence to Figure 5. This saves three inversions for the evaluation of each multiplexer.

   XORs can even be simulated by one instruction since the bitwise XOR of two words is a machine instruction.

The acceleration of simulation runs is shown in Table 4. Times required for a fault free simulation of the circuits with complex cells are opposed to those of their expanded counterparts. Both effects (acceleration and reduced number of simulation runs) are summarized in Table 5 which gives running times for complete fault simulations. The additional running time for the complicated fault injection phase is more than outweighted by these effects. In nearly all cases, there is a considerable speed-up. Again the speed-up factor rises with increasing complexity of the basic cells. The circuits c6288 and c7552 which are only built from simple gates have the smallest speed-ups. The relation between the times needed for fault insertion and for simulation depends on circuit complexity. Thus for a small circuit as c499 with only 202 complex cells time savings are smaller than for the larger circuits as for example im16 which consists of 1371 cells. The exact speed-up depends not only on circuit size and cell complexity but also on structural properties of the circuit. E.g. for large pattern numbers , it is important where faults with low detection probabilities are located within the

circuit	simulation time		speed-up
	simple gates	complex cells	factor
c499/c1355	361 μs	135 μs	2.7
ia64	5110 μs	776 μs	6.6
fa16	2221 μs	620 μs	3.6
im16	14514 μs	2458 μs	5.9
fm16	9690 μs	1072 μs	9.0

Table 4: Times taken for one simulation

circuit. The closer they are to the PIs, the higher is the time saving due to an avoided simulation. This explains for ia64 why the overall speed-up factor for 30016 patterns is "only" 43.7 while the corresponding factor for the reduction in the number of simulations is 195.1.

6 Conclusions

In this paper, we have shown how to modify the compiler driven fault simulation algorithm of [DG87] to cope with arbitrary sets of basic cells and combinational fault models for these cells. It was shown that this generalization of the algorithm even results in a speed-up for the special case of stuck-at faults. Thus, the results are encouraging for further research in this direction. (A test generation algorithm supporting the fault modeling technique proposed in this paper is currently under development. Both algorithms will be integrated in the design system CADIC [BHK*87] as components of an interactive tool for test generation. First steps in this direction have been described in [BBH*90].)

Let us conclude by some critical remarks about compiler driven fault simulation. Of course, compiler driven simulation has the advantage of easy implementation. The generation of the most critical part of the algorithm (the software routine which simulates the circuit) is straightforward. On the other hand, there is one major drawback. For large circuits, the simulation routine gets extremely large. Thus, compilers or linkers may not be able to handle it. (For example we can not use the fault simulator on our VAX 11/780 since the C-Compiler of this machine supports only case statements of limited size.) The modifications proposed in this paper are a first step to reduce the size of the generated program but further research concerning this topic seems necessary. (For example hierarchical representations of circuits could be utilized to generate very compact simulation routines [KS91].)

7 Appendix: Cell oriented fault injection

Figure 6 gives a detailed description of the fault injection phase in COFS. It is assumed that the sets $OUTPUTS_t(.)$ are stored as linked lists. They can be processed by the functions *first()* and *next()*. For a set S and any $s \in S$, *first(S)* returns the first element of S and *next(S,s)* the successor of s in S. In the implementation of COFS, the sets $FAULTS_t(.)$, UF, MBP, and $DIFF$ are bit vectors. Therefore, the set intersections in lines (4,20,37) can be performed efficiently by bit parallel logical AND operations. Moreover, the following actions are done word-parallelly as well:

- identification of matching bit positions (6-10),
- overwriting of output signals (21-23),
- determination of changes at POs (30-34),
- removal of covered faults from the fault list (38-40).

8 Acknowledgements

The authors gratefully acknowledge Prof. Dr. Günter Hotz for his support and encouragement.

circuit	patterns	running time		speed-up	fault
		line oriented	cell oriented	factor	coverage
c6288	224	8.5 s	3.5 s	2.4	99.6%
	1024	9.7 s	4.8 s	2.0	99.6%
	30016	58.3 s	54.6 s	1.1	99.6%
	192†	8.3 s	3.4 s	2.4	99.6%
c7552	224	23 s	12.5 s	1.8	89.9%
	1024	43.2 s	23.1 s	1.9	91.9%
	30016	530.9 s	321.6 s	1.7	95.0%
	1024†	43.2 s	23.1 s	1.9	91.9%
c499/	224	0.3 s	0.1 s	3.0	91.5%
c1355	1024	0.3 s	0.2 s	1.5	98.5%
	30016	1.7 s	2.1 s	0.8	99.4%
	448†	0.3 s	0.1 s	3.0	95.1%
ia64	224	19.6 s	1.1 s	17.8	88.6%
	1024	53.7 s	1.8 s	29.8	89.4%
	30016	1214.8 s	27.8 s	43.7	90.0%
	832†	45.7 s	1.7 s	26.7	89.3%
fa16	224	7.9 s	1.1 s	7.2	80.0%
	1024	17.8 s	2.6 s	6.8	83.4%
	30016	276.9 s	36.8 s	7.5	91.7%
	896†	16.3 s	2.4 s	6.8	83.4%
im16	224	112.9 s	2.9 s	39.0	98.3%
	1024	123.3 s	3.4 s	36.3	98.5%
	30016	402.1 s	22.6 s	17.8	98.6%
	448†	112.9 s	2.9 s	38.9	98.3%
fm16	224	114.0 s	5.1 s	22.4	75.2%
	1024	327.0 s	15.0 s	21.8	80.7%
	30016	1436.4 s	94.1 s	15.3	97.7%
	1888†	490.6 s	22.5 s	21.8	88.9%

Table 5: Running times for complete fault simulation in seconds(† dynamic abort)

```
UF ← set of uncovered faults at CUFI;
ITEMS ← ∅; /* ITEMS will contain items that have to be processed */
for all i ∈ INPUTS_t do
     if FAULTS_t(i) ∩ UF ≠ ∅ then
        MBP ← ∅;
        for l from 1 to M do
             if bit position l matches i at CUFI then
                MBP ← MBP ∪ {i};
             fi;
        od;
        if MBP ≠ ∅ then
           o ← first(OUTPUTS_t(i));
           ITEMS ← ITEMS ∪ {(i,o,MBP)};
        fi;
     fi;
od;
while ITEMS ≠ ∅ do
        fault_injected ← false;
        for all (i,o,MBP) ∈ ITEMS do
             if FAULTS_t(i) ∩ UF ≠ ∅ then
                for all l ∈ MBP do
                     set o as output combination of CUFI in bit position l;
                od;
                fault_injected ← true;
             fi;
        od;
        if fault_injected then
           simulation(TSN(CUFI)+1);
           DIFF ← ∅
           for l from 1 to M do
                if a value at a PO has changed in bit position l then
                   DIFF ← DIFF ∪ {l};
                fi;
           od;
           if DIFF ≠ ∅ then
              for all (i,o,MBP) ∈ ITEMS do
                   if MBP ∩ DIFF ≠ ∅ then
                      for all f ∈ FAULTS_t(i,o) do
                           remove f at CUFI from fault list
                      od;
                   fi;
              od;
           fi;
        fi;
        NEWITEMS ← ∅;
        for all (i,o,MBP) ∈ ITEMS do
             o' ← next(OUTPUTS_t(i),o);
             if o' ≠ nil
                then NEWITEMS ← NEWITEMS ∪ {(i,o',MBP)};
             fi;
        od;
        ITEMS ← NEWITEMS;
od;
```

Figure 6: Fault injection in COFS

References

[AS87] Kurt J. Antreich and Michael H. Schulz. Accelerated fault simulation and fault grading in combinational circuits. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):704–712, September 1987.

[BA83] P. Banerjee and J.A. Abraham. Generating tests for physical failures in MOS logic circuits. In *Proceedings of the 1983 International Test Conference*, pages 554–559, 1983.

[BART82] C.C. Beh, K.H. Aray, C.E. Radke, and K.E. Torku. Do stuck-at fault models reflect manufacturing defects? In *Proceedings 1982 International Test Conference*, pages 35–42, 1982.

[BBH*90] B. Becker, Th. Burch, G. Hotz, D. Kiel, R. Kolla, P. Molitor, H. G. Osthof, G. Pitsch, and U. Sparmann. A graphical system for hierarchical specifications and checkups of VLSI circuits. In *Proceedings of the 1st European Design Automation Conference (EDAC90)*, pages 174–179, 1990.

[BCRR87] Zeev Barzilai, J. Lawrence Carter, Barry K. Rosen, and Joseph D. Rutledge. HSS – A high-speed simulator. *IEEE Transactions on Computer-Aided Design*, CAD-6(4):601–617, July 1987.

[BF85] F. Brglez and H. Fujiwara. A neutral netlist of 10 combinational benchmark circuits and a target translator in fortran. In *Proceedings IEEE International Symposium on Circuits and Systems*, 1985.

[BHK*87] B. Becker, G. Hotz, R. Kolla, P. Molitor, and H.G. Osthof. Hierarchical design based on a calculus of nets. In *Proceedings of the 24th Design Automation Conference (DAC87)*, pages 649–653, June 1987.

[BHKS91] B. Becker, R. Hahn, R. Krieger, and U. Sparmann. Structure based methods for parallel pattern fault simulation in combinational circuits. In *Proceedings of the 2nd European Design Automation Conference (EDAC91)*, 1991.

[CA87] Hongtao P. Chang and Jacob A. Abraham. Use of high level descriptions for speedup of fault simulation. In *Proceedings 1987 International Test Conference*, pages 278–285, 1987.

[DG87] Wilfried Daehn and Manfred Geilert. Fast fault simulation for combinational circuits by compiler driven single fault propagation. In *Proceedings 1987 International Test Conference*, pages 286–292, 1987.

[FC88] F.J. Ferguson and J.P. Chen. Extraction and simulation of realistic CMOS faults using inductive fault analysis. In *Proceedings 1988 International Test Conference*, pages 475–484, 1988.

[GCV80] J. Galiay, Y. Crouzet, and M. Vergniault. Physical versus logical fault models MOS LSI circuits: Impact on their testability. *IEEE Transactions on Computers*, C-29(6):527–531, June 1980.

[HSU87] D. Harel, R. Sheng, and J. Udell. Efficient single fault propagation in combinational circuits. In *Proceedings 1987 International Conference on Computer-Aided Design*, pages 2 – 5, 1987.

[KCD90] D. Kannemacher, J. Castagne, and Wilfried Daehn. Vector length control for compiled code event driven pattern parallel fault simulation. In *Kurzfassungen der Beiträge zum 2. ITG/GI-Workshop: Testmethoden und Zuverlässigkeit von Schaltungen und Systemen*, 1990.

[KS85] S. Köppe and C.W. Starke. Logiksimulation komplexer Schaltungen für sehr große Testlängen. In *NTG-Fachberichte*, pages 73 – 80, Baden-Baden, 1985. NTG-Fachtagung Großintegration.

[KS91] R. Kolla and B. Serf. The virtual feedback problem in hierarchical representations of combinational circuits. *Acta Informatica*, 1991. To appear, ca. 13 pages.

[LA83] P. Lamoureux and V.K. Agarwal. Non-stuck-at fault detection in NMOS circuits by region analysis. In *Proceedings 1983 International Test Conference*, pages 129–137, 1983.

[Mal87] W. Maly. Realistic fault modeling for VLSI testing. In *Proceedings 24th Design Automation Conference*, pages 173–180, 1987.

[RGA87] William A. Rogers, John F. Guzolek, and Jacob A. Abraham. Concurrent hierarchical fault simulation: A performance model and two optimizations. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):848–862, September 1987.

[SSB89] Michael H. Schulz, Bernhard H. Seiss, and Franc Brglez. Hierarchical fault simulation in combinational circuits. In *Proceedings 1989 European Test Conference*, pages 33–40, 1989.

[Wad78] R. L. Wadsack. Fault modeling and logic simulation of CMOS and MOS integrated circuits. *The Bell System Technical Journal*, 57, 1978.

[WEF*85] John A. Waicukauski, Edward B. Eichelberger, Donato O. Forlenza, Eric Lindbloom, and Thomas McCarthy. A statistical calculation of fault detection probabilities by fast fault simulation. In *Proceedings 1985 International Test Conference*, pages 779–784, 1985.